Summer 8-2023

Towards Intelligent Runtime Framework for Distributed Heterogeneous Systems

Polykarpos Thomadakis
Old Dominion University, tpolukarpos@gmail.com

TOWARDS INTELLIGENT RUNTIME FRAMEWORK FOR DISTRIBUTED HETEROGENEOUS SYSTEMS

by

Polykarpos Thomadakis
B.S. June 2016, University of Thessaly, Greece

A Dissertation Submitted to the Faculty of
Old Dominion University in Partial Fulfillment of the
Requirements for the Degree of

DOCTOR OF PHILOSOPHY

COMPUTER SCIENCE

OLD DOMINION UNIVERSITY
August 2023

ABSTRACT

TOWARDS INTELLIGENT RUNTIME FRAMEWORK FOR DISTRIBUTED HETEROGENEOUS SYSTEMS

Polykarpos Thomadakis
Old Dominion University, 2023
Director: Dr. Nikos Chrisochoides

Scientific applications strive for increased memory and computing performance, requiring massive amounts of data and time to produce results. Applications utilize large-scale, parallel computing platforms with advanced architectures to accommodate their needs. However, developing performance-portable applications for modern, heterogeneous platforms requires lots of effort and expertise in both the application and systems domains. This is more relevant for unstructured applications whose workflow is not statically predictable due to their heavily data-dependent nature. One possible solution for this problem is the introduction of an intelligent Domain-Specific Language (iDSL) that transparently helps to maintain correctness, hides the idiosyncrasies of low-level hardware, and scales applications. An iDSL includes domain-specific language constructs, a compilation toolchain, and a runtime providing task scheduling, data placement, and workload balancing across and within heterogeneous nodes. In this work, we focus on the runtime framework. We introduce a novel design and extension of a runtime framework, the Parallel Runtime Environment for Multicore Applications. In response to the ever-increasing intra/inter-node concurrency, the runtime system supports efficient task scheduling and workload balancing at both levels while allowing the development of custom policies. Moreover, the new framework provides abstractions supporting the utilization of heterogeneous distributed nodes consisting of CPUs and

GPUs and is extensible to other devices. We demonstrate that by utilizing this work, an application (or the iDSL) can scale its performance on heterogeneous exascale-era supercomputers with minimal effort. A future goal for this framework (out of the scope of this thesis) is to be integrated with machine learning to improve its decision-making and performance further. As a bridge to this goal, since the framework is under development, we experiment with data from Nuclear Physics Particle Accelerators and demonstrate the significant improvements achieved by utilizing machine learning in the hit-based track reconstruction process.

Dedicated to my family.

ACKNOWLEDGMENTS

First, I would like to extend my sincere gratitude to my research advisor, Nikos Chrisochoides, who has provided me with immeasurable direction and guidance throughout my graduate studies. I would also like to thank Gagik Gavalian for introducing me to the field of High Energy Physics and giving me the initiative to study machine learning in this context. I also wish to thank my committee members: Adolfy Hoisie, Charles Hyde, Dimitrios Nikolopoulos, and Jiangwen Sun, whose insights and feedback contributed significantly to the quality of this work.

Thanks to all the members of the CRTC group: Fotis Drakopoulos, Daming Feng, Jing Xu, Angelos Angelopoulos, Kevin Garner, Eleni Adam, Thomas Kennedy, Emmanuel Billias, and Chris Rector, for all the joyful but also stressful moments that we shared. A special thank you to my friend, colleague, and housemate Christos Tsolakis for helping me adjust when I first arrived in the US, his support throughout these stressful years, and the countless hours we spent discussing the different problems we faced in our research. Last but not least, I would like to thank my family. My wife Olga, who believed in me and encouraged me to come to the US and pursue the doctorate degree, my daughter Sofia, whose birth brought so much joy to my life (along with many sleepless nights) and, of course, my parents and brothers for all their love and support.

This research was funded in part by the NASA Transformational Tools and Technologies Project (NNX15AU39A), NSF grant no. CCF-1439079, the Richard T. Cheng Endowment, the U.S. Department of Energy, Office of Nuclear Physics under contract DE-AC05-06OR23177, and the Dominion Scholar Fellowship of Old Dominion University. Experiments were conducted at the Research Computing clusters at Old Dominion University.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

Figure Page

CHAPTER 1

INTRODUCTION

Following the advancements in computer hardware has been a task of increasing difficulty for software developers, especially in recent years. Scientific applications have always struggled for improved computing performance, requiring a tremendous amount of data and time to produce results. To accommodate the needs of such applications, scientists quickly turned to distributed computing (i.e., networks of interconnected workstations) to utilize more computing power and produce results faster.

To this day, developing such complex applications is a difficult task; one has to guarantee correctness while efficiently utilizing the available concurrency of the computing platform. Moreover, applications need to be rewritten based on the peculiarities of the targeted platform and, because of the plethora of heterogeneous devices, might need to be adapted again to run on a platform that uses other types of devices. This continuous code rewriting process quickly gets impractical and not viable; as a result, applications remain sub-optimal due to the effort needed to port them to new architectures and their limited re-usability.

In this work, we introduce a runtime system, Parallel Runtime Environment for Multinode/core Applications (PREMA), that can alleviate the application developer's challenges. Specifically, we present several high-level constructs that allow the application to use the most advanced computing infrastructures and abstract the underlying hardware's details. The issues that this work handles include the following:

- Scheduling: The runtime system can manage the computational work of the application, including both distributed and local computing tasks. It is responsible for assigning work to the available computing units of each computing node based on their pending workload and capabilities. Instead of implementing a single scheduling policy as an integral part of the runtime system, we provide an interface to develop custom scheduling policies to allow the application to choose and develop what is the most efficient.

- Hardware heterogeneity: Utilizing computing units and memories of different types is essential. The runtime can dynamically assign tasks to heterogeneous computing devices, which also requires handling data transfers between heterogeneous kinds of memory. We focus on GPUs and support the implicit and asynchronous transfer of data between CPUs and GPUs without any involvement of the application. Moreover, the application defines computing kernels uniformly that can be run on both CPUs and GPUs. At the same time, transformations in the data layouts can be provided by the application through callbacks.

- Global Namespace: Application data are accessible and uniquely identifiable no matter where they reside in the distributed, heterogeneous system (remote nodes, CPUs, GPUs, memory layers). Completely hiding the cost of accessing data in different memories can lead to sub-optimal performance since the application cannot optimize its access partner. Thus, we provide a two-level abstraction for data; one for fine-grained co-located data, which is essential for tasks requiring tightly coupled synchronization, and one for coarse-grained data where tasks rarely synchronize. This distinction allows the runtime system to monitor the location of the data, perform scheduling, load balance, and move data in the memory hierarchy while letting the application co-locate data and express fine-grained relationships between them.

This work on the runtime system also serves as a platform to generate performance data on various applications for training machine learning models. The future goal, out of scope for this work, is to incorporate the model trained by this exercise into the runtime system to help it make better-informed decisions in scheduling, load balancing, pre-fetching, etc., based on the specific patterns of different applications. Even though there is currently not enough data to train such a deep learning model, since the platform to produce it is still under development, we experiment with machine learning and deep learning approaches to solve other issues in scientific applications, as presented in the second part of this proposed work. Specifically, we apply machine learning in the context of particle accelerators for nuclear physics experiments.

In nuclear physics experiments, measuring scattered particle parameters is the most computationally intensive process. The process relies on measurements of particle tracking detectors to construct a particle trajectory by combining the detected hits and resolving the particle momentum by fitting the trajectory points (using Kalman Filter[1]). This process becomes more time-consuming in high-luminosity experiments since multiple particles are produced from each interaction, and noise is present in particle tracking detectors. Isolating detector hits for each particle trajectory relies on considering each combination of hits that can form a track and then fitting each hypothesis to determine which represents a valid trajectory. This process can be time-consuming, amounting to about 94% of the total data post-processing time.

Recent advances in artificial intelligence and machine learning allow substituting of some of the existing algorithms with predictions from machine learning models. This substitution reduces the code complexity needed to select the right track hit combinations by providing only the most likely track trajectory candidates. This work focuses on the track-candidate identification process

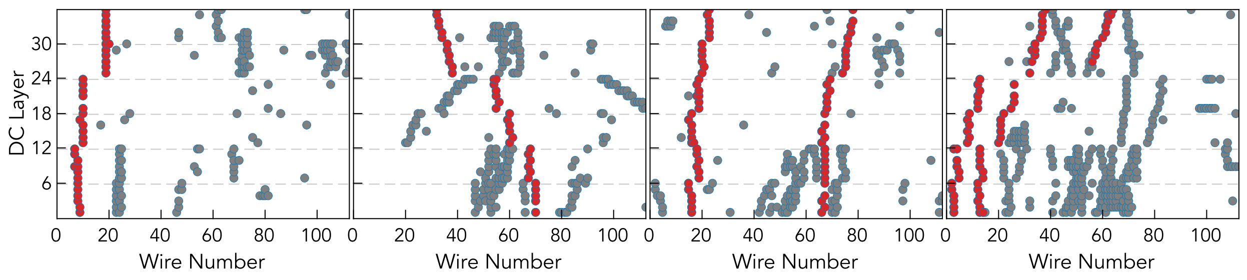


Fig. 1. Example of Drift Chamber data. Each circle represents a wire with hits identified as belonging to a track by the tracking algorithm annotated with red. Each plot presents data from one sector from different events. Cases with one and two tracks in one sector are shown.

for the CLAS12[2] detector at Jefferson Laboratory (JLab), Newport News, Virginia. Examples of reconstructed tracks (for one sector) are shown in Figure 1. The detected hits in drift chambers are presented with gray points, while hits of a valid track are red. Each reconstructed track has an associated cluster in each super-layer (super-layer boundaries indicated by dashed lines), and plots with 12 clusters are events in which two tracks were reconstructed. Using machine learning, we filter the candidates that represent a possible good track and have the tracking algorithm analyze them. Filtering the hits to analyze leads to tracking code speedup and possibly much simpler and maintainable code for tracking candidate selection. To train our model, we use data where the conventional tracking algorithm has already isolated the good tracks.

The technical contributions of this work include:

In Parallel Runtime Systems

- A runtime system that virtualizes hardware address spaces by exposing a uniform programming interface that allows developers to seamlessly develop irregular applications without the need to capture low-level architectural details. It also provides implicit two-level scheduling and load balancing in distributed memory environments while simplifying experimentation with different scheduling policies and the implementation of custom ones.

  Publication:

  P. Thomadakis, C. Tsolakis and N. Chrisochoides. "Multithreaded Runtime Framework for Parallel and Adaptive Applications". *Engineering with Computers*, 2022. Impact Factor: 8.0

- The system is integrated with lightweight preemptable threads that simplify latencies tolerance, provide the means to implement high-level group communication constructs for ease of use and allow the utilization of task and data over-decomposition to enhance the runtime system's ability to handle workload distribution, through a fine-grained tasking framework.

Publication:
P. Thomadakis and N. Chrisochoides. "Runtime Support for Unstructured Exascale-era Applications". *Journal of Supercomputing*, 2023. IF: 2.5

- A heterogeneous tasking framework for the development of performance portable applications that is integrated with PREMA to scale applications from single-core to multi-core, multi-device (CPUs, GPUs), distributed platforms efficiently, without any code refactoring.

  Publication:
  P. Thomadakis and N. Chrisochoides. "Runtime Support for Efficient Handling of Multi-device Heterogeneous Platforms". *Journal of Supercomputing*, Submitted. IF: 2.5

In Machine Learning

- An MLP network classifier, implemented as part of the CLAS12 reconstruction software, providing tracking code with recommended track candidates. The classifier achieves accuracy greater than 99% and results in an end-to-end speedup of 35% compared to conventional algorithms.

  Publication:
  P. Thomadakis, A. Angelopoulos, G. Gavalian and N. Chrisochoides. "Using Machine Learning for Particle Track Identification in the CLAS12 Detector". *Computer Physics Communications*, 2022. IF: 4.7

- A Convolutional Autoencoder (CAE), able to denoise the detector observations used for track reconstruction. The algorithm improves tracking efficiency by more than 35% in real data production procedures with nominal conditions and up to 200% in synthetically generated data with high luminosity conditions (90 - 110 nA).

  Publication:
  P. Thomadakis, A. Angelopoulos, G. Gavalian, V. Ziegler and N. Chrisochoides. "Denoising Drift Chambers in CLAS12 Using Convolutional Autoencoders". *Computer Physics Communications*, 2022. IF: 4.7

- A machine learning regressor network capable of reconstructing particle track parameters (particle momentum, and polar and azimuthal angles) with accuracy similar to conventional methods but up to 150 times faster.

  Publication:
  P. Thomadakis, K. Garner, G. Gavalian and N. Chrisochoides. "Charged Particle Reconstruction in CLAS12 using Machine Learning". *Computer Physics Communications*, 2023. IF: 4.7

Apart from the technical contributions, through this work we have also provided educational contributions as well as services to the communities of the respective fields including:

Educational Contributions

- Leveraging our experience in designing and implementing PREMA based on the principle of separation of concerns, we were able to apply the same principles to a distributed mesh generation application (PDR.PODM [3]) that helped us detect and solve errors in the code. Moreover, the integration with PREMA provided significant performance improvements (Section 4.5.8).

- The lessons learned from this experience are now applied to a different distributed mesh generation code to guarantee correctness and improve performance.

  Publication:
  K. Garner, C. Tsolakis, P. Thomadakis and N. Chrisochoides. "Towards Distributed Speculative Adaptive Anisotropic Parallel Mesh Generation". Accepted in AIAA Aviation Forum 2023, San Diego, CA, 2023.

Services

- Compiled and presented a tutorial on how to efficiently use PREMA in the context of distributed memory parallel mesh generation to enhance performance and end user productivity.

  Presentation:
  N. Chrisochoides, C. Tsolakis, and P. Thomadakis. "Parallel Mesh Generation and Adaptivity". *Invited talk in 27th International Meshing Roundtable*, Albuquerque, NM, 2018.

- Integrating the MLP classifier into JLAB's workflow (with Dr. Gavalian), resulted in "... *35% more physics from the same data which would otherwise cost about \$5mil per year*", according to Dr. Weinstein, ODU Eminent Scholar and professor of physics and past chair of the Jefferson Lab Users Organization [4].

- Applying the denoising CAE as a preprocessing step of the MLP classifier, according to Dr. Gavalian, Staff Scientist at JLAB, "... *resulted in a physics outcome increase of 65%-82%*" [5], which can potentially lead to savings of about $10mil per year.

1.1 BACKGROUND

In the past, applications developed for single-processor hardware of their time could still be performant and efficient when running on hardware developed years later. The transistor power reduction afforded by Dennard's scaling allowed manufacturers to drastically raise clock frequencies between generations, which would result in significant speedups for applications without any modifications. Computer networks (or supercomputers) would consist of several uni-processor computing nodes; thus, upgrading to the next generation would automatically improve the performance of applications. However, the interconnecting network would be the performance bottleneck. Accessing remote data would be orders of magnitude slower than accessing local data; thus, applications needed to be optimized to avoid network communications or mitigate their effects by overlapping them with computations. Even today, inter-node communication is a significant overhead for distributed applications and needs to be treated accordingly.

The breakdown of Dennard scaling and the inability to increase clock frequencies significantly caused most CPU manufacturers to focus on multicore processors as an alternative way to improve performance. This architectural change meant that applications could no longer leverage substantial speedups of new generations of processors "for free", even though they would still experience gradual improvements from the growing count of transistors. To take full advantage of the new multicore processors, application developers would need to rewrite their codes, having the target architecture in mind. This shift brought many of the challenges of distributed computing (work distribution, load balancing, data locality, race conditions, etc.) to local processing and created the need for new programming paradigms. Dealing with the issues stemming from parallelism dramatically increased the burden of software developers in every field; combining this change with the existing difficulties of distributed programming resulted in exponentially more complex scientific applications. To

Recent trends in applications utilizing Machine Learning, and data science, along with the decline and foreseeable end of Moore's Law, are quickly leading to an explosion of heterogeneous and application-specific computing platforms. Deep learning is becoming ubiquitous for scientific research and industry; as a result, GPUs are utilized in most fields. This trend has also led companies to design and manufacture their own domain-specific processors (e.g., Tensor Processing Units) to optimize their applications further. Moreover, the decline of Moore's Law accelerates the need for new, heterogeneous resources in processing and memory. To keep up with these architectural advancements, scientific applications must be adapted to this new era of heterogeneous computing platforms that further increases their complexity.

The United States Department of Energy (DoE) is also playing a key role in this transition. To address future challenges in economic impact areas and threats to security, the United States is making a strategic move in HPC—a grand convergence of advances in co-design, modeling and simulation, data analytics, machine learning, and artificial intelligence. To accelerate research, the DoE established the Exascale Computing Initiative (ECI), the major components of which include the delivery of three exascale computing systems and the Exascale Project (ECP)[6], which is focused on delivering specific applications, software products, and outcomes on DoE computing facilities. As part of the ECP, DoE labs have achieved significant improvements on a set of 24 applications in different fields, including, among others, chemistry, data analytics, and earth and space science, that allow for solutions and insights to key national challenges. To enable these improvements in scientific applications, 35 projects are currently in active development to provide advances in mathematical libraries and frameworks, extreme-scale programming environments, tools, and I/O and visualization libraries. Last, the Hardware and Integration (HI) Area is focused on the deployment and integration of applications, software, and hardware innovations for the forthcoming exascale platforms.

Thus, it becomes apparent that we are in the process of a large transition in the capability and complexity of current and future software/hardware technologies. The presented work is a step towards handling the idiosyncrasies of modern, exascale systems efficiently and is part of a greater project, which aims to provide a high-level Domain Specific Language (DSL) to enable the development of highly scalable parallel mesh generation applications. The runtime system presented here will be used as an abstraction layer between the high-level language, the underlying low-level libraries, and the hardware to provide a platform that exposes all the tools required to achieve high performance without dealing with hardware-specific implementations. Figure 2 shows a high-level representation of the layers that constitute the end-to-end software stack of a DSL for highly scalable parallel mesh generation. As presented in the figure, PREMA is the low-level software targeted by the DSL and is responsible for managing different software technologies, memory models, and hardware variations.

The software stack presented in 2 will be used to support CRTC's Telescopic Approach [7],

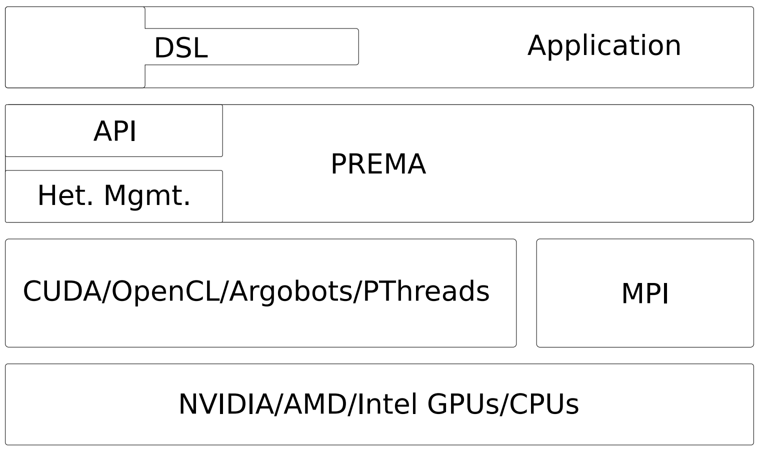


Fig. 2. Software Design modules of the parallel mesh generation DSL project. The focus of this work is on the PREMA runtime framework.

which applies a combination of decomposition techniques for current and emerging architectures with multiple memory/network hierarchies. Parallel mesh generation methods often (over-)decompose the original mesh generation problem into n smaller sub-problems, which are solved (i.e., meshed) concurrently using $n \gg p$ cores [8]. The communication required when solving the generated sub-problems defines the coupling between them. In particular, methods that require a high amount of communication between the different meshing tasks are categorized as Tightly-Coupled. Methods that reduce the coupling of the meshing tasks by grouping them into partially independent domains are characterized as Partially-Coupled. Methods that decompose the problem into (almost) independent tasks that require only minimal (or no) interaction between them are classified as Loosely Coupled. Finally, methods that generate independent meshing tasks where no communication is required are categorized as Decoupled.

This work and the DSL in the future are designed to provide a high-level platform that allows the programmer to develop such a complicated code without explicitly handling all of the scalability and synchronization aspects. For example, the parallel optimistic component is handled by PREMA's intra-node tasking framework that allows applications to define tasks for execution, offloading scheduling, dependency resolution, and load balancing to the runtime system. Moreover, applications can define dependencies between tasks that guarantee correctness and can be utilized to derive efficient scheduling decisions. At the level of Parallel Data Refinement, the different domains created by decomposing the initial mesh are registered with PREMA as mobile objects. Mobile objects constitute the central data abstraction around which many of PREMA's features are implemented. One such feature is the ability to refer to them uniformly through the distributed

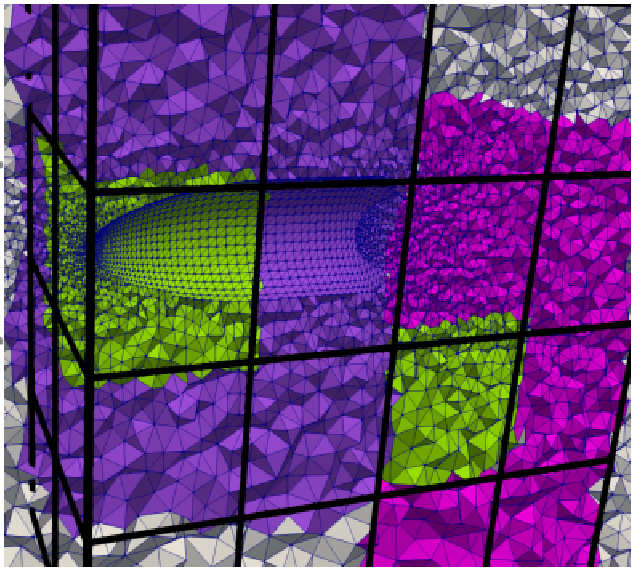


Fig. 3. An example of mobile objects in the context of parallel mesh generation. Each "cube" that decomposes the mesh can be mapped to a single mobile object entity.

system, providing a virtual global namespace. Thus the programming model shifts from Single Program with Multiple Data to a workflow around interactions of the different mobile objects and parallel tasks. Figure 3 shows an example of mobile objects defined in the context of mesh generation; the different cubes represent independent domains, registered as mobile objects, that can be freely migrated in the distributed system while remaining uniformly addressable.

Mobile objects allow PREMA to perform implicit load balancing in the distributed system by migrating them and their workload in response to load imbalance. Mobile objects can also move into the memory of accelerators (GPUs) to perform tasks that can efficiently leverage such devices. Because GPUs are dedicated to certain types of computations and have a limited amount of memory, a finer-grained abstraction has been introduced on top of mobile objects, namely hetero objects. Hetero objects constitute data decompositions of a mobile object that can be referred to globally but are targetted by fine-grained tasks. They provide a high-level abstraction for heterogeneous data tracked by the runtime system and can implicitly move between different memory layers. Their underlying data can be duplicated among multiple devices for performance; coherency is handled automatically, removing the need to reason about their location (in the host or a device) or state before issuing a task.

CHAPTER 2

RELATED WORK

In this chapter, we present work in the field of parallel runtime systems closely related to PREMA. Many of those systems are academic projects that have yet to achieve widespread use; however, they have contributed essential ideas that are utilized to a degree by modern, industry-based systems. Most systems cover a subset of the features of the current work, but no one system provides all features presented.

2.1 DISTRIBUTED MEMORY SYSTEMS

2.1.1 High-Level Language Extension Directives

Directive-based approaches accomplish high performance and parallel system utilization with minimal changes to the serial code. They come as extensions to existing languages that provide information to the specialized compiler in order to map work and data to different processors and coordinate synchronization points. Such approaches often limit the computing and data types to specific, well-defined abstractions that allow the compiler to perform optimizations.

High-Performance Fortran(HPF)[9] extends the Fortran language with directives that the user can use to distribute data to available processors. HPF targets the parallelization of loops. Once the application determines the data distribution, loops are automatically parallelized based on the data distribution. Required communication messages are also injected automatically based on the static inspection of the code. The approach of HPF is quite elegant since the sequential code is only amplified with directives to turn into parallel, but its application is limited to loops and structured data.

ZPL[10] is a high-level language that provides directives similar to those of HPF named regions. Regions define index sets that are used to declare parallel arrays, as well as specify indices for a statement's array references. The regions can then be mapped to a grid of processors while the workload is mapped implicitly based on the data regions accessed. A set of operations is also provided to express the interaction between data mapped to different processors, e.g., scan or reduction operations, that would eventually trigger communications. Like HPF, ZPL is limited to well-defined array-based computations and cannot extend to more complicated patterns.

Jade[11] is a portable, implicitly parallel language designed for exploiting task-level concurrency. Like HPF and ZPL, it extends a standard serial imperative language with constructs to generate parallel computing automatically. In contrast to the systems above, Jade constructs can be applied to any data, and Jade automatically derives the mapping of data to physical cores implicitly. Moreover, dependencies and data allocations can be redefined dynamically. The programming model consists of serial, imperative code with annotations for (1) shared objects, which declare data that can be accessed by different parts of the computation, (2) tasks, which encapsulate the decomposition of the serial code into blocks of computations, (3) access specifications, which declare how each task will access different shared objects. Based on these simple annotations, Jade can automatically generate parallel tasks, assign them to local or distributed cores (message passing is also generated automatically), and provide load balancing. Like the previous systems, Jade is not maintained and updated; thus, it lacks support for accelerators.

2.1.2 Global Array Decomposition

A lot of work has been done using the approach of array decomposition languages. In this approach, the application has an understanding of the structure of the program data through the use of arrays. Having this structured mapping of data, those systems allow the application to easily decompose data, hide inter-node communications, and optimize data distribution and remote access latencies.

Co-Array Fortran(CAF)[12] exposes a Single Program Multiple Data (SPMD) programming model, like MPI, on top of Fortran. Instead of explicitly using messages for inter-node communication, CAF provides support for explicitly describing distributed arrays across all processes (or images as called in CAF), and inter-node data exchanges are performed by explicitly accessing arrays belonging to remote processors. Distributed memory models that expose such an interface that hides message passing are characterized by Partitioned Global Address Space (PGAS). Even though multiple data distribution policies are provided, the data mappings are static and cannot be changed in response to application needs, leading to poor performance in cases where the workload is not uniform.

Split-C[13] is another SPMD programming language that extends C exposing a PGAS memory model that utilizes global pointers and distributed (spread) arrays to hide message passing. Global pointers can point to data on remote, globally addressable data and are used to perform get and set operations in “split-phase” to allow computation and communication overlap. Local globally addressable data are distinguished through the “local” keyword that allows the user to reason about the cost of accessing different pieces of data. Spread arrays can simplify the development of scientific applications that make extended use of large arrays; their distribution can be defined through language extensions on compile time.

Unified Parallel C(UPC)[14] is an offspring of the Split-C language, but UPC goes one step further and completely unifies the interface to access distributed arrays, using the notations of a standard sequential C code, whether local or remote data is accessed. The compiler statically performs the partitioning of the distributed arrays without any application involvement. The unification of the interface to access data tremendously simplifies the development of parallel applications since the application can be written like there is only a unique processor. However, achieving performance is much more challenging because the application cannot distinguish between local (fast) and remote (slow) memory accesses.

Titanium[15] is another array decomposition language that attempts to overcome the aforementioned issue of UPC by explicitly distinguishing between local and remote data. Thus, programmers can reason about the performance overhead each application’s data access would incur. Unlike most of the runtime systems presented so far based on C or Fortran, Titanium is a language extension of Java. However, due to the expected performance degradation of running in a Virtual Machine (VM), it compiles down to native code. As with the previous systems, Titanium can perform several optimizations based on static analysis by the compiler at the cost of only supporting static data distributions. As a result, relocating data based on the workload of the available processors to perform load balance is impossible.

Global Arrays(GA)[16] is another PGAS runtime system that attempts to offer the best features of shared and distributed memory models. It implements a shared-memory programming model in which the programmer manages data locality. This management is achieved by calls to functions that transfer data between a global address space (a distributed array) and local storage. Moreover, the GA model acknowledges that remote data is slower to access than local data and allows data locality to be specified by the programmer and hence be managed. GA is related to the PGAS languages such as UPC, Titanium, and Co-Array Fortran. In addition, by providing a set of data-parallel operations, GA is also associated with data-parallel languages such as HPF and ZPL. However, the Global Array programming model is implemented as a C/Fortran library and does not rely on compiler technology to achieve parallel efficiency. It also supports a combination of task and data parallelism and is available as an extension of the message passing (MPI) model.

However, GA, like most other array-decomposition runtime systems, does not allow data redistribution and is thus prone to load imbalance.

Unified Parallel C++ (UPC++)[17] , as its name suggests, is a modernized, thoroughly reworked runtime from the creators of UPC. UPC++ is moving away from the concept of the implicitly shared PGAS model its predecessor held by making all remote memory accesses explicit and distinct from local accesses. It introduces the notion of distributed objects, collectively allocated objects of the same type and name, one copy for each distinct process in the system. Getting a pointer to remote copies of a distributed object is performed through explicit fetch methods that request a pointer to the copy of a specific rank. Accessing the actual data of such pointers is performed through explicit Remote Procedure Calls (RPCs) or RMA requests; thus, the user can easily comprehend the communication that is triggered by such accesses. Even though UPC++ brings many new features to UPC, it still maps its data statically and uniformly and does not allow remapping.

2.1.3 Place-Based Languages

Place-based programming models abstract the notion of distributed memory spaces using the construct of places. Places are virtual or hardware locations, allowing the application to control data and computation assignments explicitly. In contrast to the SPDM model followed so far, where all processes run the same program, a place only runs some tasks when the application assigns work.

X10[18] is a parallel programming language based on Java that exposes a PGAS memory model but does not follow an SPMD programming model, unlike the previously presented systems. Programmers are given a single entry point from which they can explicitly allocate data and tasks on local and remote nodes. The compiler then takes care of spawning multiple processes and distributing data efficiently. It introduces the abstraction of places that are independent and disjoint pieces of virtual addresses that can map to different pieces of hardware. In-place parallelism is achieved through issuing tasks, while remote computations on other places require explicitly targeting a place. Accessing remote data and passing local data remotely is performed automatically by the language (which means that increasing serialization and messaging operations would be invoked). Array distributions and explicit RDMA are also provided for better handling big amounts of data; however, data redistribution is left to the user and, as a result, load balancing too. Support for GPUs is provided as another implementation of places, but data allocations and transfers have to be handled explicitly; no implicit construct is provided.

Chapel[19] is in many aspects similar to X10. It provides its own flavor for abstract virtual addresses called locales and, as X10, starts from a single entry point from where tasks and data are distributed. Again, a 2-level memory model is provided with asynchronous tasks running in a single locale, while remote tasks are issued by specifying a remote locale. Chapel also allows issuing a task directly on data that might not be local, and it handles delivering the task to the owner locale. A higher-level interface is also available that provides a data-parallel programming model where data can be mapped to locales through a high-level interface, and task execution is handled by a "forall" construct that distributes work according to the placement of data. GPUs are handled uniformly, i.e., data transfers are automatically generated and monitored; however, the user has to assign work to each GPU explicitly. Like X10, data mappings are done statically from the compiler and cannot change, which raises the same issues of load balancing and bad fit for irregular applications.

Habanero Java & C[20, 21] derive from the X10 programming language in an attempt to support both homogeneous and heterogeneous clusters. They extend X10 with hierarchical place trees, which allow places to be built hierarchically instead of a flat model. Moreover, they introduce the phasers synchronization primitive to allow unified collective and point-to-point synchronization with guarantees of deadlock freedom and phase order. Even though these languages extend X10, they still lack the ability to redistribute work efficiently while also requiring explicit data and task management for accelerators.

2.1.4 Actor Model

Another popular programming model for parallel and distributed systems is the actor model. In this model, data and computations are constructed around the abstraction of actors. Actors consist of some data and processing elements while also being able to send and receive messages. Actors can only communicate with each other through messages, and the application workflow is designed as a series of actor method invocations and message exchanges.

Emerald[22] is an object-oriented language and runtime system with the goal of support for distributed object mobility. Emerald objects can freely move within a distributed system and move their computations with them. Many similar features exist with PREMA, including globally identifiable objects, operations that can be invoked remotely, and interfaces to move objects easily between nodes. In contrast to PREMA, Emerald requires its dedicated language and compiler to provide object mobility while leaving the burden of load balancing to the user. The application is given the tools to move objects which can be used to provide load balancing; however, no load monitoring is provided automatically. Therefore, the application must explicitly develop and maintain the load balancing module.

Amber[23] is a runtime system very similar to Emerald, heavily based on object-oriented programming where local and remote data reads and writes are performed by running operations on objects. Like Emerald, it allows object mobility between nodes but does not perform load balancing, leaving this burden to the user. In contrast to Emerald and PREMA, Amber uses globally unique memory addresses by confining the address space each node is allowed to use; thus, all addresses returned from memory allocations in a node should differ from the ones in any other node. Moreover, Amber uses C++ rather than a new language and dedicated compiler.

Orca[24] is another runtime system whose programming model is based on the interaction of objects that can move in the distributed system. Orca incorporates thread parallelism in each node while forcing operations on objects (method calls) to be serialized to maintain coherency. An important difference from Emerald and Amber is the heavy use of replication to improve data access performance. Like Amber and Emerald, it does not facilitate any features for load balancing.

Charm++[25] is the runtime system that is closest to PREMA. It presents an object-oriented programming model for distributed memory machines where interprocess communications are presented as object method invocations on remote machines like Amber, Emerald, and Orca. Charm++ is based on C++ but requires a dedicated compiler that can generate the object marshaling methods, as well as generate the code necessary to invoke methods remotely. Like PREMA, Charm++ facilitates load balancing as one of its features; however, load balancing is initialized based on the user's guidance. Charm++ also provides support for GPU programming; however, submitting tasks to GPUs needs to be explicitly handled by the user, who should allocate memory, issue data transfers, and handle scheduling in case of multiple GPUs.

Tarragon[26] combines elements of static dataflow with object-based operations. Tarragon allows programmers to construct static graphs of tasks; edges between tasks specify where communication between tasks is allowed to occur. At runtime, tasks can then send arbitrary messages to other tasks with which they share an edge. In this way, Tarragon can know the communication pattern of an application statically while still permitting it to engage in a dynamic object-based communication pattern. By leveraging the static knowledge of the communication graph, the Tarragon compiler can optimize many of the communication paths at compile-time.

HPX[27] is a distributed asynchronous many-task runtime system library that exposes a PGAS model where objects are able to migrate between computing nodes. It is designed to fully adapt to modern C++ constructs to allow easier porting of applications. In intra-node concurrency, HPX exposes a task creation interface, while for distributed memory, it introduces the concepts of server and client objects. Server objects are migratable data objects containing information about marshaling their encapsulated data and wrapping boilerplate code required to invoke remote methods seamlessly. Client objects present unique identifiers to server objects that make it easier and safer to utilize them. Even though HPX allows object migrations, it lacks the capability of implicit load balancing, and only recently have efforts started towards this feature. Moreover, issuing work to GPUs is explicit, requiring the designation of a specific GPU, the creation of buffers, and the transfer between devices.

2.1.5 Coordination Languages

Coordination languages extend existing languages with a minimal set of functions and a virtually shared address space that allow different processes to synchronize and communicate. Applications written using coordination languages can run on a uni-processor, a multi-threaded multiprocessor, or a network without changes. Processes in this model are decoupled from each other; communication is achieved through the virtual shared memory, where processes define templates for data produced or consumed.

Linda[28] is a coordination language that enhances existing languages with a collection of primitives for interprocess communication. The central abstraction of Linda is the Tuple Space (TS), a distributed shared memory that processes can use to communicate and synchronize with each other. Linda's primitives operate on the shared tuple space to deposit, write, or read tuples whose content matches a template specified in the primitive. A tuple can contain almost any primitive of the targeted language e.g., values, class objects, functions to be called etc., which are tagged with a string literal and can be retrieved using a primitive (e.g. rd for read) with a respective pattern. Linda provides a clean interface to express interprocess communication that abstracts any implementation details from the user. Process creation, message passing, load balancing, and scheduling are all handled by Linda's runtime and compiler. Despite its beautiful interface, implementations of Linda have fallen short of providing the desired performance. However, Linda is still used in fields where a few processors are enough to speed up their problems.

Concurrent Collection (CnC)[29] is another coordination language inspired by Linda that provides implicit parallelism through high-level operations along with semantic ordering constraints. A CnC program consists of a graph where the nodes can be step, data, or control connections, and the edges represent producer, consumer, and prescription dependencies. These nodes are described in a CnC dialect that describes the flow of data between different application steps along with the code that constitutes a step. Once those specifications are given for an application, the runtime system implementation can use them to implicitly map data to different hardware threads/cores/nodes and take care of the scheduling. Distributed and heterogeneous computing is handled similarly and using the same specifications. Like Linda, CnC provides an attractive abstraction for programmers, but the actual performance of the implementation can vary significantly.

2.1.6 Irregular Fine-Grained Access

The following runtime systems have been developed for very specific purposes and are optimized to handle tasks with very fine-grained data accesses.

Standard Template Adaptive Parallel Library (STAPL)[30] is a parallel library designed as a superset of the ANSI C++ Standard Template Library (STL). It is sequentially consistent for functions with the same name and executes on uni- or multiprocessor systems that utilize shared or distributed memory. STAPL provides an SPMD model of parallelism and focuses on irregular programs that utilize data structures like linked lists and graphs. STAPL delivers a set of parallel containers, algorithms, schedulers, and executors. Data of parallel containers can be partitioned using high-level constructs that use the interface of iterators to be defined and also for iteration during task executions. Dependencies between tasks are provided through the ranges that tasks access, and the required synchronization or message passing is derived automatically. To efficiently use this model, the user needs to explicitly provide the task dependency graph, in terms of ranges, before issuing a parallel operation. Also, once data and tasks have been mapped to specific locations, they cannot be redistributed.

Galois[31] is a graph processing tasking runtime that provides abstract set iterators, giving the application/user the ability to extract parallelism out of the worklists of a sequential application. Custom data structures and a runtime scheduler are responsible for detecting and recovering unsafe accesses to shared memory, allowing for speculative execution.

2.1.7 Task and Graph-Based

A more recent approach to explore concurrency is through the use of tasks where the dependencies between them are defined through a directed acyclic graph. In contrast to previous models where parallelism is explored mainly through data decomposition, the task-based model is oriented around tasks that can be stolen/shared across the distributed system along with the data they target.

ParSec[32] is a Directed Acyclic Graph (DAG) scheduling engine, where the nodes of a DAG are sequential computation tasks, and the edges are data movements between tasks. Programmers must explicitly provide a DAG representing their application, while ParSec is responsible for mapping the DAG to actual data at the runtime. It supports shared and distributed memory heterogeneous platforms where communication requirements are handled automatically. Even though it supports high-level data distribution constructs, moving data between distinct computing nodes is not allowed. ParSec is very powerful for more structured applications where communications and task interactions are known statically; however, it is unsuitable for dynamic and irregular workloads where load imbalance could be an issue.

StarPU[33] is another task-based heterogeneous runtime system that allows dynamically constructing a DAG with dependencies to represent workflow. Data transfers between different memory devices are performed implicitly while StarPU maintains consistency. The application needs to map data to computing nodes, then StarPU can infer message-passing needs between tasks that target the data mentioned above. For shared memory, StarPU provides a powerful interface to declare performance models for the different tasks and uses those to perform better scheduling decisions. However, this is not the case for distributed memory, where StarPU does not include any kind of data redistribution; thus, the quality of load balance depends upon the initial mapping given by the user.

Legion[34] is a high-level task-based heterogeneous runtime system for distributed memory architectures. It is a very powerful system that separates the application workflow from mapping computations and data to hardware. It utilizes the primitive concepts of logical regions as the piece of data to present data organization and expose task-data interactions. It shares many common ideas with StarPU in this aspect to allow implicit data movements and to handle heterogeneous platforms but extends upon that with scheduling on the distributed memory. Even though it is a powerful system, it requires a lot of effort and code rewriting to port existing applications on top of it. Moreover, its design better conforms to structured data.

TaskTorrent[35] is a lightweight distributed task-based runtime system expressing task dependencies and interactions as a DAG. In contrast to systems like Legion, StarPU, and ParSec, it expresses inter-node communication explicitly through one-sided active messages. Like ParSec, in contrast to StarPU and Legion, it utilizes a Parametrized Task Graph approach in which the DAG is not handled as a whole. Tasks signal their completion to activate other tasks, thus, discovering the DAG piece by piece. TaskTorrent presents good scalability compared to StarPU but does not support heterogeneous systems.

Open Community Runtime (OCR)[36] is a fine-grained, asynchronous, task-based, event-driven runtime. Its programming model consists of a set of primitives that enable a directed acyclic graph's static or dynamic construction. Events are the primitives that encapsulate dependencies between tasks and data, and any interaction between the two constructs is monitored by a respective event. Its abstract programming model and platform-agnostic interface were designed as a leading effort for the exascale computing era; however, there were no complete implementations to showcase its performance.

2.2 SHARED MEMORY SYSTEMS

2.2.1 Pragma-Based

In pragma-based models, a sequential code is decorated with directives that allow the compiler to generate code that spawns threads and assigns them computations. This approach started with a focus mainly on parallelizing simple loops but has been extended since to general tasks and even offloading data to accelerators.

Open Multi-Processing(OpenMP)[37] is a pragma-based multi-threading programming model and API for shared memory architectures, extending C, C++, and Fortran. OpenMP provides a variety of parallel computing paradigms, including automatically parallelizing loop iterations and task parallelism. More recently, offloading computations to accelerators was included; however, the consistency between host and accelerator copies of the data must be explicitly handled by the user before issuing tasks on the same data with different targets. While its performance to parallelize loop iterations (which was its original purpose) is excellent, the performance of its other models varies to a high degree. Moreover, since its implementation depends on the compiler provider, its performance can differ substantially based on the compiler.

SSMP[38] is a shared memory, task-based, runtime system accompanied by its dedicated compiler. SSMP provides the ability to extend C code with pragmas that can designate functions that should run as tasks and decorate their parameters with their access type (in, out, inout), which can be used to infer dependencies. The scheduler processes these dependencies at runtime to infer parallelism between them and schedule them accordingly. This pattern is efficient for inferring task dependencies in a shared memory node; however, it does not support scheduling on GPU devices or distributed systems.

2.2.2 Performance Portability Centric

The following systems come from the increasing need to utilize computing units of different types. Their focus is to allow the programmer to write a single code that can run on multiple devices while maintaining performance. They provide abstractions that hide the specifics of the underlying implementation and thus can be easily tuned among different device targets.

RAJA[39] is a suite of tools for developing performance-portable applications that can run on heterogeneous applications. It consists of four pieces of software intended to separate the concerns for different issues in heterogeneous platforms. RAJA provides an interface of C++ templates that capture platform-specific constructs and allow users to write kernels with a single source that can run on different devices. RAJA can then map those kernels to the specific platform implementation requirements and handle data transfers. Even though it abstracts the application code to a high degree, users need to explicitly issue tasks to distinct accelerators to utilize multiple devices and take care of tasks that depend on each other.

Kokkos[40] implements a programming model in C++ for writing performance portable applications targeting heterogeneous systems. It provides the same capabilities as RAJA, including abstractions for data objects, layouts, and portable kernels using a single source code. In addition to features also provided by RAJA, Kokkos provides the ability for task-based parallelism allowing for the creation of task dependencies through the construction of DAGs. Moreover, it has more default implementations for decisions that depend on the application, e.g., automatic data layout formatting, kernel block size inference, etc. Like RAJA, Kokkos does not explicitly handle multi-GPU scheduling; the application must submit tasks to different GPUs to saturate multi-GPU platforms.

SYCL[41] is an evolution of the OpenCL standard presented as a higher-level programming library to accommodate various hardware accelerators. In contrast to OpenCL, SYCL is a fully compliant C++ library that provides all the capabilities of OpenCL through higher-level constructs. It also extends it with support for implicit asynchronous executions and data transfer overlapping. Even though it eases application development for multi-accelerator platforms, it still leaves the responsibility for scheduling to different devices on the user.

2.2.3 Recursive Tasking

Recursive tasking systems provide a simple interface to express fine-grained tasks that the underlying runtime system can potentially run in parallel. Such systems require the application to provide where parallelism and synchronization happen. Moreover, to achieve better performance, they rely on recursive task creation; applications should generate tasks in a tree-like flow to hide latencies better.

Cilk[42] is an extension of the C language that allows function calls to be spawned as independent, potentially parallel tasks. Cilk was one of the first systems to present the fork-join programming model abstracting the handling of threads from the application. Dependencies in Cilk are handled by the user and are expressed indirectly by explicitly waiting for previously executed tasks to complete. Cilk was the system that inspired other more modern shared memory parallel systems to arise.

Threading Building Blocks (TBB)[43] is a C++ template library for multi-paradigm parallel programming. TBB's initial conception is based on Cilk's programming paradigm but has expanded its functionality considerably. It provides interfaces for explicit task creation where the dependencies are managed by the user, graph-based task dependencies where the runtime guarantees the partial order of task execution, as well as a set of common parallel operations applied on loops. In addition, TBB comes with a plethora of data structures optimized for parallel processing. Support for GPUs is currently not part of TBB.

2.2.4 Low-Level Computing Libraries

Low-level computing libraries provide the barebones for executing parallel applications. These libraries are utilized as the bases of all systems providing support for GPUs and a small set of highly optimized features.

Compute Unified Device Architecture (CUDA)[44] is a low-level parallel computing platform for developing applications targeting NVIDIA GPUs. CUDA is implemented as an extension to C/Fortran, providing parallel computing specialists easier accessibility to GPU resources compared to graphics programming languages like OpenGL. CUDA provides the barebones to submit computing kernels to the GPU, data transfers between the CPU and the GPU, and mechanisms to develop asynchronous executions managed by asynchronous events. As a low-level language, it requires significant programming effort to fully utilize a GPU.

Open Computing Language (OpenCL)[45] is a low-level programming standard for developing applications that can run across different heterogeneous platforms, including CPUs, GPUs, DSPs, FPGAs, and other hardware accelerators. It is implemented as a library for C with small additions that can be observed in its computing kernels. OpenCL utilizes Just In Time (JIT) compilation to compile computing kernels based on the targeted device, while host-side interactions are provided through C/C++ libraries. OpenCL offers the primitives CUDA offers and, thus, also requires significant effort to manage high performance on multi-device environments.

2.2.5 Summary

In this chapter, we have presented several runtime systems related to PREMA. We have presented systems that focus both on shared and distributed memory following different approaches to achieve scalability. As Table 1 summarizes, most of these systems have not been designed having data redistribution in mind, causing load imbalance to be a big issue. Support for GPUs also varies and, in many cases, even if it exists, requires the user to handle data transfers, task issuing, and synchronization. Our work covers the above properties leveraging from the existing works and extending them to provide a feature-rich system targeting dynamic applications.

TABLE 1

Software systems that support functionality similar to the PREMA system.

System Name	Global Namespace	Data Migration	Shared Mem. Load Balance	Dist. Mem. Load Balance	Custom Policies	Heterogeneity Aware
HPF	●					
ZPL	●					
Jade	●	●	●	●		
CAF	●					
Split-C	●					
UPC	●					
Titanium	●					
GA	●		●			
UPC++	●		●			
X10	●		●			
Chapel	●		●			●
Habanero	●		●			
Java & C						
Emerald	●	●				
Amber	●	●				
Orca	●	●				
Charm++	●	●	●	●	●	
Tarragon	●		●			
HPX	●	●	●			
Linda	●		●			
CnC	●	●	●	●		
STAPL	●		●			
Galois	●	●	●	●		●
ParSEC	●		●	○		●
StarPU	●	○	●			●
Legion	●	●	●	○	●	●
Task Torrent	●		●			
OCR	●	●	●	●		
OpenMP			●			●
SSMP			●			
RAJA			●		●	●
Kokkos			●	●	●	
SYCL			●			●
Cilk			●			
TBB			●			
CUDA						
OpenCL						●
PREMA	●	●	●	●	●	●

●: Feature is supported. ○: Support for the feature is limited.

CHAPTER 3

DESIGN FOR MULTI-CORE ARCHITECTURES

The ever-increasing intra-node parallelism exacerbates the complexity of developing efficient applications. One can ignore the hardware structure and use message-passing to explore concurrency at both inter-/intra-node levels (by mapping one MPI rank per core) to simplify the programming model. A significant issue in this approach is that cores on the same node execute on distinct virtual memory address spaces. Scheduling and load balancing require increased overheads for redistributing data and workload due to data marshaling and copying, while decision-making implies expensive communication and coordination. Another approach is a hybrid programming model where inter-node concurrency uses message passing (e.g., MPI), and intra-node concurrency utilizes multi-threading (e.g., OpenMP). This approach can improve performance by allowing cores in the same node to share workload and data directly. However, it requires adapting the application to a new, more complex programming model. Applications must be adjusted to handle synchronization between threads, maintain correctness, and avoid issues like race conditions and deadlocks.

This work presents PREMA, the Parallel Runtime Environment for Multinode/core Applications [46, 47], originally introduced in [48]. PREMA implicitly handles inter- and intra-node scheduling and load balancing and facilitates an approach between the two (i.e., flat-MPI and hybrid programming paradigms), utilizing their advantages and overcoming their disadvantages. It implements one-sided communication and remote method invocations similar to Active Messages [49], which are enhanced with a globally addressable namespace, transparent object migrations, and message forwarding, as opposed to static global address space implemented in languages like Split-C [13] and UPC [14]. On top of this infrastructure, it provides a set of load balancing policies and an API that allows for easy-to-use/implement custom inter/intra-node scheduling and load balancing without modifying the application. All of the above features are supported by a low-level multi-threading subsystem.

PREMA adheres to two principles. First, application data are encapsulated in *mobile objects*. Second, communication patterns follow an object-oriented paradigm where mobile objects communicate with each other directly. Thus, the runtime system abstracts the underlying structure of hardware, processing elements, and memory spaces. Following this Mobile Object Driven (MOD) programming model, it can utilize resources across shared and distributed memory and perform efficient scheduling and load balancing while presenting a uniform programming interface that implicitly handles issues related to concurrency. Messages that trigger remote or local method invocations are assigned access privileges. Access privileges express how method invocations may use a mobile object (exclusively, shared), providing information about data dependencies, which is used to maintain correctness and extract parallelism.

The runtime system offers implicit, 2-level (inter- and intra-node) scheduling and load balancing, as opposed to the previous version [46] that only provided support at the inter-node level. Shared memory scheduling and load balancing are provided through the parallel execution of computations on independent objects or non-conflicting computations on the same set of mobile objects by multiple threads. At the distributed level, load balancing is handled by transparently migrating mobile objects between nodes. The load balancing and scheduling strategies are isolated in a separate module, allowing users to develop their application focusing only on correctness. A user might experiment with different strategies to improve performance without modifying the application at a later optimization step.

The major contributions in this chapter are as follows:

- An efficient and scalable runtime system for exascale-era platforms, providing one-sided communication, remote method invocations, and a global namespace on top of transparent object migrations for implicit shared and distributed memory load balancing and scheduling through a uniform interface. This interface allows developers to encode algorithms efficiently without the burden of capturing low-level architectural details while allowing experimentation with different scheduling policies and implementing custom ones.
- An evolutionary methodology for porting legacy MPI systems to multi-core platforms based on the principle of separation of concerns and the lessons learned from this transition

3.1 SOFTWARE STACK AND PROGRAMMING MODEL

3.1.1 Software Stack

PREMA consists of three software layers following the principle of separation of concerns: the Data Movement and Control Substrate, the Mobile Object Layer, and the Implicit Load Balancing layer. This section presents the functionality provided by the three layers.

The *Data Movement and Control Substrate* (DMCS) [50], originally introduced in [51] acts as a thin layer that abstracts the underlying hardware and the communication library (MPI in this case) from the application or the higher-level libraries that constitute PREMA. This abstraction allows PREMA to be easily ported to a different computing environment by only adapting the DMCS layer. Apart from that, DMCS offers a message-driven programming model by implementing one-sided communication and remote method invocations, similar to the *Active Messages* [49] paradigm. Methods that are supposed to be available for remote invocation are referred to as *remote handlers* in the context of the runtime system, and they need to be explicitly registered as such before being used. This paradigm allows the runtime system to hide remote data access latency behind computations performed by both the system and the application.

The *Mobile Object Layer* (MOL) [52], introduced in [53], builds on top of *DMCS* and extends it with a globally addressable namespace while it introduces the construct of *mobile objects*. A *mobile object* is an abstract, location-independent container implemented by the runtime system to store application data. Data defined as mobile objects can be freely moved to remote destinations while remaining addressable from any node in the system through their unique identifiers, the *mobile pointers*. This functionality is made possible by extending the remote method invocation capabilities of DMCS to target mobile objects directly, wherever they might reside. Thus, applications are encouraged to follow a programming model oriented around mobile objects and invoke computations through messages without knowledge about their locations. We refer to this programming model as Mobile Object Driven (MOD).

A distributed directory maintains the last known locations of all the mobile objects in the local and remote nodes. When sending a message to a mobile object, the object's last known location is retrieved from the sending node's directory, and the message is transmitted there. The target node might no longer hold the mobile object, so the message is forwarded based on that node's directory information. Once a forwarded message arrives at its destination, an update message is sent back to the original sender, updating the object's last known location. There are more location-updating policies available; the *lazy updates* presented in [54] is the one that exhibits the best performance.

The *Implicit Load Balancing Layer* (ILB)[46] makes use of of both DMCS and MOL to provide transparent and implicit data and load migration in the event of load imbalance detection. Based on the observation that no single load-balancing algorithm is optimal for all platforms and applications, the ILB's scheduler is implemented as a separate module that allows plugging-in new policies without any modifications to the application. This design allows developers to experiment with different methods without affecting their application code. Moreover, for more experienced users, ILB provides an API to develop custom load-balancing policies.

In ILB, all mobile objects are associated with pending computations, constituting their load. Thus, migrating mobile objects will implicitly migrate workload from one node to another, resulting in load balancing. Since mobile objects contain user-defined data, the application must provide callback functions that let the runtime system pack, unpack, and retrieve their sizes to migrate them between nodes. Another set of callbacks may also be provided to pass information about the cost of migrating a mobile object and dynamically calculate its load with respect to its pending handlers.

3.1.2 Programming Model

PREMA encourages the use of the MOD programming model. It exposes a data-centric design where communications and computations happen between mobile objects rather than processors. Parallelism is achieved by executing handlers simultaneously on multiple nodes, cores, and mobile objects. Applications have their data broken into N chunks, encapsulated into mobile objects, where $N \gg P$ and P is the number of processing elements. This technique is known as over-decomposition and allows greater flexibility in intra/inter-node scheduling and load balancing.

A typical application starts by performing some pre-processing steps and then over-decomposes its data. Then it creates mobile objects for each chunk of data, defines (de) serialization functions, and optionally distributes them to available nodes. The core of the application logic is then expressed through messaging among the mobile objects. Message handlers include computational tasks, the creation of new mobile objects, and data transfers. By adhering to this programming model, task executions are addressed to individual mobile objects, whether they reside on the local or a remote node, abstracting the actual data/hardware mapping from the application. Data encapsulated in a single mobile object are guaranteed to be local, allowing the application to access them directly in the context of handler execution.

Throughout the application's execution, PREMA handles scheduling and maintains the location of mobile objects. Moreover, it monitors nodes' workload, initiates load balancing when necessary, and guarantees that messages are delivered to their destination, whether local, remote or in the migration process. The application is unaware of these operations and is developed as if all mobile objects were locally accessed through a uniform API.

3.1.3 Adapting to Application Needs

Even though we use scalable parallel mesh generation applications as a case study for evaluation, PREMA is designed as a general-purpose runtime library. PREMA supports different execution models [55, 56] and has been tested with different applications. In addition to several parallel mesh generation methods [56], PREMA is used to implement a seismic wave simulation application (section 3.3.3). In earlier efforts, we used DMCS and MOL (PREMA's low-level communication substrates) to develop an N-body simulation code [57]. These use cases include both legacy applications following a black-box approach [56] and new applications developed with

PREMA in mind (presented in this work). In both cases, PREMA successfully scaled the applications using different scheduling models like the master-worker model and diffusive load balancing.

PREMA offers multiple abstractions at all levels of the runtime system software stack. One can choose to use a subset of them depending on the specific needs of their application as well as their familiarity with the runtime system and expertise with load balancing policies. PREMA's design allows using a subset of its capabilities when there is no need for its extra features. For example, users needing an active messages library can just utilize DMCS without building and studying the other software layers. Later, one might need the global namespace and messages provided by mobile objects. If later implicit load balancing is desired, the application can be easily extended to utilize ILB without much effort. Moreover, a user can choose how the workload of a mobile object is calculated by providing a callback function or a static value and explicitly disable or re-enable distributed load balancing by calling the respective function *ilb::enable_dist_balancing(bool).*

Another feature that gives more flexibility to PREMA is the ability to use, adapt, and modify load-balancing policies that ship with PREMA or implement new ones through the provided API. Since there is no holy grail for load balancing that fits all applications and platforms, this interface can be used to experiment with different methods without the need to touch the application's code. For example, an application expert may prefer to utilize existing load balancing strategies and choose the one that gives the best results for the specific needs. Such a user might feel more comfortable only experimenting with high-level parameters provided by existing strategies to improve performance. An advanced user might try to extend an existing policy to extract more performance, while an expert in load balancing might want to develop a new one. Thus, PREMA can accommodate users with different needs, skills, and experiences.

Figure 4 shows the abstract class (API) that a user can extend to implement virtually any new policy. Four methods need to be implemented to allow the new policy to communicate with PREMA and receive updates about the system state. In summary, these methods provide: (1) the trigger for starting a new load balancing phase (*dist_balance()*), (2) updates from PREMA to the policy regarding the load of each mobile object (*notify()*), (3) abstractions to implement shared memory scheduling (*push(), pop()*). In section 3.1.4, we present the implementation of two simplified load balancing policies, a master worker and a diffusive scheme.

3.1.4 Scheduling Policies Examples

This section aims to present the (simplified) implementation of two load balancing strategies, Master Worker and Diffusion, utilizing PREMA's scheduler API. The two strategies have been used to scale different irregular applications [56] while significantly reducing code complexity

```
class ilb::scheduler
{
public:

    // Initialize required structures here
    scheduler() {};
    // Free structures here when application terminates
    virtual ~scheduler() {};

    // This method is periodically called by PREMA to check and start distributed load balancing
        when needed
    virtual void dist_balance() = 0;

    // Notify scheduler about a change to mo's load and update node level load
    virtual void notify(ilb::mobile_object mo) = 0;

    // Thread with ID thread_id tries to get some handler to execute
    virtual ilb::handler* pop(int thread_id) = 0;

    // Thread with ID thread_id tries pushes a new handler to the scheduler
    virtual void push(int thread_id, ilb::handler* hdlr, ilb::mobile_object mo) = 0;
}
```

Fig. 4. PREMA's minimal interface to declare a new load balancing policy.

(removing load balancing-related code) and line count (e.g., 1200 vs. 2500 LOC in the first application) compared to the respective MPI implementations.

Master Worker

Figure 5 presents the simplified master-worker implementation. The derived class assigns a single node as the master and defines a load threshold under which a worker node is considered underloaded (line 3). Each node keeps a custom map (provided by PREMA) that holds mobile objects and their workload and tracks the overall node workload. When a worker finds its load under the threshold, it sends a remote request to the master for a new mobile object migration (lines 5-7). If the master holds enough load, it picks a mobile object, packs it, and sends it to the requesting worker (lines 27-30). Otherwise, it requests the worker to wait and pushes its rank to a list of waiting workers (lines 33-34). In the reception of the master's reply, the worker unpacks and installs the packed object to the local node, which updates PREMA about the migration(lines 38-41). If there is no mobile object to unpack, the worker sets a flag that it should wait from the master for a new workload when it becomes available (line 44). In this simplified case, we use a simple list to maintain handler invocation requests and support the *push()/pop()* operations; a more sophisticated implementation could use work pools per thread, per mobile object, or a combination of the two. Method *notify()* keeps the mobile objects - load map and node workload up to date and is called each time the node workload changes.

Diffusion

Figure 6 presents the simplified diffusive scheme implementation. In this scheme, each node assigns a "neighborhood" of other nodes from which it can request workload. In each new load balancing phase, the underloaded node tries to steal from the node with the largest workload in the neighborhood. If no neighbor has enough workload, a new neighborhood is assigned for the next load-balancing phase. In this implementation, *dist_balance()* checks if the node is underloaded and initiates a new load-balancing phase by requesting the workload levels of its neighborhood. Once the underloaded node receives all the responses, it chooses the neighbor with the highest load and requests for mobile object migration or assigns a new neighborhood if not enough workload exists(lines 25-36). The receiver of a migration request picks its mobile object with the largest workload and, if its workload is enough, packs and sends it to the underloaded node. Otherwise, it refuses to migrate any work (lines 39-44). Depending on this response, the requester will either unpack and install the received mobile object or replace the neighbor in the neighborhood set and prepare for a new load-balancing phase.

```
class master_worker: public ilb :: scheduler {
    master_worker( int limit , int master): m_limit( limit ),m_master(master),m_worker_wait(0) {}
    void dist_balance () {
        if (prema::my_rank() != m_master && ! m_worker_wait)
            if ( m_mo_map.get_total_load() < m_limit) // load dropped below threshold ?
                dmcs::send(m_master, work_request); // request work from master
        else while(m_mo_map.get_total_load() > 0 && !m_waiting.empty()) { //while enough load
                int dst = m_waiting. front () ; m_waiting.pop_front () ;
                work_request( dst ); // master sends work to waiting nodes
            }
    }
    void notify ( ilb :: mobile_object mo) {m_mo_map.insert(mo−>get_load(), load);} // update load
    ilb :: handler∗ pop( int thread_id ) {
        return m_hldr_pool. front () ; // pick the next handler in the pool
    }
    void push( int thread_id , ilb :: handler∗ hdlr , ilb :: mobile_object mo) {
        m_hldr_pool.push_back(hdlr) ; // insert a new handler in the work pool
    }
    void work_request( int src ) {
        if (m_mo_map.get_total_load() > 0) { // if enough load
            ilb :: mobile_object mo = m_mo_map.pop(); // master picks mo with largest workload
            void∗ buffer = mo−>pack(); // packs it
            dmcs::send( src , work_request_reply , buffer ); //and migrates it to requesting worker
        } else { m_waiting.push_back(src); // keep track of waiting workers
            dmcs::send( src , work_request_reply , NULL); // put worker to wait
        }
    }
    void work_request_reply ( int src , void∗ buffer ) {
        if ( buffer != NULL) { // master replied with mo (workload)
            m_worker_wait = 0;
            ilb :: mobile_object mo(buffer); // unpack mo
            mo. install () ;                  // and notify PREMA
        }
        else {m_worker_wait = 1;} // wait for master
    }
    int m_master, /∗ master node ID ∗/, m_worker_wait, m_limit; // load threshold
    ilb :: mo_work_map m_mo_map; // map wrt load for mobile_objects, also keeps total workload
    list <handler∗> m_hldr_pool; /∗pending handlers ∗/ list <int> m_waiting; // workers waiting
}
```

Fig. 5. Sample implementation of the Master-Worker model using PREMA's API.

```
class diffusion : public ilb :: scheduler {
    diffusion ( int limit ): m_limit( limit ) {
        m_neighbors = assign_new_neighbors() ; // Pick a set of neighbors
    }
    void dist_balance () {
        if ((m_mo_map.get_load() < m_limit)){ // low workload and no lb pending
            m_levels_recv_cnt = 0; // new lb phase
            for( int n: m_neighbors) dmcs::send(n, load_request ); // Request workloads
        }
    }
    void notify ( ilb :: mobile_object mo) {m_mo_map.insert(mo−>get_load(), load);}// update load
    ilb :: handler∗ pop( int t_id ) { return m_hldr_pool. front () ;} // pick next handler in the pool
    void push( int t_id , ilb :: handler∗ hdlr , ilb :: mobile_object mo) {m_hldr_pool.push_back(hdlr);}
    void load_request ( int src ) {dmcs::send( src , load_request_reply ,m_mo_map.get_load());}
    void load_request_reply ( int src , size_t load){
        m_levels_recv_cnt++; // Count how many neighbors replied
        m_load_to_neigh[src] = load; // Track their loads
        if (m_levels_recv_cnt == m_neighbors.size () ){ // If received load levels from all
            int max_neigh = max_load_neigh(m_load_to_neigh); // neighbor with largest workload
            if (m_load_to_neigh[max_neigh] == 0){ // Found no neighbor with workload
                m_neighbors = assign_new_neighbors() ; // set new neighbors
            } else { dmcs::send(max_neigh, work_request);} // Found workload, request
        }
    }
    void work_request( int src ) {
        if (m_mo_map.get_total_load() > m_limit){ // If I have enough load
            ilb :: mobile_object∗ mo = m_mo_map.top();
            void∗ buffer = mo−>pack(); // pack mo
            dmcs::send( src , work_request_reply , buffer ); // and send it to source
        } else { dmcs::send( src , work_request_reply , NULL);} // Otherwise, send NULL
    }
    void work_request_reply ( int src , void∗ buffer ) {
        if ( buffer ) { ilb :: mobile_object mo(buffer); mo. install () } // unpack mo and notify PREMA
        else { assign_one_new_neighbor(src); } // No mo (worload) received , replace neighbor
    }
    int m_limit, m_level_recv_cnt;
    ilb :: mo_work_map m_mo_map; // map wrt load for mobile_objects, also keeps total workload
    list <handler∗> m_hldr_pool; /∗ pending handlers ∗/ list <int> m_neighobrs; // neighbors
}
```

Fig. 6. Sample implementation of a Diffusive model using PREMA's API.

3.2 LEVERAGING MULTI-CORE ARCHITECTURES

3.2.1 Multi-Threaded Design

In order to efficiently handle multi-core platforms, PREMA is natively integrated with a low-level multi-threading library. Utilizing multi-core platforms improves PREMA's performance on communication and computation aspects by making them genuinely asynchronous, increasing opportunities to overlap and hide latencies. Moreover, it allows applications to seamlessly utilize multi-core platforms without explicitly dealing with concerns raised by shared memory concurrency.

Data Movement and Control Substrate

We perform the integration with the low-level multi-threading library at the level of DMCS, which is the basis of PREMA's software stack. DMCS consists of three components, the **handler-execution**, **communication** and **application** components (Fig.7). Note that this structure remains internal to DMCS and is not exposed to the application. An application implicitly utilizes these components by requesting local or remote handler invocations. The functionality handled by each component is described below.

The **communication component** handles operations destined for remote nodes. It consists of a loop responsible for sending/receiving messages and signaling other components of PREMA regarding message requests related to/targeting them. We offer two configurations; either a dedicated thread is used to perform the communication, or any of the threads in the handler execution component is assigned this task periodically. When a thread of the two other components desires to send a message, it pushes a request to its respective list, which is handled appropriately by the communication component. , This component maintains ongoing messaging requests, and their progress is checked periodically. Once a completed request is found, it is freed, and any thread waiting for its completion is signaled to resume its execution.

Two types of messages are used, *fixed* and *split-phase*, based on whether the message size exceeds a predefined threshold. When the size of the data to be sent is less than this threshold, a fixed-size message will be used regardless of the data size. This message will contain a header needed from the runtime system, followed by the actual data that will be passed to the remote handler. When the data size exceeds the threshold, split-phase messages are used. Split-phase messages consist of two parts; a fixed-size message and a variable-sized message. The former contains the message header that will inform the receiver about the existence of a second message

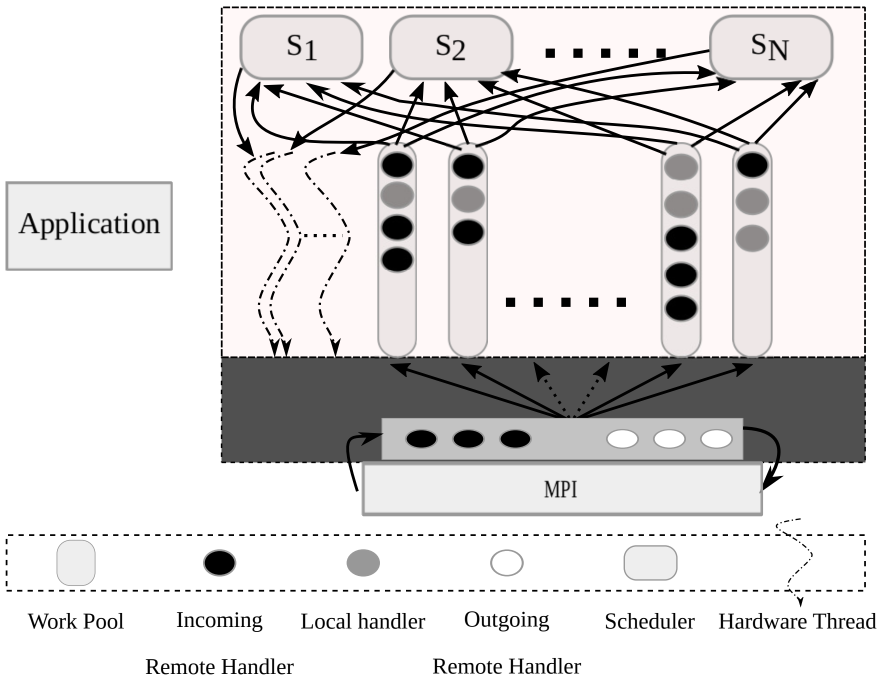


Fig. 7. The DMCS execution model. Each hardware thread is associated with a scheduler $S_1, ..., S_N$ that might use one or more private or shared work pools of handler invocation requests. A handler created locally can target the local or a remote node. The computation component (dark gray box) keeps a list of incoming handlers, which it will assign to one of the work pools and outgoing handlers that will be delivered through the communication library

and the actual size of its data. The latter contains the actual message data. The receiver can then use this information to issue a second receiving operation, eliminating the need to query the network for the size of the following incoming message.

The use of fixed-size messages also allows preallocating a pool of such messages for each thread that can be used to send or receive remote handlers. Incoming messages will be received into a preallocated message of one of the threads' pools and will be pushed to its list of pending remote handlers until the handler execution component schedules it. Once the handler is executed, the preallocated message will be pushed back to the pool for reuse. Likewise, a thread that desires to issue a remote method request will use one of the preallocated messages in its pool. Thus, new memory allocations are avoided for small and large messages, except when the pool runs out of preallocated messages and needs to be resized.

The **handler-execution component** executes the remote method invocation requests. This is the component where the bulk of computations are running and where the parallelism is exploited for application and system-related operations. When a new handler request is issued -either locally or from a remote node- an object containing this request is created by the issuer and is pushed, by the issuer if it is local or the communication thread if it is remote, to a pool of handler requests. The user can decide the number of threads this component utilizes and whether or not to bind the threads to specific hardware cores. The default number of threads is that of the available cores of the node minus one reserved for the application and optionally one for the communication component if such configuration is desired. Each thread is associated with a handler request pool and is responsible for servicing it. If the associated work pool is empty, a thread will employ work stealing with random victim selection [58] to avoid remaining idle and to balance the load in the node. When the thread's work pool is empty, and a steal attempt fails, the thread will back-off [59] for an exponentially increasing amount of time (that is constrained to a maximum value) until it finds some work, in which case the back-off period is reset to zero. Enforcing this delay reduces the memory contention and the amount of power wasted. The scheduling part of this component can be modified through an interface mainly utilized by higher-level libraries of PREMA, as discussed later.

The **application component** consists of the main function of the application. In this component, the application can start the runtime system, register the methods to be run remotely, define the number of threads used in the handler-execution component, orchestrate the application's logic, etc. Once all the pre-processing steps have been completed, the application can issue remote handler requests from this component to produce work for the other components. Once the former two components are active, the application thread can be released to the runtime system, contributing to the handler-execution threads' work until a condition (e.g., the current phase of the algorithm has finished) is met.

Before adopting DMCS to a multi-threaded design, each available core ran on a separate instance/process (i.e., similar to running one MPI rank per core). This design created the issue that each processing element (PE) could only work on its tasks, having no access to the workload/handlers of others in the same node. Thus, being susceptible to resource under-utilization in cases where only a subset of the PEs is the target of handler executions. Moreover, because all of its operations (i.e., communication, handler execution, application workflow) ran on a single thread-/core, it required explicitly calling a polling function that handled the progress of all operations. Even though this design provided a sequential execution model that simplified the process of writing applications on top of it, it could not take advantage of the benefits provided by shared memory architectures. Instead, the new design addresses those issues and further improves its performance.

The multi-threaded design allows all processing elements in a single computing node to access any pending handler invocations targeting this node. Parallelism is explored by executing multiple handlers concurrently while resources are utilized efficiently by allowing individual threads to steal their peers' work. Moreover, no explicit polling operation is required since background threads handle progress implicitly. Implementing intra-node work-sharing/stealing in the previous single-threaded design would require expensive inter-process communication and data copies, increasing the overheads sustained by orders of magnitude. DMCS does not offer abstractions to handle issues related to concurrency implicitly; instead, those are handled by MOL/ILB. This choice was made to keep DMCS as lightweight as possible since including such mechanisms would increase the critical path of one-sided communication and handler executions.

An important lesson from porting a low-level communication substrate, like DMCS, to a genuinely multi-threaded model supported by the hardware is that incorporating message passing with intra-node parallelism can significantly affect the latencies incurred. We found that maintaining per-thread message pools and funneling communication operations to a single thread can help mitigate such effects. Moreover, implementing only the necessary functionality at this low level helps to avoid overheads that may arise when aiming for ease of use and forcing correctness (e.g., checking whether the arguments of a message are valid). Finally, this decision also adheres to the principle of separation of concerns that is maintained throughout the design of PREMA.

Mobile Object Layer

Running mainly inside DMCS remote handlers, the MOL leverages the multi-threaded DMCS layer and is amplified with the ability to perform its operations in parallel. This allows running multiple remote handlers that target the same or different mobile objects concurrently and even initiate parallel object migrations from a single node. However, these operations require access to the distributed directory, and since they can run in parallel, they have to be performed in a thread-safe manner.

To avoid the contention issues created by using a lock and a simple C++ STL map, a custom hash table with chaining[1] is used instead. This approach allows elements to be inserted in different table entries safely but still exhibits possible race conditions for colliding elements. Using a lock per table entry could solve this problem; however, requiring a lock for each access is too conservative since most accesses are expected to be lookups that do not modify the directory. Instead, the atomic operation compare and swap (CAS) is used for insertions, while for deletions, the element is marked as invalid and reused instead of being removed from the list. The combination of CAS and no deletions result in a thread-safe list that does not suffer from the ABA [60] problem.

[1]in case of collisions, a list is used to keep the colliding elements in the same entry of the table

MOL message handlers are assigned access privileges that express how they may use the target mobile object exclusively (e.g., write) or shared (e.g., read). PREMA utilizes this information to extract parallelism and maintain correctness. Moreover, MOL guarantees the execution order of message handlers issued to a mobile object within a single handler context. For example, a mobile object that issues multiple message handlers to the same target is guaranteed that all of its issued messages will execute in order. Handler invocations targeting the same mobile object with exclusive access are serialized, while handlers with shared access can run concurrently. When targeting different mobile objects, handler invocations are non-conflicting whether they desire shared or exclusive access since data enclosed in mobile objects are independent.

To maintain order, messages are assigned an identifier defining the version ($lversion_i$) of the information in the local copy of the distributed directory and an increasing sequence number ($lseq_i$) maintained by each node i for each mobile object in the directory. When a new message handler arrives from node i, $lversion_i$ and $lseq_i$ are checked against the receiver j information $lversion_j$, $lseq_j$ and are only considered for execution if they are consistent. If it is determined that preceding messages are still on the way, the out-of-order messages are set aside until earlier messages have been received. Once messages are in order, their access privilege is finally examined to make decisions about the execution. For operations that do not run inside MOL handlers but may also target mobile objects, a set of locks is introduced that offers the same functionality for exclusive and shared access. An example of such a need is when a mobile object has to be migrated; it has to be locked exclusively first so that no handler tries to access it in an inconsistent state, and then packed and sent to its new location.

In summary, the multi-threaded design allows the same functionalities as the single-threaded model while utilizing all PEs of a computing node. This enables implicit shared memory load balancing, which would be even heavier to implement in MOL using distributed memory paradigms because over-decomposition requires hundreds of mobile objects in a single node to communicate through inter-process communications. Applications can use the added concurrency by annotating message handlers with access privileges. Thus, by utilizing MOL and the MOD programming model, an application can efficiently run on multi-core platforms without explicitly dealing with concerns like maintaining consistency and avoiding critical sections.

The key to achieving efficiency in a data-flow-centric system like MOL is the implementation of the lock-free distributed directory. At the distributed memory level, this problem was handled using partial independent copies of the directory, possibly holding out-of-date information that was updated lazily to avoid excessive communications. We used a lock-free hash table at the shared memory level that allowed quick, concurrent access to different mobile objects. By allowing low-overhead concurrent access to different mobile objects and keeping information related to each mobile object in a per-object structure instead of a global structure, one avoids thread contention unless it becomes necessary (i.e., handlers targeting the same mobile object).

Also, because of over-decomposition, the chances of having too many requests for concurrent access to a single mobile object are expected to be relatively low; thus, the contention is also expected to be low. Even when that is not the case, using atomic operations instead of locks (when possible) mitigates performance issues.

3.2.2 Implicit Load Balancing

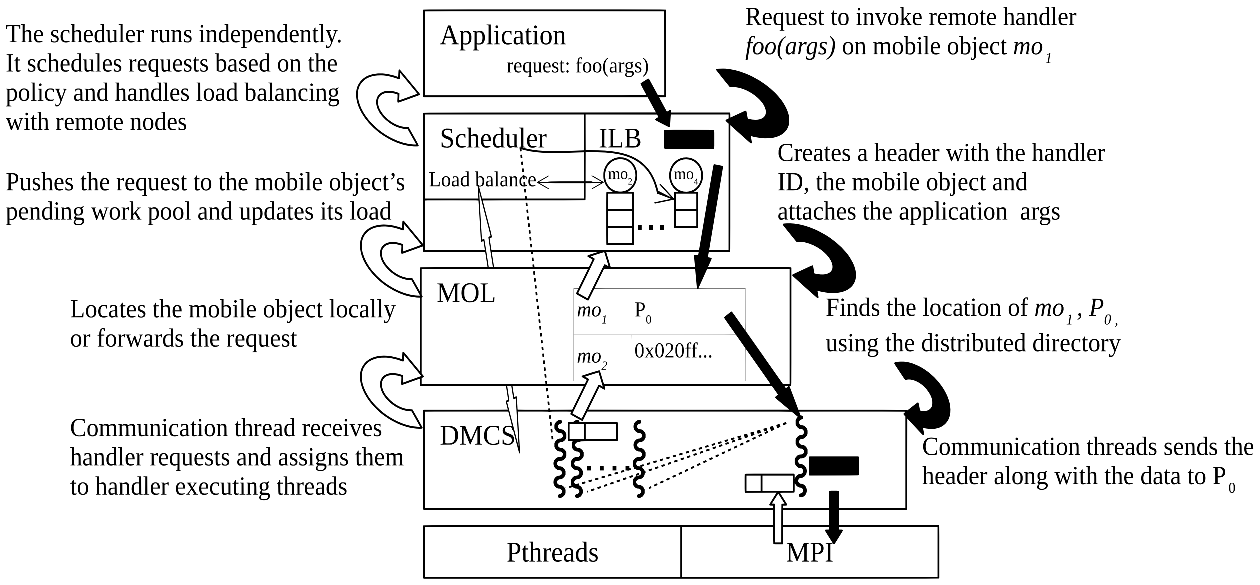

The scheduler runs independently. It schedules requests based on the policy and handles load balancing with remote nodes
Application
request: foo(args)
Request to invoke remote handler *foo(args)* on mobile object mo_1
Scheduler
ILB
Load balance
mo_2
mo_4
Creates a header with the handler ID, the mobile object and attaches the application args
Pushes the request to the mobile object's pending work pool and updates its load
MOL
mo_1
P_0
mo_2
0x020ff...
Locates the mobile object locally or forwards the request
Finds the location of mo_1, P_0, using the distributed directory
DMCS
Communication thread receives handler requests and assigns them to handler executing threads
Communication threads sends the header along with the data to P_0
Pthreads
MPI

Fig. 8. High-level representation of DMCS, MOL, and ILB interactions.

ILB is responsible for scheduling and load balancing at both the shared and distributed memory levels. At the distributed memory level, ILB maintains the workload of each mobile object and, consequently, the workload of each node. Also, it encapsulates the operations required to maintain load balance across nodes, including information dissemination, bookkeeping, and decision-making. The operations requiring inter-node communication are built utilizing the DMCS layer and can run in parallel, allowing concurrent mobile object migrations from the same or different nodes. Running such operations in parallel improves response time, makes it easier to overlap them, and enables more sophisticated load-balancing schemes. On the other hand, it increases the complexity of developing new load-balancing algorithms; however, this complexity is constrained to the scheduler module, which a typical user does not need to modify or adapt unless a new policy needs to be developed. Mobile object-specific locks are convenient in this context since policies can use them to guarantee that no handler is running on a mobile object before trying to migrate it. After exclusively locking an object, it is safe to pack and migrate it to another node along with its workload. By design, the scheduling interface does not need to check whether the pending handlers in its pools target mobile objects that might have migrated. Such handlers are invalidated during the packing process and are ignored when popped from the scheduling interface.

The handler-executing threads periodically call the ILB scheduler once they have finished pending work in the DMCS and MOL layers. Before initiating distributed load balancing, a scheduler would usually prefer to execute the local workload. The workload consists of pending ILB handlers residing in the local work pools of the scheduler (and internally in the respective list of individual mobile objects). ILB handlers are assigned exclusive or shared access, holding the same principles as discussed for the MOL. When a handler is safe to be executed (i.e., it is in order and does not conflict with others), it is pushed to the local scheduler work pools through the respective callback (see section 3.1.3). By enforcing all dependencies to be resolved before the local scheduler is informed about a new handler, we relieve the scheduling policy from having to maintain execution order correctness. In contrast, the load of local mobile objects is updated even before the respective handler can execute. This allows the scheduler to be informed about the whole workload of a node regardless of whether it has dependencies.

A high-level representation of the interactions between the different layers is presented in Fig. 8. When a new message is available, it is received from the communication component of DMCS and assigned to one of the handler-executing threads (light gray arrows). If it is a load-balancing request, it is routed directly to ILB. If it is an application request targeting a mobile object mo_2, it will pass through the MOL to look up the object's location using the distributed directory. If the mobile object is not local, the request is forwarded to the object's last known location. Otherwise, the request is passed to ILB, which will update the load of the respective mobile object and push the request to its work pool. The ILB scheduler runs independently and can schedule some requests or initiate load balancing by sending such requests to other nodes using the DMCS interface. When the application needs to invoke a handler remotely to some mobile object mo_1 (dark gray arrows), a new request is created by ILB with relevant information attached. Finally, the request is passed to MOL, which finds the location of the mobile object and sends the request using the DMCS layer.

Last but not least, ILB offers the *ilb_multicast* operation. This handler execution request can be sent to multiple mobile objects and will only start running when all of those mobile objects reside in the same node. Its inputs are the mobile objects needed to be in the same node, a buffer for the arguments, and the index of the mobile object on which it should start running. Once called, it migrates (if needed) the mobile objects on the same node, locks them so that another handler cannot move them, and schedules the requested handler. This operation was implemented to support applications that may require access to multiple mobile objects in a single handler before they can start their computations like the one presented in [61]. An effort that uses this functionality can be found in [56] but is out of the scope of this work.

In summary, the multi-threaded design of ILB allows efficient utilization of multi-core nodes while monitoring their workload and providing load balancing. Using an easy-to-use API, one can quickly implement custom shared and distributed memory (2-level) scheduling and load balancing policies without the need to handle dependencies or reason about mobile object locality. Work units available at the scheduling module are either safe to execute (i.e., have no dependency conflicts) or have already been invalidated by the runtime. As opposed to a single-threaded design, load balancing among cores in the same node does not need to go through the process of the mobile object (de)serialization and expensive rounds of synchronization. Instead, just passing a pointer between threads is enough to share workload. Moreover, it enables concurrently migrating multiple mobile objects while computations are also in progress, thus, hiding and overlapping latencies in a single node.

By utilizing the mobile object-specific locks, one is able to safely and correctly maintain the load of individual objects and monitor their pending handlers in the respective lists by only serializing conflicting operations on the same mobile object. The lessons learned from implementing a 2-level implicit load balancing framework is to separate the concerns between local and global decision-making as well as between correctness and performance. Monitoring the information for each level separately allows for easier, cleaner, and more efficient implementation. Handling correctness at a lower level also improves performance and mitigates errors. Moreover, maintaining workload and pending handler information in a per-object fashion rather than a global one improves performance and avoids excessive contention.

3.2.3 Correctness

Formally proving correctness in a complex system like PREMA is a difficult task. Formal methods, Markov chains, discrete event simulations, or Petri nets could be some approaches to tackle this problem. Such a systematic study for correctness is left as future work and is outside the scope of this paper. As an initial step towards proving correctness, we try to demonstrate that our system is free of inherent deadlock problems based on its design.

Starting from the lowest layer, DMCS, the communication component is implemented using only non-blocking MPI calls to avoid deadlocks in the two-sided communication model. The message-receiving part of the component uses *MPI_Iprobe()* to check for messages from any source. Once an incoming message is found, the respective memory is allocated, and the receiving call is issued safely. The sending part also uses asynchronous sending operations. Progress for sending and receiving is checked periodically between or parallel to method invocations. The asynchronous messaging operations and the restriction that only one thread per MPI rank can send/receive messages at a time guarantee that messaging will not lead to deadlocks.

In the threading component, threads are associated with thread-safe work pools that hold handler invocation requests and share their work through stealing. To guarantee correctness, handlers encapsulate tasks that do not rely on a future handler invocation for progression. Handlers can issue blocking messaging calls, which internally progress message-passing, and use mutual exclusion (locks) if they are acquired and released in the context of a single handler (i.e., no inter-handler lock-unlock) but cannot block waiting for a handler that has not yet started executing. For example, it is safe for multiple handlers to compete for the same lock as long as they release it before completing their execution. This restriction guarantees that all handlers complete at some point, and there is no case of deadlocks.

The MOL builds on DMCS' handler threads and follows the same restrictions. In [52], it is formally proven that MOL, with a single thread, can always keep this information current and consistent no matter how many migrations occur in the distributed system. We build upon this for the multi-threaded version to maintain correctness. As presented in section 3.2, we use a thread-safe hash table to track mobile objects and maintain atomic access per mobile object entry. Atomic access guarantees that information like messaging sequence numbers and directory version identifiers are checked and updated safely in the presence of multiple threads. Moreover, handlers targeting mobile objects are associated with an access type that lets users define mutual exclusion at a higher level, leaving PREMA to handle the correct submission of conflicting handlers. The information about the access type of a handler executing on a mobile object is also maintained in the distributed directory entry. When a new handler is about to be executed on a mobile object, its access type is checked against the current mobile object state. If another handler is already running on the object with a conflicting access type, the new handler is suspended in a thread-safe list of waiting tasks maintained per mobile object. When the current handler completes, it resubmits its dependent handlers back to the main work pool. By utilizing access types, PREMA can guarantee that no conflicting handlers will run concurrently, relieving the application from this burden and improving performance.

The ILB utilizes DMCS and MOL to route handler invocation requests to the appropriate mobile object and inject them into its thread-safe pool of pending work. Updates to the load balancing policy are also triggered in this process, constraining the policy implementations to the same restrictions as handler invocations. ILB utilizes the mobile object-specific locks of MOL to guarantee that a mobile object cannot migrate while at least one handler is executing on it. Respectively, it also guarantees that while a mobile object is in the process of packing and migration, no handler can start running on it. Moreover, handlers targeting migrated mobile objects are automatically invalidated, removing this burden from the scheduling policy. Thus, given that the handler invocations and the load balancing policies are implemented under the above restrictions, ILB and PREMA are free of inherent deadlock situations.

In summary, arguments can be made for the absence of inherent deadlocks at the communication, handler execution, message ordering/forwarding, and scheduling/workload-related bookkeeping. However, a complete formal proof of the correctness is beyond the scope of this work.

3.3 PERFORMANCE EVALUATION

3.3.1 Experimental Setup

The following experiments have been conducted on a computing cluster consisting of 190 Intel(R) Xeon(R) (E5-2660 v1-2, E5-2670 v2, E5-2698 v3, E5-2683 v4) computing nodes. Each node has two CPUS of 16-32 threads, 128 GB of memory, and runs Red Hat Linux. We used the MPICH 3.1.3 MPI implementation as the communication library and gcc 6.3.0 for compiling. The SW4lite proxy application was run on a newer cluster with Intel(R) Xeon(R) Gold 6148 @ 2.4 GHz CPUs of 40 cores each, using OpenMPI 3.1.4.

3.3.2 Communication

We first evaluate the performance of the communication functionalities of PREMA, a vital part of the runtime system since PREMA adopts a message-driven execution model. We evaluate the performance of each layer of PREMA using a simple ping-pong benchmark and compare the results with those of an MPI implementation. For DMCS, we use two processes residing on two different nodes where process zero sends a remote method invocation request of size X to process one and blocks, waiting for the response. On arrival, the remote method executes and sends a request of the same size back to process zero, which unblocks and triggers it to send the next message. This pattern is repeated 1000 times for each message size, and the average latency and bandwidth achieved are reported. For MOL and ILB, the procedure is the same, but the messages are sent between two mobile objects residing in processes 0 and 1, respectively.

In Fig. 9a, one can see that DMCS, MOL, and ILB add a roughly fixed amount of overhead to the latency independent of the message size. Each layer is bound by the performance of the lower layers and the performance of MPI. Since the overhead is relatively fixed, its effect is more noticeable in smaller messages where the MPI time is low, and the overhead is a significant percentage of the overall time, as seen in Fig. 9b. The effects on the bandwidth seem to be less significant for both small (Fig. 9d) and large messages (Fig. 9c). ILB messages experience the highest overhead; passing through DMCS and MOL, and being registered with the load balancing module before scheduled, increases their critical path. However, the overhead added is acceptable for the functionality provided.

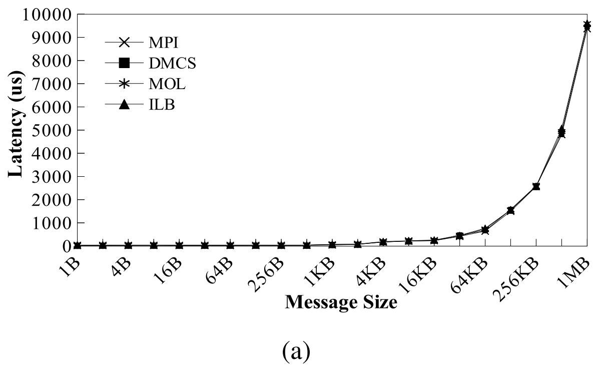
As mentioned before, DMCS uses pools of preallocated, fixed-size messages consisting of the headers required by the remote handler requests. By maintaining preallocated pools, we reduce the overhead accountable to memory allocations and avoid querying for the size of incoming messages. As a result, the latency for initializing the message delivery process is reduced and becomes relatively stable among different invocations. When the arguments of a handler are small enough to fit inside a fixed-size message, they are copied into it; otherwise, a separate message is sent for them. In the former case, the receiver has the preallocated messages ready to receive, while for the latter, the buffer can be allocated, and the receiving call can be issued as soon as the headers are received. This helps to overlap the time the sender takes to send the second message with the time it takes for the receiver to prepare for the delivery.

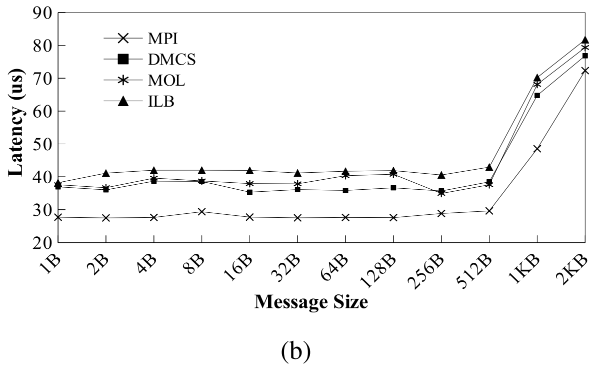
Figure 10 shows the advantage of using preallocated, fixed-size messages. Small messages (smaller than 2KB) are highly benefited by this optimization, whether arguments are copied to the header (up to 512B) or sent separately (1KB); latency (Fig. 10a) is almost half of the case where no preallocated messages are used (39 compared to $77\mu s$). Furthermore, the bandwidth (Fig. 10b) is also significantly affected for small messages larger than 64B with up to 100 percent improvement. For messages larger than 2KB, the performance of the two approaches is similar because the transmission cost becomes the dominant factor.

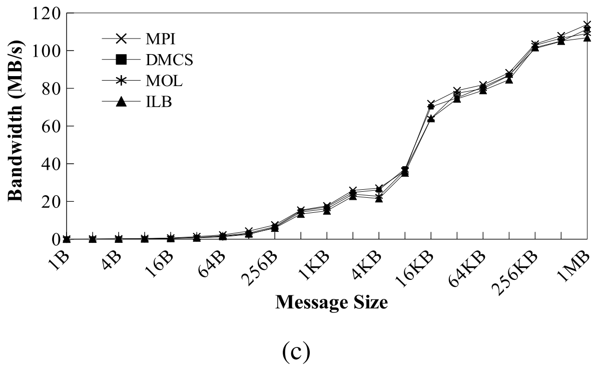
3.3.3 Load Balancing

Synthetic Benchmark

Next, we evaluate the performance of PREMA in terms of load balancing, overall application runtime, and the overhead imposed by the runtime system. We start with a simple synthetic bench-







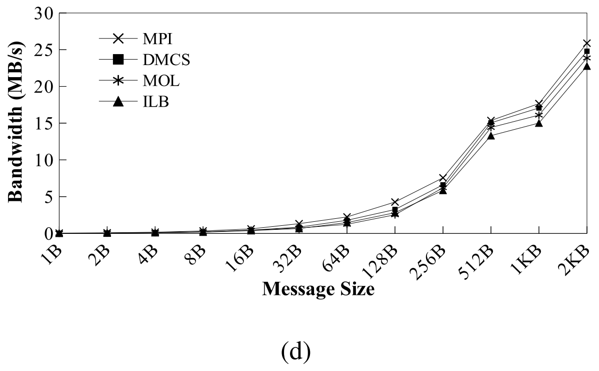
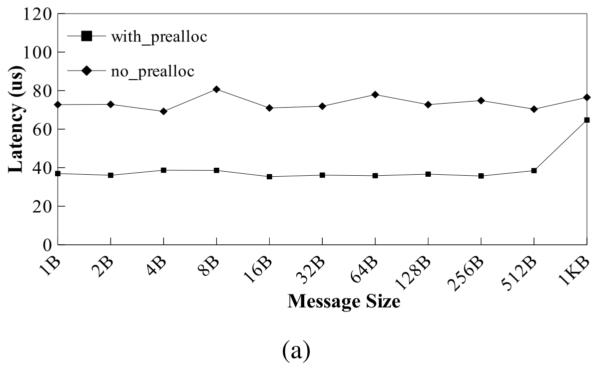


Fig. 9. Ping-pong benchmark performance for all three layers compared to MPI. Comparison of (a) latency, (b) latency for small messages, (c) bandwidth, and (d) bandwidth for small messages.



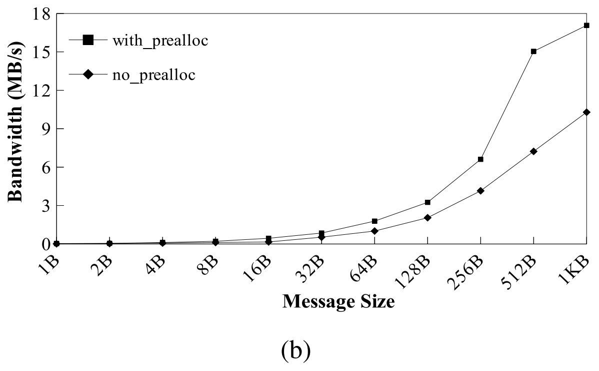


Fig. 10. Ping-pong measurements for DMCS without preallocated messages. The effect of using preallocated messages in (a) latency and (b) bandwidth. The performance for messages larger than 1KB is comparable to the optimized version and is not shown here.

mark to test the system in a fully controlled and isolated environment. The benchmark starts by creating mobile objects on process zero and dispersing them to the available cores; a computation is then invoked on each mobile object via PREMA's messaging mechanism. When all computations on a mobile object have been completed, a notification is sent back to process zero; the benchmark terminates once all completion notifications have been received. We assign ten mobile objects to each available core and specify a weight (workload) from two categories, light and heavy, for each mobile object. The average execution time of a heavyweight mobile object is 2.5x the execution time of a lightweight, and 20% of the mobile objects are assigned to the heavyweight category. Each instance of PREMA consists of the same amount of cores, and we employ a diffusive load-balancing algorithm to evaluate its performance.

For the MPI version of the benchmark, we replace the mobile objects with plain data objects and perform only static load balancing. Once an MPI rank has received all its data, it will execute its computations and terminate. Note that even though PREMA uses one core exclusively for message handling, this core is counted as an available core when we calculate the number of mobile objects to distribute to each node since the MPI version can utilize all cores of the node for computations.

The performance comparison for the benchmark is shown in Fig. 11. The workload distribution indicates the impact of using PREMA compared to the MPI implementation. Figures 11a and 11b compare the workload distribution of 320 cores using plain MPI (320 ranks) and PREMA (10 ranks (nodes) of 32 threads each). In Fig. 11a, the heaviest workloads have been gathered roughly to the first 130 cores for an overall running time of 683 seconds. Fig. 11b shows the results after porting the benchmark on top of PREMA, where the workload has been redistributed equally among the available cores, decreasing the overall running time to 495 seconds, an improvement of 27.53 percent.

Figures 11c and 11d show the performance of the same benchmark when we quadruple the number of cores and work units. The pattern for the workload distribution remains the same, and as such, we see the same performance breakdown as before. Fig. 11d shows that PREMA is unaffected by the vast increase in the number of cores and tasks needed to balance. Even though the overall runtime slightly increases, it is not a penalty from PREMA itself but from the initialization stage of MPI and the distribution of the initial workload of the problem. This is more evident in the MPI case; the dashed line indicates the overall running time, including the initialization step. Figures 11e, 11f and 11g, 11h present the results for 3200 and 5600 cores where the size of the problem has increased accordingly. For these cases, PREMA uses 16 cores per instance instead of 32, increasing the number of instances and the cores reserved for communication. In other

TABLE 2
Time breakdown and comparison with the MPI version of the synthetic benchmark. The overhead presented is per thread utilized. The maximum and minimum refinement times correspond to the time spent by the most and least loaded thread in the lifetime of the benchmark. Maximum refinement time dominates the overall running time.

#cores	PREMA overhead (sec)			Min refinement (sec)		Max refinement (sec)	
	Max	Min	Avg	PREMA	MPI	PREMA	MPI
320	0.84	0.0001	0.07	464.8	324.9	494.4 (-27.4%)	681.8
1280	0.76	0.0001	0.05	464.9	324.7	497.6 (-29.3%)	704.1
3200	0.35	0.0001	0.04	480.7	322.6	515.2 (-25.5%)	691.9
5600	0.48	0.0003	0.04	479.2	310.6	515.9 (-28.1%)	717.8

words, there are fewer cores for computations per hardware node than before (2 communication cores vs. 1 communication core per node). Despite those modifications, PREMA still maintains a fair workload distribution and is not affected as much by the MPI initialization step (mainly because far fewer MPI ranks are initialized). The overall improvement for the two cases is 27.42 and 31.63 percent, respectively. Table 2 presents the overhead of PREMA, the minimum and maximum refinement time, which is also the factor that impacts most the overall running time. The minimum/maximum refinement time corresponds to the running time of the worker that finished all of its tasks at the earliest/latest, respectively. The effectiveness of the dynamic load balancing can be noticed by how the variance between these two values has been reduced, compared to the variance in the MPI version, while maintaining a very low overhead of, on average, 0.05 seconds.

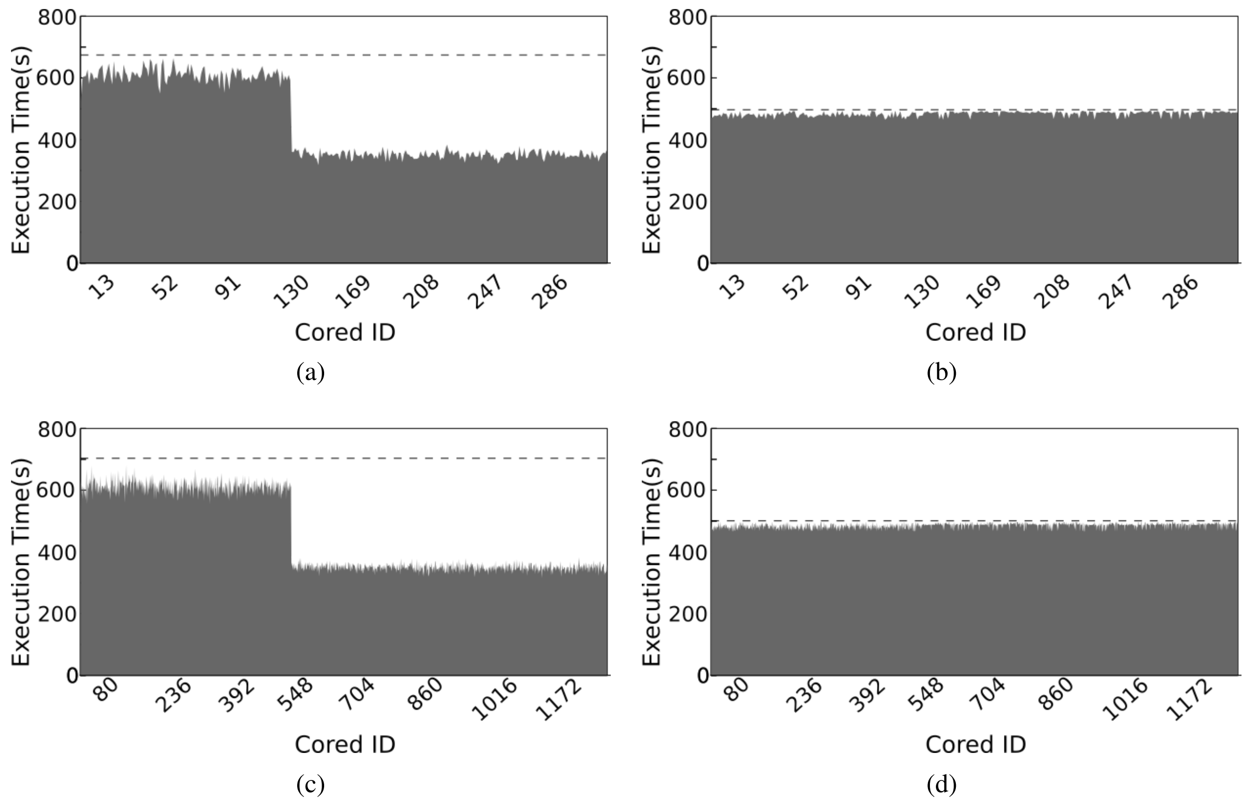
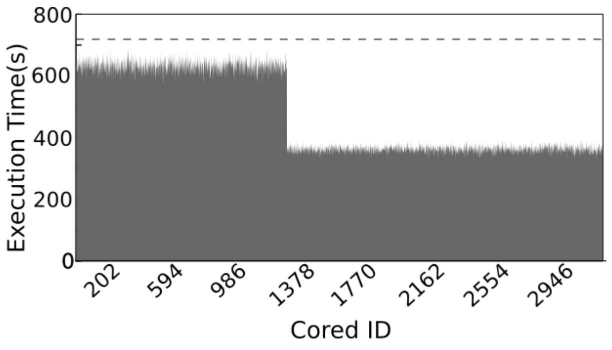
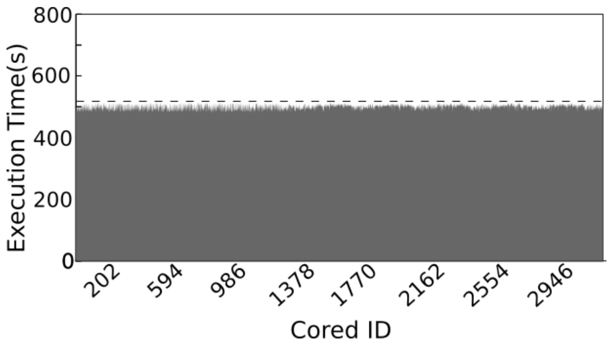


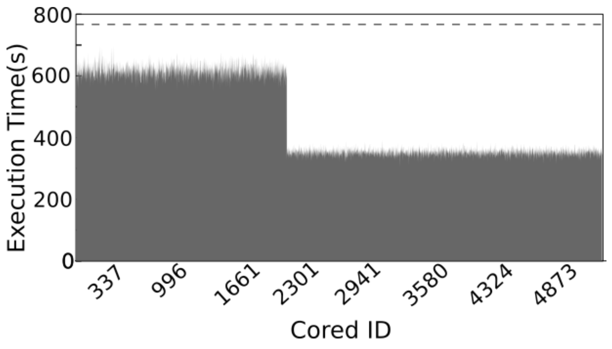
Fig. 11. Per core workload breakdown running the synthetic benchmark. Cases shown for MPI with (a) 320, (c) 1280, (e) 3200, (g) 5600 cores and PREMA with (b) 320, (d) 1280, (f) 3200 and (h) 5600 cores. Each core is a different MPI instance (rank) for the MPI cases. For the cases with PREMA, one core per instance is reserved for communication, (b) and (d) have one instance per 32 cores while (f) and (h) have an instance per 16 cores. The black dashed line indicates the overall running time, including the initialization computations and termination time.



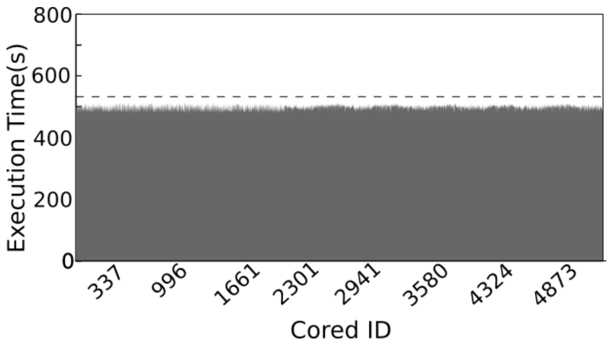
(e)



(f)



(g)



(h)

Fig. 11. Continued

Parallel Mesh Refinement Application

The initial motivation for developing PREMA is to separate the concerns of performance and algorithmic correctness for parallel mesh refinement applications. We used our in-house developed tetrahedral mesher CDT3D [62] as an application benchmark to evaluate its performance with such applications. CDT3D uses as input a triangulation of a Piecewise Linear Complex (PLC) of the domain to be discretized. The basic steps involve: creating a Delaunay Tetrahedralization of the boundary points using Delaunay point insertion, recovering the boundary using topological transformations and edge/face partitioning, and refining the mesh. During the mesh refinement procedure, points are created using an Advancing Front type point placement and then inserted by direct subdivision of the containing tetrahedra. The connectivity of the mesh is then optimized using a combination of topological transformations.

For this experiment, the first two steps (i.e., Delaunay Tetrahedralization and Boundary Recovery) were executed sequentially, as they are needed to bootstrap the mesh and are less time-consuming than mesh refinement. The resulting mesh was then partitioned into N sub-domains with $N >> \#cores$ using an octree adapted to the mesh density (see Figure 12). The sub-domains are registered as mobile objects and then serialized and distributed among the available processes by PREMA. This decomposition scheme was selected to create sub-domains with a similar number of tetrahedra to create a balanced initial workload per sub-domain. The surface of each sub-domain is constrained (remains unchanged) during the refinement, and thus, there is no need for communication between the subdomains. In the future, we plan to relax this requirement by either allowing modifications on the boundary and consequently introducing a small amount of communication along the boundaries of the sub-domains or by pre-refining the sub-domain boundaries in a separate pre-processing stage.

Porting CDT3D on top of PREMA only requires writing the appropriate handlers and callbacks, which initialize and execute the CDT3D mesher. More specifically, the handlers and callbacks used by this experiment are the following:

- Pack/Unpack Sub-domain callbacks for migrations
- Initialize, for bootstrapping the mesher's data structures
- Refine, for the sub-domain refinement
- Callback for calculating the weight (computational cost) of a handler

We ran the application with ILB and MPI to evaluate the load balancing quality. The pre-processing step is identical in both cases; once the sub-domains are created, they are assigned to

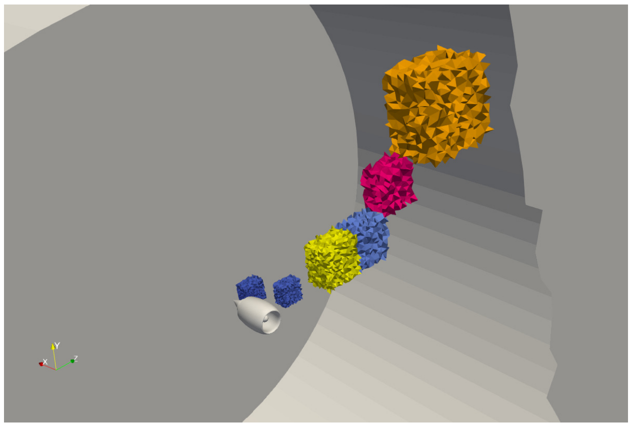
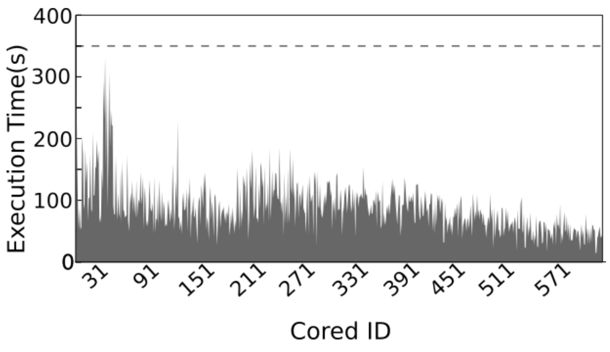


Fig. 12. Subdomains of the initial coarse mesh used to bootstrap the parallel refinement process. The input is a surface mesh of a nacelle inside a cylindrical domain. An adaptive octree is used to decompose the volume mesh to equal meshes based on the number of tetrahedra in each leaf. In this case, subdomains are the unit of work captured by mobile objects and are used to perform dynamic load balancing in the distributed system.

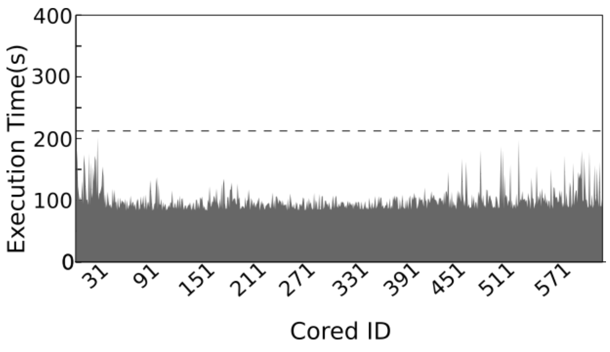
the available worker cores. Fig. 13 depicts the performance comparison of ILB versus MPI using the mesh refinement application. Figures 13a, 13b and 13c, 13d show the results when MPI and PREMA are utilizing 640 and 1280 cores respectively. Cores are allocated in the same manner described in section 3.3.3 using the same mesh size of 30 million elements, over-decomposed into 4.5 thousand sub-domains. The graph does not include the pre-processing and decomposition times since they run on a single thread and are the same for all cases. A diffusive load balancing policy is used where each instance of PREMA can only share its load with a specific subset of the available instances. If no instance in the subset has enough workload, a new subset is formed from randomly selected instances. The performance improvement that ILB offers compared to the MPI implementation with static load balancing for those two cases is 40.5 and 26 percent, respectively, by dynamically redistributing the available sub-domains to the starving workers. As can be seen from table 3, the maximum refinement time decreases from 340 to 197 seconds (40%), and the difference between them is reduced from 326 to 116 seconds while the overhead imposed from the load balancing is on average 0.48 seconds per core utilized. For the 1280 cores case, the maximum and minimum refinement time difference decreases from 223 to 131 seconds, while the overall time decreases from 225 to 162 seconds (26%). Figures 13e, 13f and 13g, 13h show the results of running the application with an initial mesh of size 110 million elements, decomposed into 27 thousand sub-domains on 3200 and 5600 cores. The performance gain is even larger in these cases as by increasing the mesh size, the load imbalance also increased. The improvement exhibited is 56 and 43.6 percent, respectively. In table 3, one can see the high variation of refinement time for MPI, which is reduced by PREMA's implicit load balancing. While for MPI, the refinement time varies by up to 754 and 488 seconds, PREMA manages to take the variance down to 239 and 215 seconds, respectively. The time attributable to the runtime system is negligible for all these cases. Table 3 shows the overhead of PREMA being less than one percent of the overall runtime and the decrease in variation between minimum and maximum refinement time.

An important observation from Fig. 13 is that even though the load distribution improved, it is still not optimal. Some cores ran for a much longer time than the average. Table 3 presents the same observation where the minimum and maximum refinement time variance is much more significant than the one noticed in the synthetic benchmark. The mesh decomposition quality causes this problem; even though the adaptive octree tries to create sub-domains of similar size, the refinement time per sub-domain can still differ dramatically.

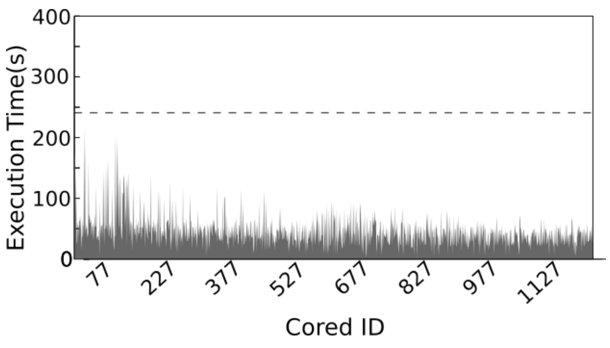
For example, we have cases where the refinement of a single sub-domain could last as long as the refinement of a hundred others, dominating the overall time. Since concurrency is exploited by running refinement on different sub-domains using one thread per sub-domain, adding more threads cannot lower the refinement time of a single sub-domain. This is also why PREMA significantly improves the application's performance compared to MPI, but it does not scale as expected when we increase the number of cores. We plan to modify the application to incorporate data decomposition on top of domain decomposition to allow parallelism inside a sub-domain and mitigate this issue. We will also look into more effective decomposition methods for the initial work distribution.



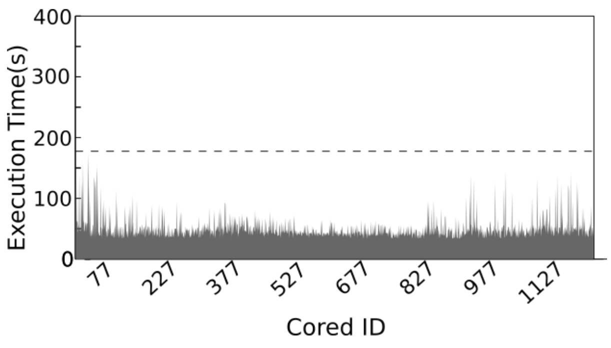
(a)



(b)

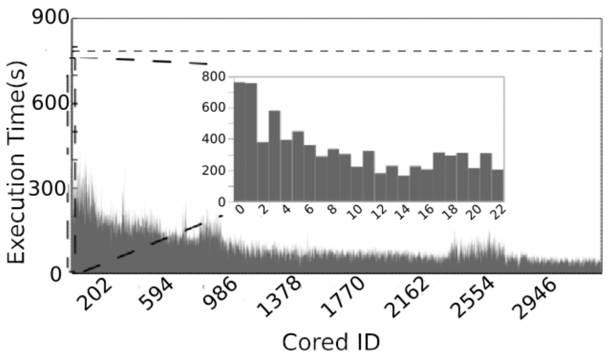


(c)

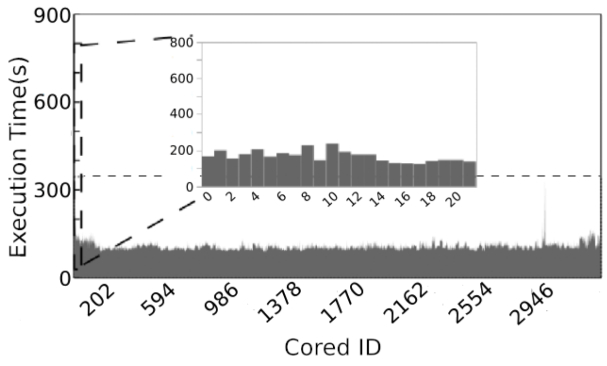


(d)

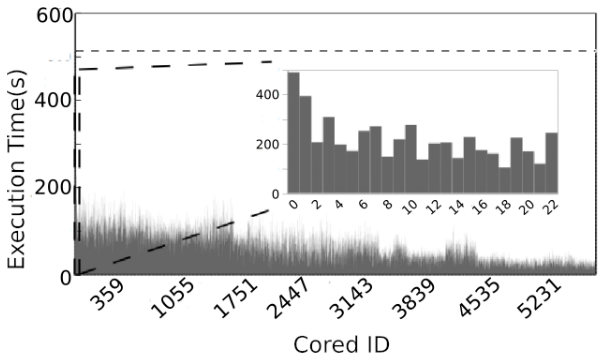
Fig. 13. Per core workload breakdown running the CDT3D benchmark. Cases shown for MPI with (a) 640, (c) 1280, (e) 3200, (g) 5600 cores and PREMA with (b) 640, (d) 1280, (f) 3200 and (h) 5600 cores. Each core is a different MPI instance for the MPI cases. For the cases with PREMA, one core per instance is reserved for communication, (b) and (d) have one instance per 32 cores while (f) and (h) have an instance per 16 cores. (a), (b), (c) and (d) use an initial mesh of 30 million elements, decomposed into 4.5 thousand sub-domains while (e), (f), (g) and (h) use an initial mesh of size 110 million elements decomposed into 27 thousand sub-domains. The black dashed line indicates the overall running time, including initialization, computations, and termination time.



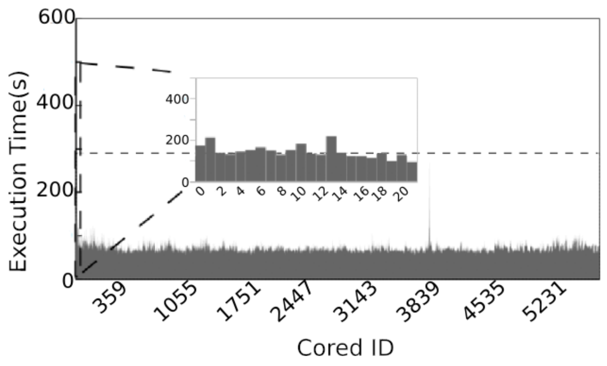
(e)



(f)



(g)



(h)

Fig. 13. Continued

TABLE 3
Time breakdown and comparison with the MPI version of CDT3D. As in table 2, the overhead presented is per thread utilized. The maximum and minimum refinement times correspond to the time spent by the most and least loaded thread in the application's lifetime. Maximum refinement time dominates the overall running time.

#cores	PREMA overhead (sec)			Min refinement (sec)		Max refinement (sec)	
	Max	Min	Avg	PREMA	MPI	PREMA	MPI
640	0.7	0.004	0.18	80.4	13.9	197.12 (-42.1%)	340.7
1280	0.82	0.001	0.05	31.12	2.1	162.2 (-27.9%)	225.1
3200	37.9	0.04	0.65	86.1	9.8	325.2 (-57.4%)	764.4
5600	29.5	0.02	0.48	51.2	1.6	266.8 (-45.5%)	489.9

Seismic Wave Simulations

SW4lite [63] is a proxy application that models the workflow of SW4 [63], an application that implements substantial capabilities for 3D seismic modeling. Sw4lite is a simplified version of SW4 intended for testing performance optimizations in a few essential numerical kernels. It was developed to support the Exascale Proxy Applications Project [6], an effort to improve the quality of proxies created for Exascale systems and maximize the benefit of their use.

Even though this application is not impacted by load imbalance, it is communication-intensive and tightly coupled, a good candidate to showcase PREMA's low overhead and applicability even on applications that are not the main target of PREMA. We make the case here that even an application that consists of different kernels, where only a subset of them benefit from dynamic load balancing, can be ported to PREMA without being affected negatively in the other kernels.

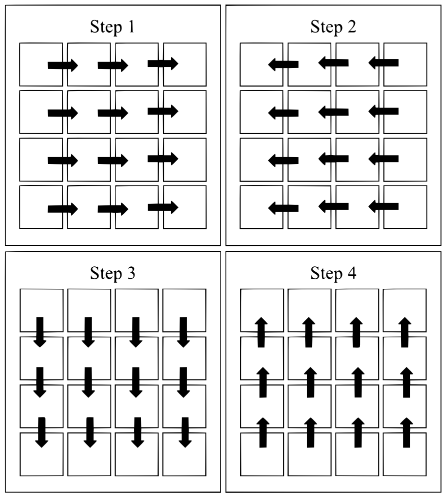
The proxy application starts by decomposing the original 2D grid into several partitions equal to the number of available processes positioned into a logical 2D grid. Each processor is assigned a partition of the application grid, and a pre-processing step follows before the main kernel starts. The main computation kernel runs in an iterative fashion consisting of computations intercepted by two cycles of neighbor-to-neighbor communication per iteration. The communication pattern for each cycle is as follows: Each processor sends some data from the partition it holds to its left neighbor and waits to receive the respective data from its right neighbor. Once the data are received, the same processor pair shares data in the opposite direction. Next, the same process executes for the y-axis; each processor sends another portion of its data to its bottom neighbor and waits to receive the respective data from its top neighbor. Next, the communication continues in the opposite direction. Processors at the grid edge only send/receive data to/from the available neighbors. Figure 14a shows the communication pattern schematically.

To port the application on top of PREMA, we have undergone the following process:

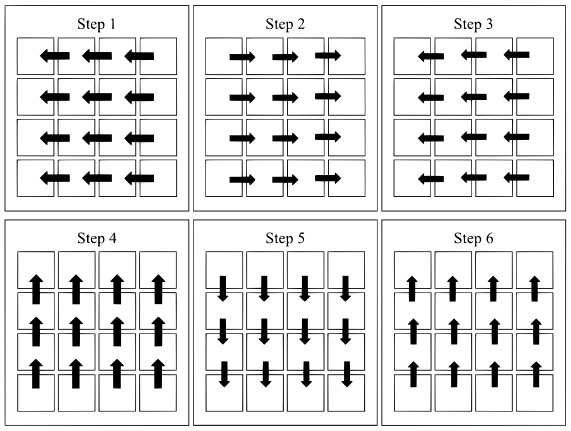
- Each partition of the decomposed 2D grid is registered as a mobile object.
- Each MPI rank holds partitions equal to the number of cores it utilizes.
- Pre-processing computations are performed by invoking remote handlers on each partition.
- Two-sided communications are replaced with one-sided asynchronous remote method invocations.

The original four-step communication pattern of the iterative process has been modified to ensure correctness. In contrast to MPI, where the receiver can explicitly request the data to be received, PREMA's message receiving is implicit; thus, we need to ensure that the receiver is ready to accept the data without corrupting its state. We have added a data request for each data transfer to achieve that. The idea is that the communication in each direction begins with the receiver neighbor requesting the data when it is ready to receive them. The sender will send the data when it is ready to do so. In this way, we guarantee that both neighbors' data are consistent after the first and third communication steps. The requests for the second and fourth steps are implicit as part of the data sent in the first and third steps. Figure 14b demonstrates the modified communication pattern.

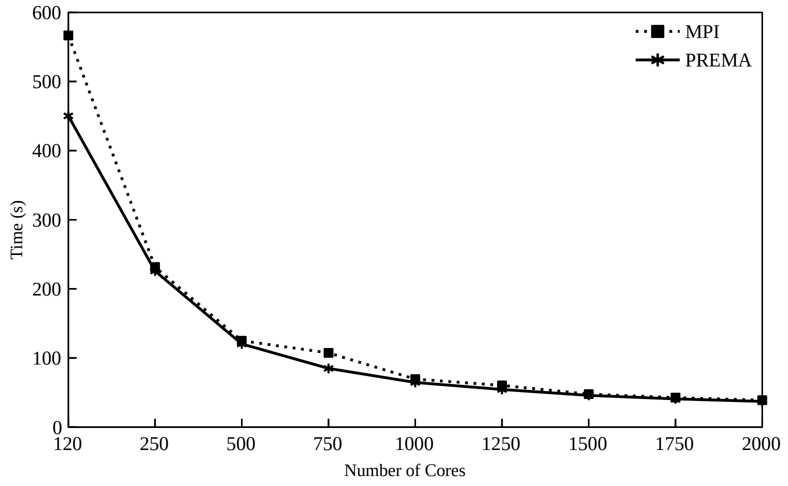
For this work, the problem LOH.1 presented in [64] is given as the input to SW4lite. PREMA runs using one MPI rank per available socket of ten hardware cores, while the MPI version runs with one MPI rank to one core mapping; each computing node holds four sockets for a total of 40 cores per node. We use the configuration where PREMA does not have a dedicated thread for communication, but the handler execution threads run the communication functions periodically. Figure 14c shows the performance comparison for the two approaches; PREMA's performance is equal and, in some cases, even better than the MPI implementation, even though the application does not benefit from load balancing. Moreover, in the implementation with PREMA, we needed to add an extra step of message passing per iteration. Nevertheless, PREMA managed to maintain and improve performance by overlapping communications with computations (since both are asynchronous) and sharing threads in a socket to handle different requests. It is important to note that all load monitoring-related operations are run for PREMA, even though there is no need for load balancing, and there is no significant overhead.



(a)



(b)



(c)

Fig. 14. The communication patterns of the two implementations. (a) Plain MPI SW4lite implementation, (b) modified implementation on top of PREMA. The arrows in steps 2 & 5 in (b) depict the data requests that have been added for the PREMA implementation of SW4lite. (c) The performance comparison between the two implementations for different numbers of cores.

3.4 SUMMARY

We have presented the new design and implementation of our runtime system PREMA. We have shown the advantages of enhancing it with multiple threads dedicated to specific operations like message passing and computation and stated the lessons learned from this process. The new implementation can implicitly leverage shared and distributed memory parallelism by exploiting task access privileges through a uniform interface. We presented the abstract interface that one can utilize to develop custom 2-level scheduling and load-balancing policies. The multi-threaded model does not heavily impact the latency and bandwidth of the communication library while enabling more efficient communication and computation overlapping. Furthermore, we demonstrated the performance improvements of using PREMA with a parallel 3D advancing front mesh refinement application. We have noted an improvement of up to 56% compared to an MPI implementation without load balancing in varying sizes of core allocations ranging from 640 to 5600 cores with a negligible overhead of less than one percent. We have also demonstrated that the performance of tightly coupled and communication-intensive applications that do not need dynamic load balancing is not affected significantly by the additional load monitoring-related operations of PREMA.

CHAPTER 4

TOWARDS EXASCALE COMPUTING

While designing and implementing high-level DSL constructs on top of PREMA, we realized some of its limitations, also found in similar systems, that inhibit its performance and ease of use. Some of these limitations include the following: Requiring remote method invocations (Active Messages) or tasks to run to completion to avoid delaying the progress engine. Inability to preempt task execution. Limited high-level constructs to check remote task completion and task dependencies. Inability to balance a coarse-grained work unit's load once scheduled.

Restricting applications to run-to-completion tasks is a common requirement among shared and distributed memory runtime systems like PREMA. Tasks are usually expected not to synchronize with each other, except for "parent" tasks synchronizing with their "children". Therefore, they should refrain from being involved in a (busy)-waiting routine, e.g., waiting for an MPI message reception or an interrupt to fire. Moreover, such systems lack the ability for tasks to voluntarily release control of a CPU thread when busy-waiting is unavoidable to allow other tasks to execute. These restrictions are put in place to prevent tasks from delaying the execution flow or even causing deadlocks; however, they increase the complexity of developing applications on top of such systems, especially when porting legacy codes. In section 4.2, we present an attempt to loosen these constraints by introducing the *wait_until()* construct that allows PREMA to preempt a task execution until a given condition is satisfied. We show that this feature can quickly lead to live-lock scenarios when using tasks that do not use dedicated stacks and cannot context switch.

Unstructured and adaptive applications are usually hard to scale across multiple computing cores and nodes due to the difficulty in statically dividing them into uniform chunks of work. One way to handle the irregular workload of such applications is to apply dynamic load balancing to fix workload imbalance when it arises at runtime. PREMA implements implicit load balancing through a uniform object-oriented, message-driven execution model that virtualizes memory spaces and processing elements and allows transparent data and workload migrations. Applications are designed as if those objects are located in independent processes that do not share common hardware and can only interact with each other through messages that trigger a function execution (called handlers). Internally, the objects share the resources of a computing node, the work units of an object (received handlers) can be executed, if possible in parallel, by any idle thread in the node, and objects, along with their workload, can be migrated among nodes. Thus, the workload can be redistributed in distributed and shared memory using a uniform interface to create work units (handlers). Even though this approach is elegant, it can be sub-optimal when there is a significant disparity among handlers' workloads since a handler is a single unit of work that cannot be shared among threads.

In this chapter, we present an effort to alleviate the abovementioned issues. We show that by integrating PREMA with lightweight threads (in our case, Argobots [65]), one can easily avoid the limitations imposed by using run-to-completion tasks, substantially improving end-user productivity without a high cost in performance. Moreover, by utilizing the capabilities of this new integration, we implement high-level group communication primitives that further increase ease of use. Finally, to mitigate the load imbalance imposed in cases of significant workload disparity between handlers, we diverge from the uniform message-driven execution model and integrate PREMA with a tasking framework that enables applications to express handlers as a set of partially independent tasklets. By sacrificing a little ease of use to utilize a hybrid execution model, runtime systems can achieve significant improvements. Specifically, by compartmentalizing handlers, PREMA can redistribute workload outliers among the idle cores of a computing node.

The major contributions in this chapter are as follows:

- An approach for efficient task creation and message handling, using low-latency, preemptable, lightweight threads as opposed to run-to-completion tasks to mitigate overheads and latencies stemming from blocking operations.
- The utilization of multiple levels of data and task over-decomposition to improve the quality of workload distribution by increasing the flexibility of the runtime system to assign work units across both shared and distributed memory dynamically.
- A set of high-level group communication constructs to facilitate ease of use in the context of message-driven, global address-space, and data migrations.
- An evaluation of methods for integrating remote method invocations and message-handling, where we observe up to 100% difference in performance behavior and present a notable improvement of 50% through exhibiting multi-grain task and data over-decomposition.

This work has a broad impact on scientific computing use cases; some examples of applications that use PREMA and can leverage from its new features include mesh generation applications ([56], section4.5.7), solvers (section 4.5.6), N-body simulations [57], as well as other applications of irregular and adaptive computation and communication nature. Subsequently, it can be used as the backend of a domain-specific language for unstructured and dynamic applications.

4.1 WHY ARGOBOTS

Some of the most popular and optimized general-purpose systems include QThreads [66], MassiveThreads [67], StackThreads/MP [68], Cilk [42], Intel Threading Building Blocks [43], and Argobots [65]. These libraries offer large-scale, lightweight thread support, allowing many limited-resource threads to coexist in a system managed by a set of heavy-weight OS threads. Threads generated from these libraries can block and wait for other threads to complete and migrate between OS threads before completing their execution. QThreads, StackThreads/MP, MassiveThreads, and Argobots offer (in different extends) sets of synchronization primitives (mutexes, conditional variable) that allow OS threads to context-switch between lightweight threads instead of blocking when the waiting condition is not immediately satisfied. StackThreads/MP and Cilk require compiler support, while the rest of the systems are provided as C language libraries which makes them much easier to use in applications and higher-level runtimes. MassiveThreads and Argobots are enhanced with additional support for efficient interaction with I/O operations. Threads created by the two systems can automatically detect blocking IO calls and context switches at the user level, simplifying the interaction between computation-heavy and IO-heavy tasks in a single threading model. Finally, while all other libraries adopt a predefined scheduling policy, i.e., work-first LIFO scheduling within a single OS thread and FIFO randomized work-stealing between OS threads, Argobots does not restrict users to a specific policy and provides the tools to implement their own.

Based on our review, Argobots was chosen as the best option for a low-level threading system since it achieves high performance [65, 69] and incorporates the features provided by all the other systems. It also exposes abstractions that allow low-level customization and optimization by the user while maintaining a portable and broadly applicable interface. Other libraries facilitate higher usability, but this comes at the cost of less flexibility and a lack of low-level control. For instance, all other libraries implement transparent scheduling decisions, hide work pools and provide no control over stack and context-switching. Having access to these implementation decisions and being able to customize them plays a crucial role in optimizing a complicated runtime system such as PREMA to achieve high performance. Argobots allow that while providing two types of tasks to optimize performance, exposing an explicit task-yield operation and integrating with MPI and power management systems.

4.2 INTEGRATION OF PREMA AND ARGOBOTS

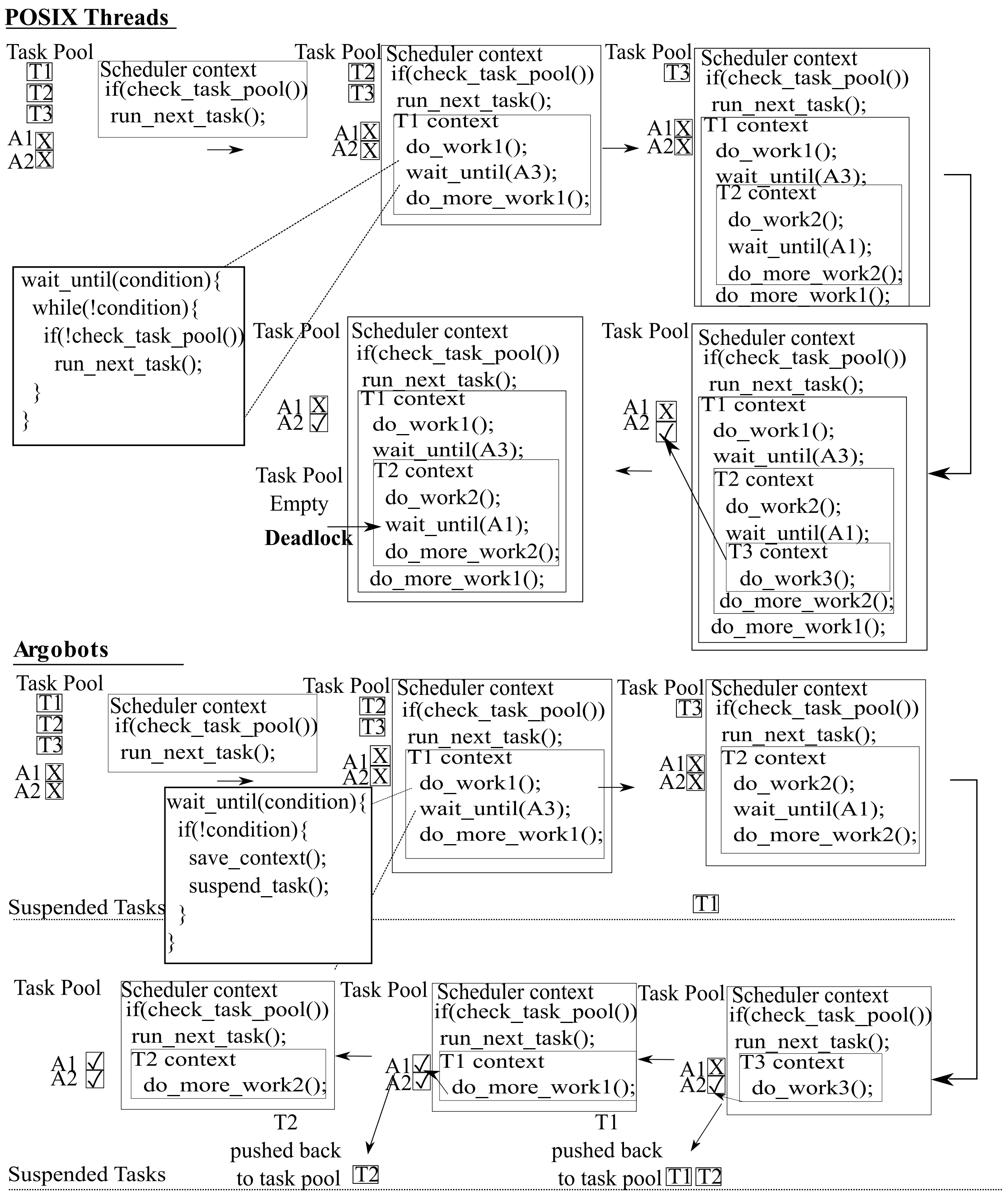
In this section, we present how each of PREMA's software layers, namely the Data Movement and Control Substrate (DMCS), the Mobile Object Layer (MOL), and the Implicit Load Balancing (ILB), are adjusted to be integrated with Argobots.

DMCS incorporates MPI to utilize multiple computing nodes in a high-performance computing cluster, as well as Argobots to take advantage of the hardware cores of each node. The programming model of DMCS is similar to that of Active Messages (i.e., each message sent is associated with a function call to be invoked on the receiver side once the message is received). This layer comprises three components: application, communication, and handler execution. The application component runs on the default, implicitly-created Argobots Execution Stream. The handler-execution component consists of multiple Execution Streams (ES), usually one less than the number of cores residing on a computing node. The communication component is either handled by a dedicated stream or any stream whose work pools are empty. The streams of the application and handler-execution components run customized Argobots schedulers, each assigned with a primary work pool while accessing each others' pools for work-stealing needs. On top of these work pools, each scheduler is assigned a private work pool to trigger a change of the active scheduler, a feature used by the higher-level layers. When a scheduler's work pools are empty, it attempts to perform work-stealing on a randomly selected work pool of a local peer. If this fails, it performs exponential backoff to minimize wasted cycles while waiting for new work units. The communication component encloses all message-passing-related operations. It is either handled by a dedicated stream or the streams in the handler-executing and application components when the rest of their work has been completed. When a remote method invocation request is received in the communication component, a new User Lightweight Thread (ULT) is created that is pushed to a randomly selected pool of the handler execution component (if any) or to the application component's work pool. In section 4.5.2, we evaluate different handler/task creation approaches using Argobots.

By encapsulating remote handler invocations in ULTs, their execution can be interrupted at any point, allowing other handlers to execute on the same hardware thread. Furthermore, the interrupted handler will continue its execution from where it stopped when there is an available core in the same node. This functionality suits the needs of parallel applications that otherwise need to use a (busy) waiting mechanism for some resources to become available. Using the yielding function, an application can interrupt the running ULT that needs to wait for another one to run. In contrast, in the pthreads implementation, using busy waiting inside a handler for reasons other than sending a message was discouraged as it could cause starvation and even deadlocked in some cases. ULTs and their yielding capabilities allow us to overcome these constraints. Some scenarios that can leverage the yielding functionality include blocking on a lock and message acknowledgment operations. Figure 15 shows how ULTs can avoid deadlock cases induced when blocking in the

PThreads implementation. The issue arises by the implementation of the *wait_until([condition])* operation, which blocks the running thread until the given condition evaluates to true. To avoid wasting cycles while waiting, PREMA tries to find another task to run by popping the next task available in the pools; however, when used from inside a handler, it can lead to a live-lock scenario like the following. Let us assume a scenario of three tasks, namely T1, T2, and T3 (see Figure 15 left) where T1 needs to wait for some acknowledgment A3 from T3, T2 waits for acknowledgment A1 from T1, and T3 does not wait for any acknowledgments. PREMA starts running T1 until it blocks waiting for acknowledgment A3, then switches to T2 until it blocks waiting for acknowledgment A1, and finally switches to T3, which acknowledges A3. Even though A3 has been acknowledged, the control will never return to T1 to unblock it. Once T3 finishes its execution, the control returns to T2's *wait_until()* operation, which will keep checking the task pool but will never run T1 since T1 has already been popped and is running T2 from its *wait_until()* operation. The Argobots implementation avoids such a scenario by using separate stacks for each task, saving their states before switching control of the execution stream and resubmitting them to the task pool when unblocked. This also allows blocked tasks to be stolen if the currently running thread starts a long-running process.

The ILB layer is implemented as an Argobots stackable scheduler that is pushed to the dedicated pool of each available execution stream to change the DMCS scheduling policy. It inherits the pools created by DMCS to continue executing remote handlers of the lower layers while also handling pools dedicated to the ILB. Handlers need to be issued through the ILB messaging operation to monitor their loads. Their execution consists of two steps. In the first step, the requests need to be routed to the current location of the target mobile objects and have their load evaluated. In the second step, handlers are scheduled for execution. For the first step, MOL finds the location of the mobile object and guarantees that it is in a valid state. Next, the ILB is notified about the new handler, and the handler's load is calculated and pushed to the list of pending handlers of the mobile object. The second step is executed later from the stacked Argobots scheduler created for the ILB. This scheduler maintains a list of all local mobile objects in the computing node to monitor the load of the whole node. When there is no other ongoing work, it picks one handler from the list of pending handlers of the next available mobile object and creates a ULT. ULTs created from this part of the system are then pushed to the fine-grained tasking module, presented in section 4.4. If no pending handlers are available, ILB starts a new cycle of distributed load balancing to find a remote mobile object with enough workload. The distributed memory load-balancing scheduler is also implemented inside a ULT as part of the ILB to allow the load-balancing policies to use operations that could block. Since this operation includes sending messages and checking containers



```
wait_until(condition){
  while(!condition){
    if(!check_task_pool())
      run_next_task();
  }
}
```

```
wait_until(condition){
  if(!condition){
    save_context();
    suspend_task();
  }
}
```

Fig. 15. A deadlock caused by blocking in a handler resolved by using Argobots.

that might require some synchronization, ILB should provide applications with this feature.

4.3 CONSTRUCTS FOR GROUP COMMUNICATION

4.3.1 Futures

A feature of Argobots that takes advantage of the ULT's ability to yield is the *eventual* data type. An eventual corresponds to the concept of futures found in many programming languages. A future is a mechanism for safely passing values between concurrent threads. Argobots implement this mechanism and enhance it with the yielding power of ULTs. For example, a running ULT may designate an eventual where it will store a value once it executes; other ULTs needing this value for their computations can then attempt to retrieve it by accessing the future. Trying to access an eventual, which has yet to be assigned a value, will cause the accessing ULTs to block. In such a case, the ULT is suspended by the Argobots framework and removed from the pool of active work units. It becomes available again only when the respective value of the event has been set. If the future has already been assigned a value, it will allow the ULTs accessing it to retrieve the value without blocking. PREMA utilizes this feature of Argobots and exposes it as a high-level construct while extending it to distributed memory.

In the old design, an application using PREMA needed to manually develop acknowledgments for remote handlers execution when such functionality was desired (Figure 16 left). In the new version of PREMA, when the application needs to invoke a function on a remote processor or mobile object, the messaging interface can optionally return a future that is signaled once the remote function call completes(Figure 16, right). Internally, PREMA creates a new Argobots eventual and attaches its identifier to the header of the remote message handler. Once the handler invocation completes on the target, PREMA checks whether the respective section of the message header is set. If it is, it will invoke a new remote handler back to the sender with the value related to the future as an argument. When the remote handler executes, it sets the future to a ready state on the original sender and, thus, unblocks all ULTs waiting for it. With this mechanism, the synchronization of the application logic becomes easier to handle and can be better optimized by PREMA. The distributed memory load-balancing scheduler is also implemented inside a ULT as part of the ILB to allow the load-balancing policies to use operations that could block. Since this operation includes sending messages and checking containers that might require some synchronization, ILB should provide applications with this feature.

```
void run_important_computation (...) ;
bool computation_done = false ;

DEFINE_HANDLER(computation_hdler_done) {
    // Signal waiting thread
    computation_done = true ;
}

DEFINE_HANDLER(computation_hdler) {
    // Execute computations as needed
    run_computation(dmcs_get_msg_args());

    // Execution is done, notify source
    dmcs::send(dmcs_get_msg_source(),
        computation_hdler_done);
}

int main() {
    double values [4];
    size_t v_size = sizeof (data);
    int target = 1;

    // Execute run_computation_hdler on target
    // args: values [4]
    dmcs::send( target ,
        run_computation_handler, data , v_size );

    // Wait until notified completion
    wait_until (computation_done == true );
}
```

```
void run_important_computation (...) ;

DEFINE_HANDLER(computation_hdler) {
    // Execute computations as needed
    run_computation(dmcs_get_msg_args());
}

int main() {
    double values [4];
    size_t v_size = sizeof (data);
    int target = 1;
    dmcs:: future done;

    // Execute run_computation_hdler on target
    // args: values [4]
    dmcs::send(&done, target ,
        computation_hdler, data , v_size );

    // Suspend ULT until future is set
    done.wait ();
}
```

Fig. 16. Examples of handler completion acknowledgment (left) without and (right) with futures.

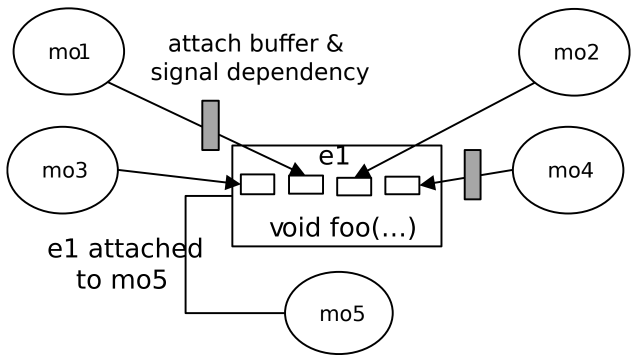


Fig. 17. A visualization of PREMA's event primitive. Event e_1 is attached to mobile object mo_5 and expects four dependencies to be signaled in order to invoke foo on mo_5. Once foo is invoked, mo_5 has access to all buffers attached as dependent data.

4.3.2 Events

Distributed futures removed most of the boilerplate code users had to write to acknowledge handler completion. They also improve concurrency and eliminate some of the constraints of the original implementations of PREMA. However, they can only signal one dependency at a time, either acknowledging the completion of a handler or used in a user-defined functionality (e.g., blocking on a future signaled from a remote message manually). To handle cases where multiple dependencies need to be satisfied, we have introduced the primitive of distributed events. An *event* is a synchronization tool associated with a mobile object, a handler task, a predefined number of dependencies, and, optionally, data buffers. It is uniquely identifiable in the whole distributed system and can be used by the application to satisfy dependencies remotely (see Figure 17). For example, an event can be triggered as the last step of a handler to signal its completion to dependent tasks and pass the required data to them. The data can even consist of a mobile object that needs to be transferred to the location of the associated mobile object. In this case, the runtime system will handle the migration process implicitly, even if the object resides remotely. PREMA transfers the signals from different dependencies along with their data or mobile objects and triggers the associated handler task when the predefined number of dependencies has been fulfilled.

4.4 FINE-GRAINED RECURSIVE TASK PARALLELISM

The need for finer-grained parallelism inside a handler arises from the significant workload disparity witnessed among handlers of irregular and adaptive applications [47]. Despite using over-

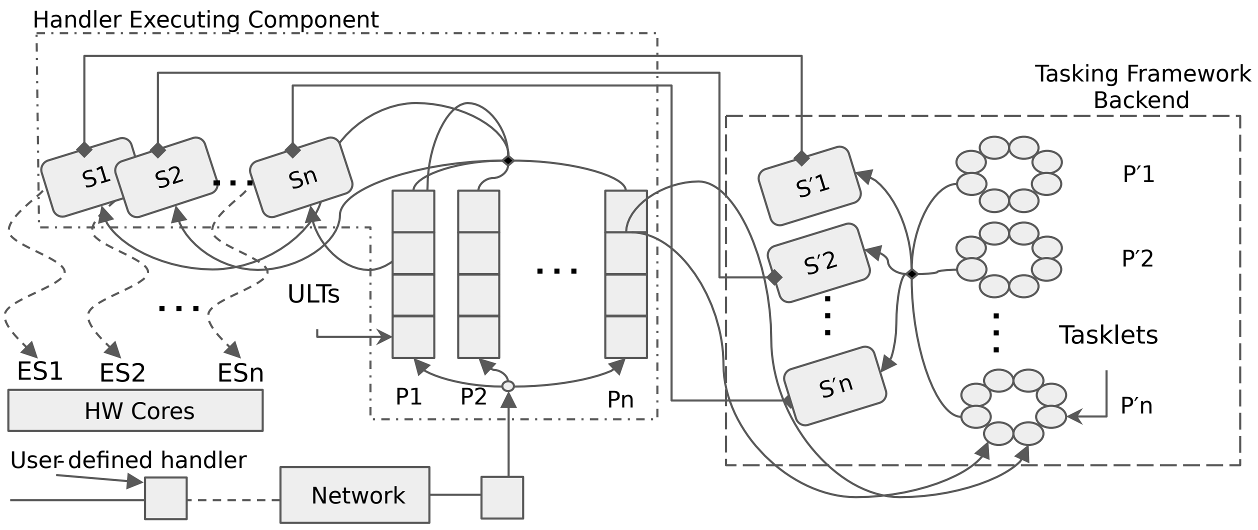


Fig. 18. Integration of PREMA and the tasking framework. From left to right: a user-defined handler, implemented as an Argobots ULT, comes through the network and is assigned to one of the task pools $\{P_1, P_2, ..., P_n\}$, managed by the schedulers $\{S_1, S_2, ..., S_n\}$; tasks are mapped to Argobots execution streams $\{ES_1, ES_2, ..., ES_n\}$ which in turn execute on hardware cores. A user-defined handler can spawn one or more fine-grained tasklets monitored by the tasking framework backend. The tasking framework (right) has its own task pools $\{P'_1, P'_2, ..., P'_n\}$ and schedulers $\{S'_1, S'_2, ..., S'_n\}$.

decomposition to diffuse workload[70], the workload disparity among handlers targeting different mobile objects can still be prominent. This is an effect of the decomposition being computed at the initialization stage of the application. To bridge this gap, we developed a standalone tasking module on top of Argobots. It is designed to help utilize multiple hardware threads in a single handler execution. In [69], we study the standalone version of the module as a black box and experiment with different models to spawn parallel tasks in shared memory. This work delves into its low-level implementation and integration with PREMA and the issues raised by introducing preemption on a tasking framework utilizing lock-free task pools. Tight integration of the two systems is required for an efficient, low-latency hybrid tasking model. When used as an integrated part of PREMA, this module can utilize the existing execution streams, avoiding the creation of new threads and the possible over-subscription overheads. To distinguish between ULTs used in other parts of PREMA and tasks created by this module, the latter will be called tasklets for the rest of the paper.

The interaction between the shared and distributed memory modules within PREMA is depicted in Figure 18, namely the handler executing component and the fine-grained tasking framework backend. Two modules consist of sets of schedulers $\{S_1, S_2, ..., S_n\}$, $\{S'_1, S'_2, ..., S'_n\}$ and task pools $\{P_1, P_2, ..., P_n\}$, $\{P'_1, P'_2, ..., P'_n\}$. Handlers are issued either locally or remotely through the network and are implemented as Argobots User Level Threads (ULTs). Tasklets can be spawned in the context of a handler execution and are managed by the tasking framework backend. The schedulers of the tasking framework backend are attached as plugins to PREMA's schedulers, allowing them to utilize the existing Argobots Execution Streams, avoiding resource over-subscription.

4.4.1 Low-Level Implementation

The scheduler of this module attempts to maximize parallelism and minimize memory use by using a hybrid of depth-first and breadth-first execution policy and recursive creation of work units. Each processing element (PE) is associated with a list of tasklets. Each time a new tasklet is created by a PE, it is pushed to the bottom of its list. To pick a new tasklet to execute, the PE pops a tasklet from the bottom of its list; if it is empty, it will try to steal a tasklet from the top of another PE's list. Provided that the tasklets creation is performed recursively, this scheduling algorithm prioritizes hot tasklets in the cache (latest created) when there is pending work in the pool while maximizing the amount of stolen work when stealing is attempted by targeting tasks that were created early in the recursion steps. To implement this scheduling algorithm, the Argobots abstraction of custom pools was used to encapsulate a lock-free implementation of a circular double-ended queue (deque)[71]. The interface of the abstract pools only provides push and pop (and remove but is not needed in our case) operations to manipulate the contents of a data structure. Thus, stealing is implemented as part of the pop operation, and each abstract pool is implemented as an array of pointers to all available deques instead of associating it with a single deque. Then, when an ES is ready to pick a new task, it calls the pop function, which, in turn, checks its deque; if empty, it will randomly steal a tasklet from another deque.

Tasklet dependencies are also provided in this module; a “parent” work unit can create several “children” tasklets and wait for their completion. The “children” can then create their own “children”, constructing a tasklet dependency tree. The function that creates a tasklet returns a handle that can be joined for completion, causing the caller to yield if not complete. The root tasklet of a tree is implemented as a ULT to enable yielding; however, children tasks are implemented using common list structures for performance. To enable switching between children tasklets when waiting for dependents’ completion, we explicitly push and pop children tasks from the lists and run them inside the parent or a separate ULT, which allows yielding from any node in the tree. Once a tasklet execution completes, the control returns to the parent tasklet, which, in turn, evaluates whether it will continue waiting for other children to complete. Waiting in this context will cause popping/stealing another tasklet.

4.4.2 Safely Yielding Tasklets

An important incentive to build such tasking frameworks on top of lightweight threads (LWTs) is the ability of LWTs to yield, allowing the creation of tasks that might not run to completion. This enables tasks to run blocking operations such as messaging and acquiring locks. However, the requirements to maintain this ability while using the approach mentioned earlier to create tasks yield some issues that can prevent the framework from operating correctly. In this section, we identify the problems raised by this task-creation strategy and present our approach to handling them.

Avoiding live-locks

The yielding operation in Argobots saves the context of the yielding ULT, pushes the ULT back into its respective work pool, and switches control to the execution stream’s scheduler; however, following the same steps in the case of our fine-grained tasking module could cause a livelock. In our work-pool implementation, push, and pop operations target the bottom of the deque. Pushing a yielding ULT back to the same pool will insert it at the bottom of the deque, making it the first ULT that will be popped in the subsequent pop request. In a scenario where a ULT yields while busy waiting for some resource, the yielding ULT would be constantly popped and pushed to the bottom

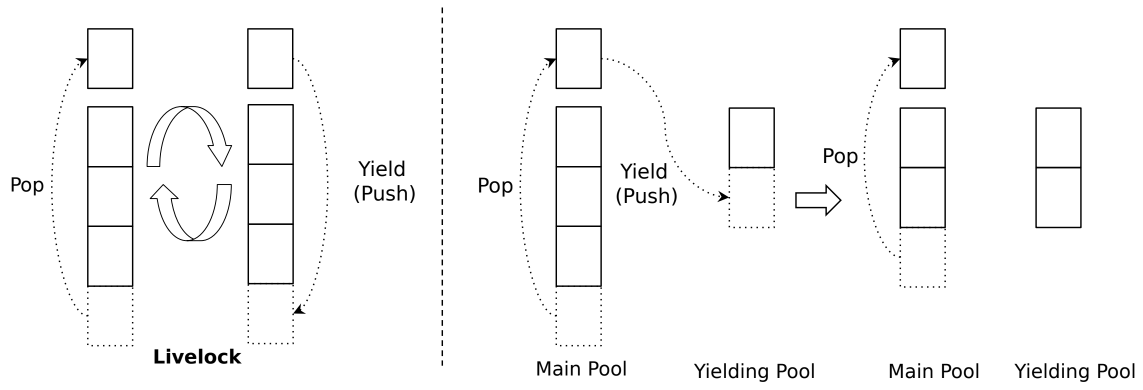


Fig. 19. Demonstration of the potential live-lock. An example of a live-lock scenario during the yield operation in the tasking framework (left) and how it is avoided by using a secondary pool to store yielding tasklets temporarily (right).

of the deque, allowing no other ULTs to be executed (Figure 19 left). To avoid such a scenario, we distinguish between new and yielding ULTs by keeping track of the last ULT executed on each ES and comparing it with ULTs about to be pushed to the respective work pool. If they are the same, we can infer that the ULT about to be pushed is yielding. Yielding ULTs are pushed to a secondary work pool, maintained per ES, to unblock the main work pools and allow other ULTs to execute (Figure 19 right). Once all ULTs in the main pools have been executed, the ULTs in the secondary/yielding pool are run; ULTs running in the secondary pool and yield are pushed back to the main pool to avoid the same live-lock scenario from occurring there.

Signaling blocked tasks

Another issue arises from the default implementation of signaling blocked ULTs in Argobots. Blocked ULTs remain in a private structure of Argobots; when an active ULT signals a blocked ULT, it changes its state from blocked to active and pushes it back to the last work pool where it resided before being blocked. This work pool might belong to an ES different than the one running the signaling ULT, which breaks the requirement that each execution stream can only push and pop to/from its work pool and only steal from other work pools (see Figure 20, left). To overcome this issue, the push operation ignores the target pool that Argobots chooses and assigns signaled ULTs to the work pool of the execution stream running the signaling ULT, as shown in Figure 20, right. Thus, pushing to a work pool can only be performed by the execution stream that owns the pool, conforming to the deque requirements.

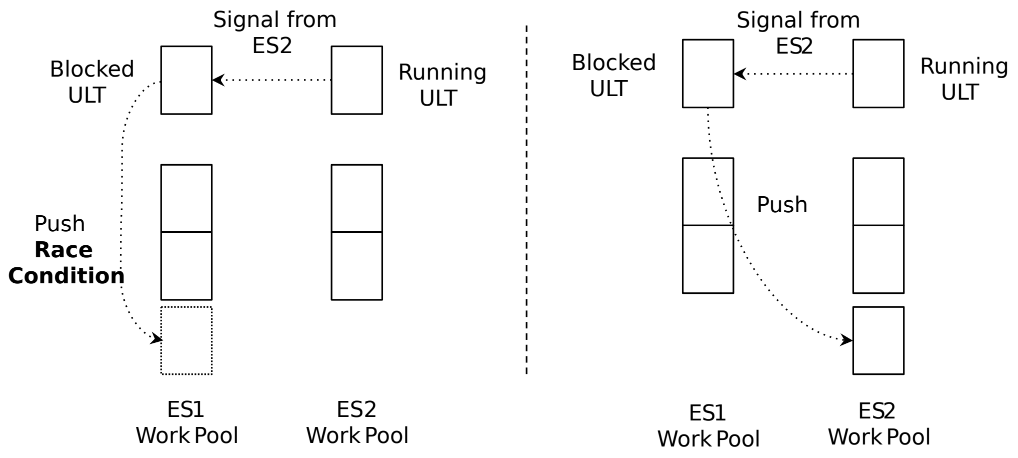


Fig. 20. Race condition scenario in signaling blocked ULTs. By default, when a blocked ULT is signaled to unblock, it is pushed back to the last pool it executed. The push operation is executed by the ES of the ULT that signals the blocked ULT; as a result, an ES may push the just unblocked ULT to the work pool of another ES, which causes a race condition in our lock-free implementation (left). To overcome this issue, our implementation ignores the pool that Argobots chooses to push the blocked ULT and always pushes it to the pool of the signaling ULT's ES. This modification conforms to the requirements of the custom deque and avoids race conditions.

4.5 PERFORMANCE EVALUATION

In this section, we measure the performance of PREMA, taking advantage of the contributions presented so far. Due to the difficulty in isolating minor overheads derived from different approaches for handler task creations and message-handling using full-scale, real-world applications, in 4.5.2 and 4.5.3, we evaluate PREMA on benchmarks derived from the widely-used OSU Microbenchmarks [72]. In section 4.5.4, we derive a synthetic microbenchmark that stresses PREMA's handler execution component, presenting an extreme condition of highly contended mutexes, and present the impact on its performance before and after utilizing lightweight threads. Section 4.5.6 presents an evaluation on SW4lite, an application designed to reflect the workflow of a few critical kernels of SW4. It is part of DoE's exascale project (ECP) initiative for benchmarks that accurately represent various scientific applications while avoiding dealing with large and complex code bases [73]. Sections 4.5.5, 4.5.7 present evaluations on real-world applications in shared and distributed memory, respectively.

4.5.1 Experimental Setup

Two computing clusters are used to run the performance benchmarks. The first one, namely Turing, is a 250-node cluster consisting of Intel(R) Xeon(R) (E5-2660, E5-2660 v2, E5-2670 v2, E5-2698 v3, E5-2683 v4) 128 GB CPUs ranging between 16 to 32 cores spread among two sockets (2 NUMA nodes). The second platform, namely Wahab, is a 200-node cluster that utilizes Intel(R) Xeon(R) Gold 6148 @ 2.4 GHz CPUs of 40 cores each in two sockets (4 NUMA nodes).

4.5.2 Handler Task Creation

This subsection examines three approaches to creating handler tasks as lightweight threads. Performance is measured as the latency in handler creation, with and without a dedicated communication stream, in a ping-pong benchmark. Two nodes exchange 20000 64B-sized messages, where the sender sends a message and then waits for an acknowledgment. The three approaches are described below:

- Unnamed: The Argobots runtime is responsible for monitoring and releasing the memory of ULTs when they are complete.
- Named: PREMA checks ULTs for completion and frees their resources explicitly. An array of handles is maintained per handler executing ES.
- Revive: A variation of the second approach where the completed ULTs in the preallocated

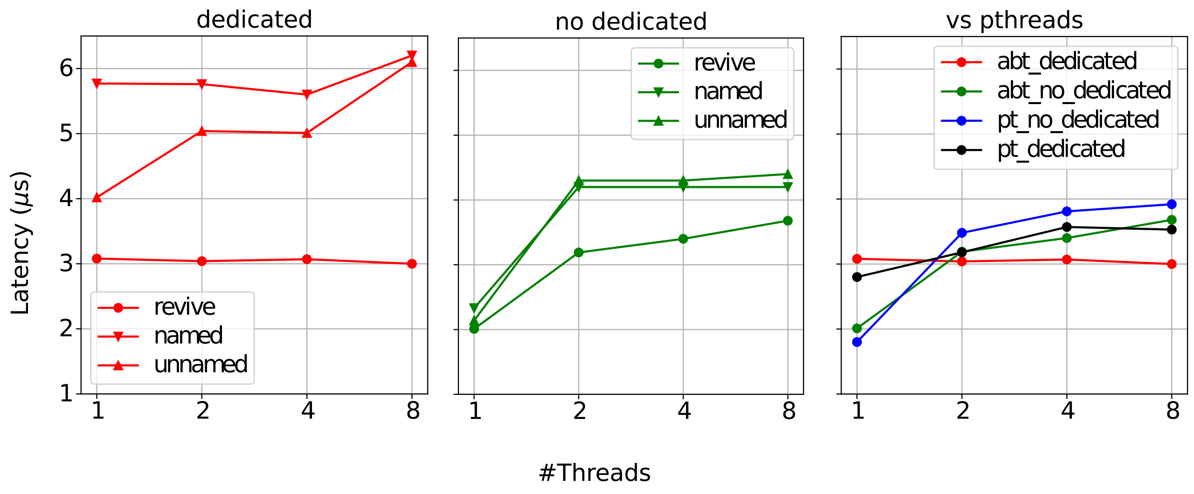


Fig. 21. Latency observed on the ping pong benchmark for different task creation approaches. (a) Using dedicated streams for communication or (b) not using, and (c) comparison of the best approaches with the PThreads implementation.

arrays are reused through the *ABT_revive()* function.

Figure 21 shows the performance observed in terms of latency for the three different approaches when using a dedicated stream for communication (21a) or not (21b) and compares the best for the two with their PThreads counterparts. In all cases, a single message is on-the-fly at any time. Two nodes exchange 20000 64B-sized messages, where the sender sends a message and waits for an acknowledgment before sending the next.

Figure 21a shows the latency observed when a dedicated stream is in use to handle communication and assign the respective tasks to other streams. The approach used plays a significant role in this case, where the “revive” approach of reusing previously completed ULTs benefits the overall performance, maintaining a very stable latency of $3\mu s$. In contrast, using the “named” or “unnamed” approach not only achieves a lower performance (of 6 and 4 μs, respectively) when a single handler executing stream is used but also faces increasing overheads when the number of streams also increases. The performance degradation observed in this case comes from ULTs using their own stack. Each time a stream creates a new ULT, it needs to allocate its stack memory, which increases the critical path of the ULT creation. Reviving ULTs, on the other hand, does not need to allocate new memory as the memory of previously completed ULT is reused.

Figure 21b shows the latency observed when no dedicated stream is used. In this case, the significance of the three different approaches is not apparent when only a single stream is in use. The reason is that Argobots use internal memory pools for the stacks of completed ULTs per ES. When a stream completes the execution of a ULT, it stores the memory used for the stack in its memory pool to reuse it later and avoid reallocations. However, when one ES creates most ULTs (producer), and other ESs execute them (consumers), the producer pool is depleted without being reused. The consumer ESs keep the stack memory in their memory pools, forcing the producer to steal from them or allocate more memory. This extra overhead is observed for "named" and "unnamed" approaches when two or more streams are used in figure 21b. In this case, the overheads are lower compared to Figure 21a because all streams have the same chance to produce or consume ULTs. Thus, they all have more opportunities to refill their pools and avoid allocations or steals.

Figure 21c shows the comparison of the revive approach when using dedicated streams or not. We also compare the performance of the two with their PThreads implementation counterparts. Using no dedicated thread for communication achieves the most negligible overhead when only a single thread is utilized, and PThreads exhibit the lower overhead in this case. When more than one handler executing stream/thread is available, using a dedicated stream showed lower latency for both the Argobots and PThreads implementation, with a difference of about 15%. Using no dedicated stream falls a little behind with the PThreads implementation achieving up to 10% worse performance than the Argobots counterpart.

4.5.3 Message Handling

Message handling is part of the DMCS and utilizes the MPI as a communication library. In this section, four ways to handle message passing are evaluated.

- ULT-Per-Message (UPM): A new ULT is created to run the MPI operation, yielding if the operation did not complete immediately or exiting otherwise. If a separate stream is used for communication, message reception is handled by a separate ULT that runs in a loop and yields; otherwise, one of the handler-executing threads directly polls the network.

- Communication-In-Pool (CIP): Two ULTs are spawned that run in a loop, one for receiving and one for sending messages. Send requests are passed through regular C++ queues. Each ULT serves the respective operations and yields its execution.

- Combining the two (CIP-UPM): A new ULT is created for each outgoing message (like UPM), while incoming messages are handled through a separate ULT running in a loop (like CIP).

- Queue: Outgoing message requests are passed through a C++ queue (like CIP). When no dedicated stream is used, this queue is served directly from the handler-executing streams serving the incoming messages. Otherwise, a single ULT is spawned in the dedicated stream that continuously polls the network and serves message requests in the queue.

The performance of each approach is presented in Figure 22 in combination with the different task creation approaches on the same benchmark. We present results for both cases where a dedicated stream is used for message handling (bottom) or not (top). As can be seen from the graph, the best performance in terms of latency observed is achieved through the combination of the “queue” message handling, and the “revive” handler task creation approach with $3\mu s$ latency achieved on average. It is interesting to note that depending on the existence of a dedicated stream, different combinations provide the best performance, except for the best case (queue-revive).

When no dedicated stream is present (top), the “queue” approach performs best, regardless of the task creation approach. CIP and UPM perform similarly but with an increasing overhead compared to queue as the number of handler-executing streams increases (up to $5\mu s$ overhead). This is expected since the two approaches require creating ULTs and context switching compared to the queue approach, which only needs to check a queue and poll the network. The combination of CIP and UPM (CIP-UPM) adds the most significant overhead in all cases, constituting the longest critical path before handling message passing (up to $6\mu s$ overhead). On the other hand, when a dedicated stream is used for message passing (bottom), we see that UPM and CIP-UPM perform better when the named or unnamed task creation approach is used (on average $3.5\mu s$ overhead). In these cases, the overhead of the other two message-handling ways is experiencing almost double the overhead as the number of handler-executing streams increases(up to $6\mu s$). Thread stack creation and reuse explain this effect, as in the task creation benchmark. In this benchmark, it has been shown that the queue + revive combination of message handling and handler task creation is the best in terms of latency, regardless of whether a dedicated stream is used for communication.

4.5.4 Blocking Operations in Handler Execution

An important feature stemming from integrating with Argobots is the ability to yield handler execution either explicitly by calling the respective function or implicitly when a blocking call is detected. The potential benefit provided is presented through a synthetic benchmark. An allocation of ten cores is used along with ten mobile objects, each using an exclusive mutex to provide access to its data. For each mobile object, 100 handler invocations are issued, where a specific percentage of them needs to take control of the mutex before executing. We set the time that the mutex is held to 50ms at a time and experiment with standard PThread mutexes and ULT-

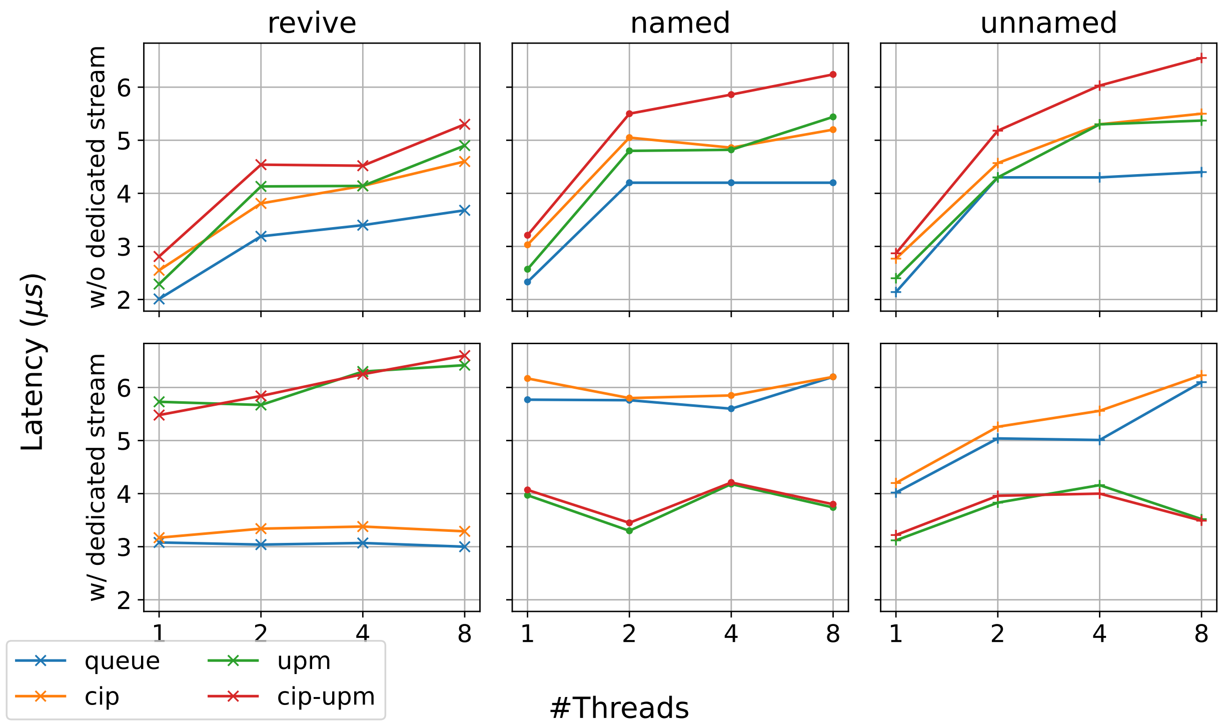


Fig. 22. Latency observed for different task creation and message handling approaches. Either using dedicated streams for communication (top) or not (bottom).

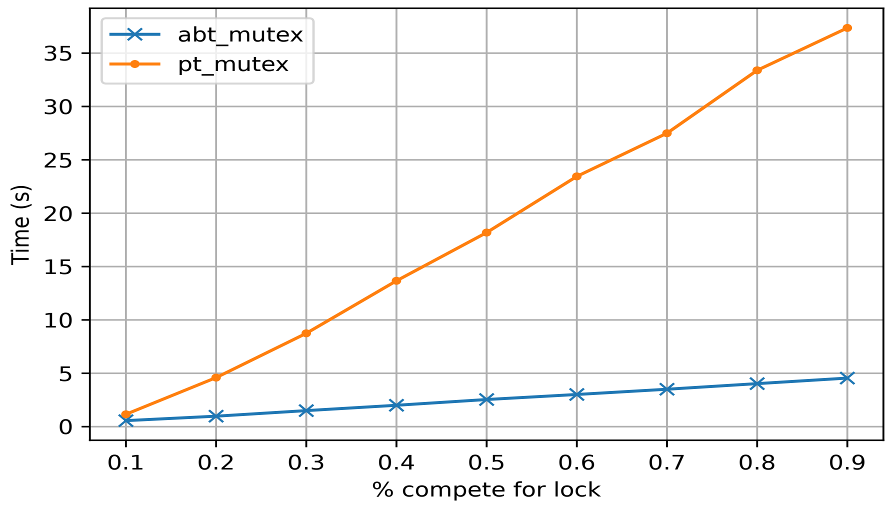


Fig. 23. Execution time with respect to the percentage of tasks of competing for a mutex. Mutex is implemented as an Argobots mutex (abt_mutex) or PThreads mutex (pt_mutex)

aware mutexes that can suspend handlers when locked. Figure 23 shows an evaluation of the two implementations with different percentages of handlers acquiring the mutexes, ranging from 0.1 to 0.9 (10 - 90 handlers/mobile object). One can observe the tremendous difference between the two implementations (up to 1000%). The issue with the PThread implementation is that a handler that tries to access a taken mutex will block the hardware thread it executes on, preventing handlers targeting a different mobile object/mutex from running. In contrast, the Argobots implementation will suspend the running handler ULT, allowing another handler to run on the same thread. In similar cases, the user might know that acquiring a resource exclusively could cause delays and, thus, may explicitly yield competing handlers to mitigate the propagation of such effects.

4.5.5 Fine-Grained Recursive Tasking

In this section, we evaluate the performance of the fine-grained tasking framework on the Barcelona OpenMP tasks suite of benchmarks[74] and compare it with OpenMP and TBB. Specifically, we focus on three benchmarks: Protein Sequence Alignment, Fast Fourier Transformation (FFT) on one billion points, and Sort on one billion elements. All three benchmarks are run on both available computing infrastructures and are implemented following a recursive parallelization model suitable for PREMA's tasklets; their results are presented in Figure 24. Another thorough comparison study for the three libraries is presented in [69] in the context of parallel mesh generation.

Protein Sequence Alignment

The performance achieved by PREMA tasklets in both systems is depicted on the left of Figure 24, achieving a speedup of 26 on the 32-core system and 37.4 on the 40-core. The difference in performance between PREMA tasklets and the two industrial-strength frameworks is negligible; PREMA tasklets outperform OpenMP in all thread variations on the Wahab cluster and fall behind on the Turing cluster but only by a small amount. A similar trend is noted when performance is compared to TBB on the two machines.

FFT

An interesting observation in this benchmark is the declining performance of OpenMP (Figure 24, middle). We see that its performance suffers in the specific application as it is affected by the large number of tasks generated. On the other hand, PREMA's and TBB's performance scales well on Turing for the first 16 cores ($\approx$ 13 speedup) and saturates to approximately 19 on 32 cores. For the Wahab system, PREMA continues to perform on par with TBB, achieving a speedup of 11 on 16 cores, with both frameworks' performance declining after this point, having no increase in speedup for 32-40 cores. We see that in this benchmark, the speedup achieved by any framework is much lower than what was observed in the previous one. The difference between the two benchmarks is that first, each task of the alignment case runs much longer than a single task in FFT, and second, FFT creates about a thousand times more tasks. Thus, the overheads related to task creation are much more difficult to hide behind computations, and also, they add up quickly due to their large number. The decline in performance on higher core counts can also be attributed to the decrease in the amount of work per thread, which limits the amount of possible concurrency while increasing the number of failed steal attempts, and to the use of more CPU sockets which leads to NUMA effects.

Sort

The declining performance of OpenMP is also exhibited in this benchmark; however, it does so once it reaches the utilization of 16 cores. The other two tasking frameworks continue to perform well on both platforms. The performance of the Turing machine continues to be superior both for the sequential case and parallel ones, even though it utilizes fewer threads. Like in FFT, both frameworks' performance scales well up to the first 16 cores with a speedup of approximately 13 on Turing and 11 on Wahab. In larger counts of cores, the performance of both frameworks declines on both platforms (Figure 24, right). This decline stems from the same factors as the

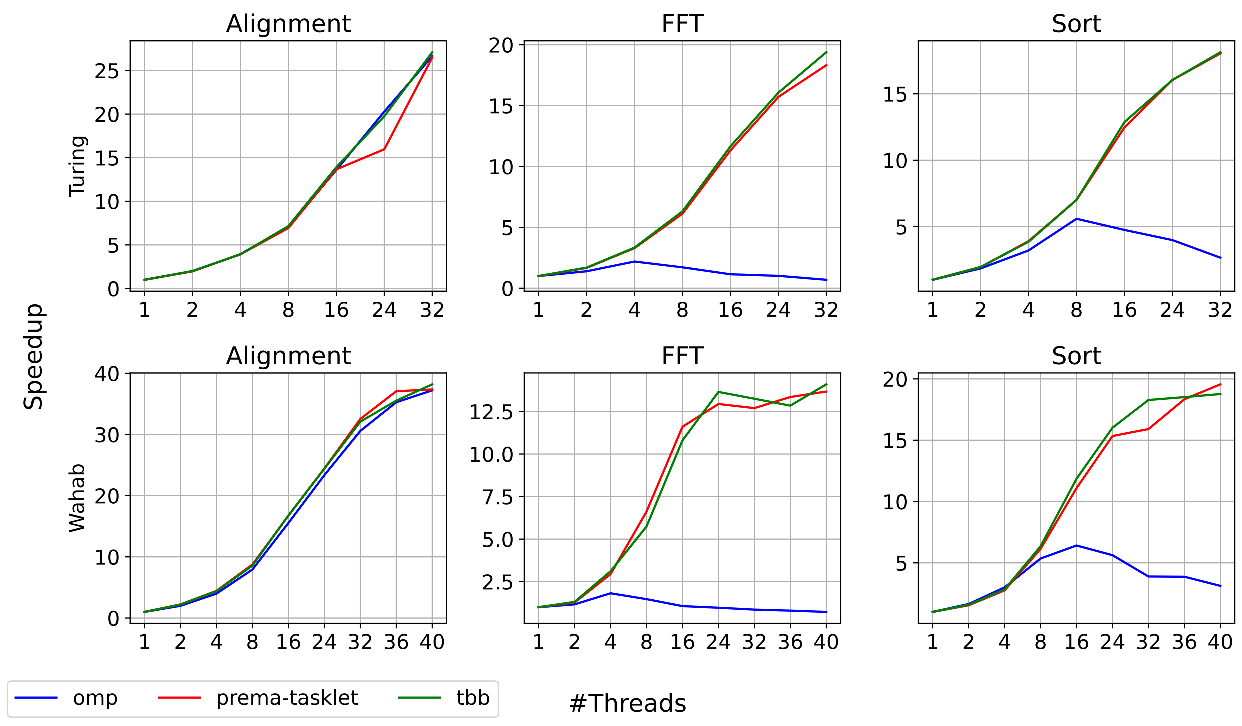


Fig. 24. Speedup achieved on two platforms, for different fine-grained tasking frameworks. The benchmarks include OpenMP (omp), Intel TBB (tbb), and PREMA tasklets for Protein Sequence Alignment, FFT, and Sort applications.

FFT case, i.e., a large number of small tasks creating overheads difficult to overlap. Moreover, this benchmark also introduces a significant amount of accesses to memory shared among multiple threads, leading to cache-line false sharing effects.

4.5.6 Seismic Wave Simulation Benchmark

In this section, we evaluate the new features of PREMA on the SW4lite benchmark [75]. We study its performance on different work unit allocations (mobile objects, PREMA tasklets) for single-node and multi-node experiments.

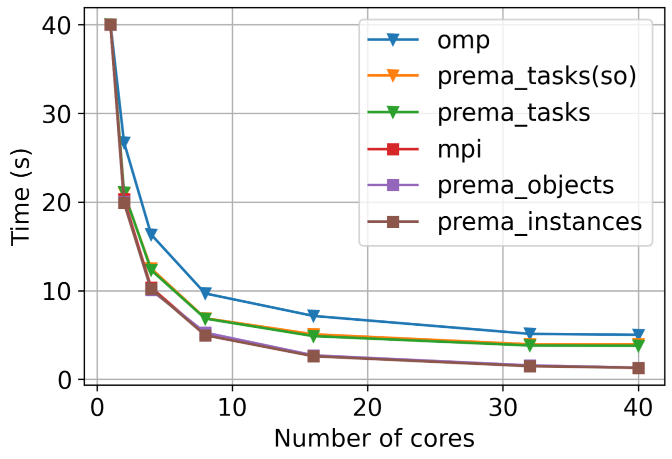
Single Node Performance

We evaluate the performance of SW4lite on a single node, utilizing different approaches to concurrency, domain decomposition, message passing, or tasking. For the task decomposition approach, we evaluate OpenMP (omp for), PREMA tasklets as a standalone library, and PREMA tasklets as part of a single PREMA instance. For the domain decomposition approach, we evaluate MPI, one instance of PREMA for the whole node using one mobile object per thread, and one instance of PREMA (utilizing a single mobile object) for each core. Figure 25 shows the results of this evaluation for a small input running the point-source functionality of SW4lite. It is clear that both versions of PREMA tasklets outperform the OpenMP version of the code (lines with triangle markers) by about 20% but only attain a speedup of 10 at 40 cores. On the other hand, the domain decomposition approach performs much better, achieving a speedup of 32 at 40 cores for all versions. This observation raised a concern about whether the performance disparity will always favor just data decomposition or whether an optimal combination of the two approaches gives the best results. In other words, whether it is always the best choice to decompose the data into more "pieces" as the number of cores increases or there is a point where it is more efficient to apply a two-level approach with a coarse-grained data decomposition and finer-grained task decomposition on top of it. In the next section, we experiment with different approaches for data combinations and tasking over decomposition.

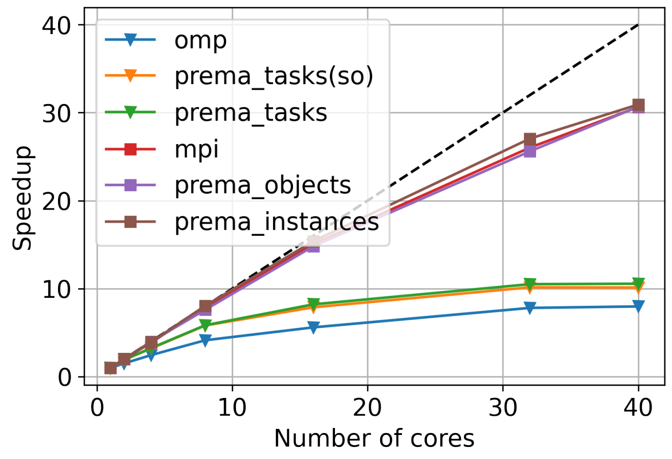
Multi-Node Performance

Our observations on the single-node evaluations lead us to run more experiments on larger inputs and different work unit allocations to find out how different combinations of data and task decomposition can affect the performance of an application like SW4lite. Adding tasklets on top of PREMA enables running such a study, simplifying the experimentation with data and task decomposition independent of the available hardware resources. In other words, one can decompose the application data/tasks domain in more "pieces" than the available nodes/cores without needing extra code to handle them or taking care of resource over-subscription. Trying to run such a study without tasklets, e.g., using OpenMP, results in a bad performance, as shown in Figure 26. In this benchmark, we exploit distributed and shared memory parallelism concurrently, using one mobile object per core for 640 cores and also try intra-handler parallelism using OpenMP for, OpenMP task loop, and PREMA tasklets. Using OpenMP causes over-subscription of hardware resources (PEs) since it is unaware of the existing threads running from PREMA, resulting in overheads related to constant context-switching and multiple threads competing for the same hardware core. In contrast, PREMA tasklets are aware of the threads already running and utilize them efficiently, achieving up to 10% improvement over the base case or 60% over OpenMP.

Using PREMA tasklets, we evaluate different approaches to (over-)decompose only at the data domain level, only at the task level, by creating tasklets and combining the two. The different work unit allocations allow different levels of freedom for PREMA to take advantage of, like load



(a)



(b)

Fig. 25. SW4 performance of threading (triangles) and message passing (squares) parallelism. The benchmark is run on a single node for small input and the performance is evaluated in terms of (a) time and (b) speedup. omp: OpenMP, prema_tasks: tasklets, prema_tasks(so): tasklets as a standalone library. mpi: 1 MPI rank/core, prema_instances: 1 PREMA rank/core, prema_objects: 1 mobile object,thread/core.

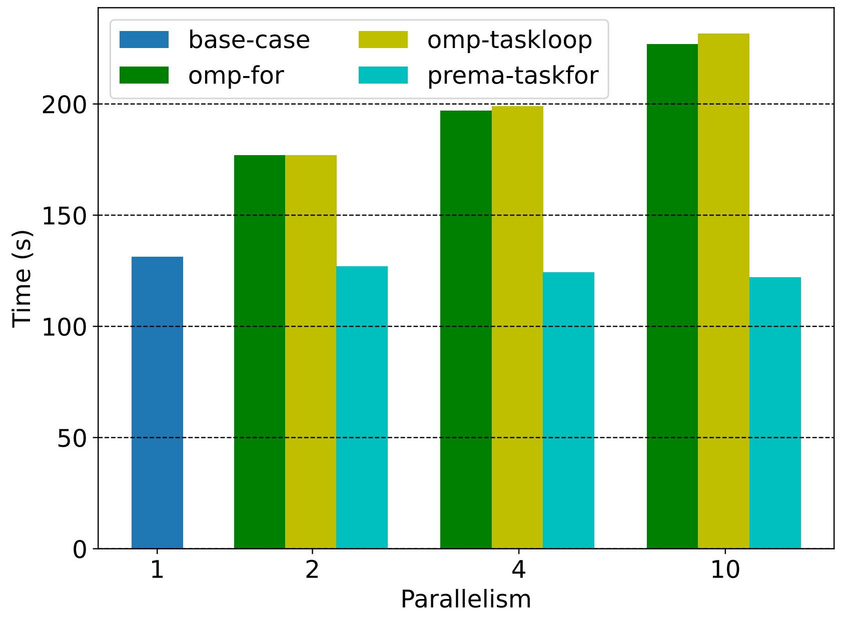


Fig. 26. Intra-handler parallelism with OpenMP for, taskloop and PREMA tasklets.

balancing and latency hiding, since both data subdomains and tasklets in a PREMA instance are shared among its threads. Figure 27 presents the overall running time on an increasing number of cores (strong scaling), normalized by the running time of the base case. The base case for this experiment is using one mobile object per hardware thread and one tasklet per handler, using one instance of PREMA per socket of 10 hardware cores to avoid NUMA effects. The x-axis shows the work-unit allocations, where the first number represents the number of mobile objects per PREMA instance, and the second one shows the number of tasklets spawned per mobile object handler. As can be seen from the figure, the application benefits from increasing the number of mobile objects per instance (green bars) when the number of cores is low and the work enclosed in each mobile object is substantial to overlap the overheads of message-passing and context-switching. However, it benefits less as the number of cores increases and the work per mobile object thins out; thus, the overheads related to the increased number of messages and context-switching can no longer be tolerated. Utilizing task decomposition only (orange bars) cannot surpass the performance of domain decomposition in any case. However, it improves as the number of tasks increases and the data size per PREMA instance decreases. This can be explained from the data observed in Fig. 25 where the task parallelism approach performs far worse than the approach utilizing data decomposition. Thus, when the are fewer but larger data domains, task parallelism performance

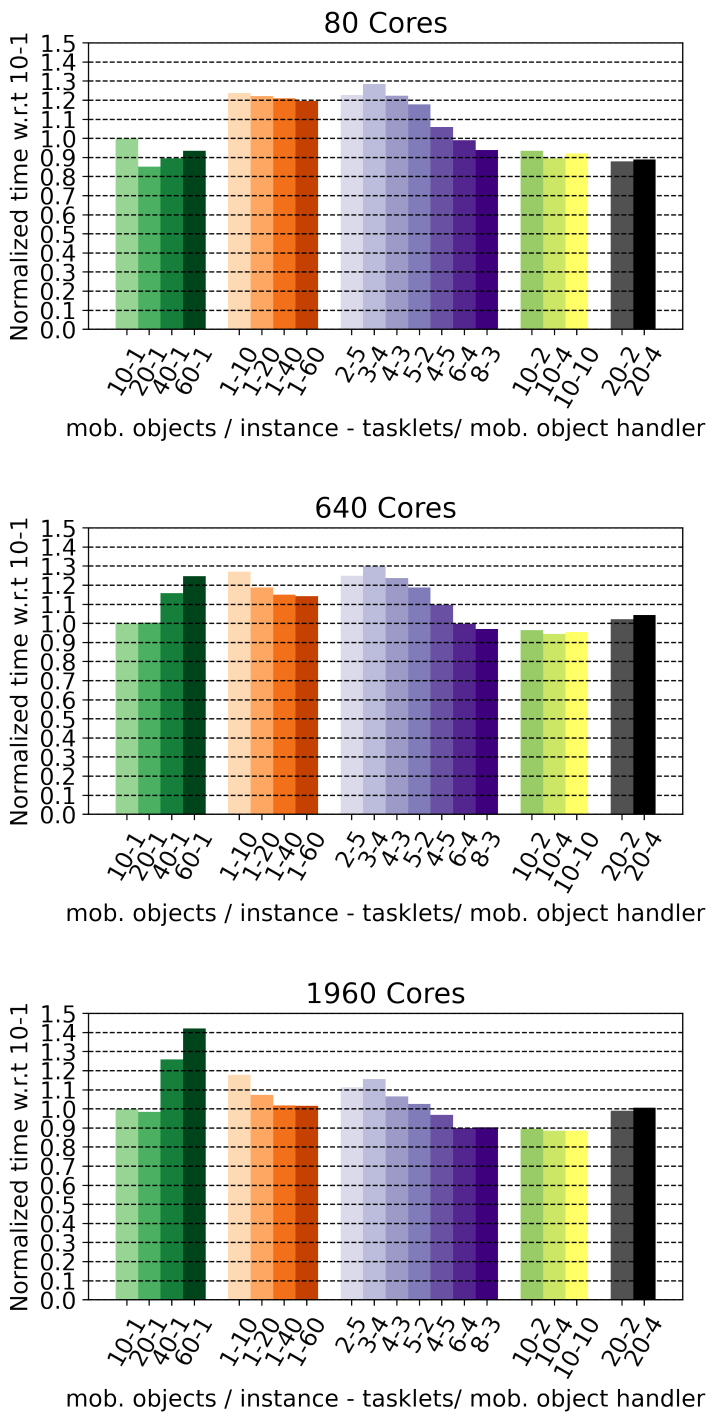


Fig. 27. Performance of the SW4 benchmark for different PE allocations. The results are presented normalized with respect to the performance achieved when mapping one mobile object per PE and one tasklet per handler (10-1).

suffers due to the inherent implementation of the applications. However, when there are more but smaller data domains, the inherent sub-par performance of task parallelism is less critical, and task over-decomposition can help to balance workload and more efficiently utilize the available cores evenly.

Combining the two approaches has better results when the number of mobile objects is close to the number of threads, and task decomposition is performed on top of that (purple bars). In most cases, our best results (12% improvements) were achieved when using a combination of 10 mobile objects -10 tasklets (yellow bars). Using 20 mobile objects per instance (2 per thread) and two to four times data decomposition per object (gray bars) also showed some improvement in lower counts of cores which diminished in larger allocations without, however, hurting performance as other cases did.

These experiments show that a reasonable level of over-decomposition in both levels can substantially benefit the performance of an application by allowing overall increased flexibility for the underlying scheduler/load balancer. The specific values of over-decomposition are application-dependent and might need some experimentation to optimize. When done by hand, adjusting the decomposition can be complex and error-prone, but PREMA completely hides this transition once the standard case has been implemented.

4.5.7 Parallel Mesh Refinement

Our final benchmark is a parallel mesh refinement application, CDT3D [62]. From our previous experience integrating this highly irregular and adaptive application with PREMA, we detected some serious limitations of PREMA that led us to enhance it with lightweight threads and fine-grained tasklets. To run the application on top of PREMA, we decomposed the initial mesh into sub-meshes and assigned them to the available cores, running a sequential refinement handler on each. We over-decompose the mesh so that each core is initially assigned ten sub-meshes to allow more flexibility for inter-handler shared and distributed memory load balancing. The issue is depicted in the first two graphs of Figure 28 that present the execution time per core when running CDT3D on 1000 cores (25 nodes) of the Wahab distributed memory machine. The left graph shows the execution time of each core when only shared memory inter-handler parallelism is used, while distributed load balancing is also utilized in the middle. While most load balancing issues are handled, a few “spikes” in execution time remain, constituting refinement processes that took too long to execute compared to the average, stemming from the adaptive nature of the application. By invoking PREMA tasklets to explore intra-handler parallelism, we can finally achieve correct load balancing that reduces the overall running time by 50% compared to the distributed load

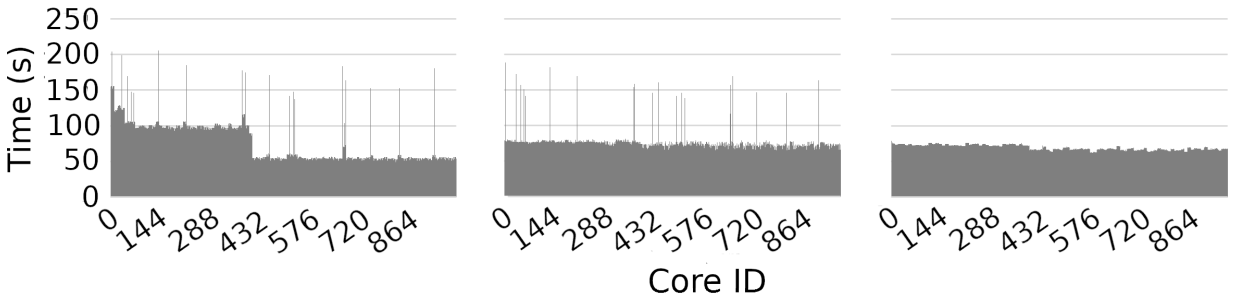


Fig. 28. Per core execution time for the mesh refinement application. Inter-handler shared memory load balancing (left), inter-handler shared & distributed memory load balancing (middle), and intra-handler tasklet parallelism on top of inter-handler shared & distributed memory load balancing (right)

balancing case and 60% compared to the base case. The spikes in execution time are removed by over-decomposing a single handler into multiple tasklets. The tasklets that constitute this handler can be shared across the available cores of a single node, effectively parallelizing the workload that would otherwise map to only a single core. Thus, mitigating the effects of any handler that executes substantially longer than others.

4.5.8 Reducing Code Complexity

In this section, we present how PREMA can remove most of the complexity required to coordinate the different needs of a distributed application. As a case study, we use a two-level parallel mesh refinement code, namely the Parallel Optimistic Delaunay Meshing (PODM) [76], on top of the Parallel Data Refinement(PDR) [77] algorithm.

PODM [76] is a high-performance, parallel mesh refinement software targeting shared and distributed shared memory (DSM). The parallel algorithm uses an optimistic (speculative) execution mode where different cavities of the mesh that do not adhere to the Delauney property are refined concurrently while each refinement process is first checked for data conflicts. Data conflicts are monitored through lightweight locks per vertex of the mesh. Before starting the refinement process for a cavity, the algorithm needs to acquire the locks of its vertices; failure to acquire a lock signifies a data conflict with another cavity refinement process. In the case of data conflicts, the refinement process that caused the conflict is aborted, releasing the locks acquired so far. Then the algorithm schedules the current cavity to be examined later, and a different cavity is picked for refinement.

To extend this algorithm to utilize distributed memory systems, first, an initial coarse mesh is generated on a single computing node using the shared memory algorithm. Next, the coarse mesh is decomposed into subregions; in this process, the mesh bounding box is partitioned into cubes, and each mesh element is assigned to one of these cubes based on the element's circumcenter. Each subregion created in this step is considered a refinement task. A node assigned such a task will then use the shared memory PODM algorithm to refine the cavities inside this subregion. However, the refinement process can potentially propagate to elements assigned to a neighboring subregion (but not further than that, as proved in [77]). As a result, to guarantee the Delaunay property, a subregion task assigned to a node requires all of its neighbors to be present on this node before its execution starts and remain exclusively accessible until it completes. Thus, when scheduling a task to execute on some node, none of its neighboring subregions should be scheduled concurrently, and all its subregions should be at the same node before it is executed. Finally, at the end of each task execution, the scheduler is updated with information about which subregions still need refinement; when no such subregion exists, the distributed refinement algorithm terminates.

The distributed algorithm PDR.PODM is presented in [3], where a Master-Worker model is used. Specifically, the master will generate the initial coarse mesh, decompose it, and then assign its subregions (along with their neighbors) to different workers. When a worker finishes with its task, it sends feedback about work left in neighbor subregions and asks for more work from the master, who checks the pool of pending refinement tasks and assigns one to the worker. However, the task data may be located remotely because of the previous refinement task. In such cases, the master keeps track of the data locations and passes this information to the worker. The worker is then responsible for issuing messages that request these data pieces from other workers. As a result, workers not only need to check their workload and request new tasks to execute, but they also need to be ready to answer data requests from other workers. The above implementation requires handling many technical details which are not part of the algorithm, making the code harder to understand and can lead to errors. We have implemented the algorithm on top of PREMA to showcase how the code can be substantially simplified when using our system without losing performance. The process to implement PDR.PODM on top of PREMA is the following:

- First, each subregion of the mesh is abstracted as a uniquely identifiable (from the whole distributed system) mobile object. Each mobile object is associated with a callback to serialize and deserialize the data it encapsulates for migration purposes. Using this abstraction, PREMA can implicitly migrate subregions to remote computing nodes when needed.

- Next, the dependencies between the task subregions and their neighbors must be defined; we use the PREMA events for this. Each subregion task has its neighboring subregions as prerequisites of the refinement. PREMA is then responsible for guaranteeing that all related data are accessible locally before the task execution starts. Once dependencies have been defined, the application only needs to provide the function that must be executed on the respective mobile object, including the refinement process and an update to the PDR algorithm about pending work on neighbor subregions.

PREMA will monitor the load of each computing node and assign tasks in a way that guarantees load balance. The application is now oblivious to the number of computing nodes, the location of the data, the load of each node, and the data requests between workers. By using PREMA's interface, the same code can scale from a single node to a big cluster while optimizations specific to load balancing, scheduling, and data placement are implemented in an isolated module without needing to modify the application algorithm itself. High-level pseudocodes are presented for both implementations, presenting the workflow of the master and the worker processes. Figures 29 and 31 present the workflow of the master in the two implementations. It is easy to understand the complexity reduction provided by PREMA, which removes most of the message handling and coordination code. The only code the user needs to define is the dependencies between the data subregions and the termination condition. For the worker processes in Figures 30 and 32, the simplification is even more apparent since the worker process only needs to run the refinement process and update the master about it.

In addition to the reduction in code complexity that has been demonstrated, the porting of PDR.PODM onto the PREMA platform has facilitated the identification of a correctness issue within the code. Specifically, in the MPI version, the master node maintains a copy of the initial mesh subdomains and includes these copies in every computational task that is sent to a worker, along with any subdomains that have yet to be scheduled for processing. Regrettably, a bug in the code causes the master to include its own subdomain copies in every task, regardless of whether these subdomains have already been scheduled and updated by another worker. As a result, workers always receive the initial (master) copies of the subdomains rather than the potentially updated versions held by other workers. An illustration of this issue is presented in Figure 33, where it can be seen that when Leaf 1 is scheduled to be refined, it receives an outdated version of Leaf 0 as its data dependence, thus causing the previous refinement step to be nullified.

The PREMA platform has been instrumental in detecting and resolving this issue by offering a unique and globally addressable abstraction for data subdomains. Firstly, data distribution is handled automatically by PREMA, which prevents the bug from arising in the first place. Secondly, it highlights that the code used by a worker to collect subdomain data is flawed, leading to PREMA attempting to move empty buffers and ultimately causing a system crash. The desired outcome is

```
1: procedure MASTERPROCESS
2:     Generate an initial mesh in parallel that conforms to $\delta_1$;
3:     Use a uniform octree to decompose the whole region into subregions;
4:     Assign the elements of the initial mesh to octree leaves;
5:     Push all octree leaves to a task queue Q;
6:     while 1 do
7:         probe for message;
8:         if message is a task request from a worker $P_i$ then
9:             if Q != $\emptyset$ then
10:                Receive message from $P_i$;
11:                Get one subregion $L$ from task queue $Q$;
12:                Send subregion $L$ and its neighbor subregions $Location$ information to $P_i$;
13:                Set process $P_i$ to status $HAS\_WORK$;
14:            else if Q == $\emptyset$ && at least one worker process's status is $HAS\_WORK$ then
15:                Receive message from $P_i$;
16:                Put $P_i$ to waiting task list $WTL$
17:                Send a message to $P_i$ with status $WAIT\_IN\_LIST$
18:            else if Q== $\emptyset$ && all worker processes' statuses are $NO\_WORK$ then
19:                Send termination message to $P_i$
20:                Send termination message to every process that is waiting in the $WTL$;
21:                if number of terminated workers == number of workers then
22:                    break
23:                end if
24:            end if
25:        else if message is a data request from worker $P_i$ then
26:            Receive message from $P_i$
27:            Pack data and send them to $P_i$
28:        else if message is a feedback message from $P_i$ then
29:            Receive message from $P_i$
30:            Set $P_i$ to status $NO\_WORK$
31:            Update task queue Q based on the feedback message
32:            while Q!= $\emptyset$ && task list $WTL$ is not empty do
33:                Get one subregion from Q;
34:                Pop one process $P_j$ from $WTL$ and set its status to $HAS\_WORK$
35:                Send subregion and neighbors $Location$ information to $P_j$
36:            end while
37:        end if
38:    end while
39: end procedure
```

Fig. 29. A high-level description of PDR PODM master process workflow.

```
 1: procedure WORKERPROCESS
 2:     while 1 do
 3:         Send a task request to master process $P_0$
 4:         Probe for messages;
 5:         if message is $Location$ information from master process $P_0$ then
 6:             Receive the message from $P_0$
 7:             Send data (sub mesh) request to each process $P_k$ in $Location$ array
 8:             while 1 do
 9:                 Probe for messages
10:                 if message is a data (sub mesh) request from a process $P_j$ then
11:                     Receive the message from $P_j$
12:                     Send data to $P_j$
13:                 else if message contains data (sub mesh) from $P_k$ then
14:                     Receive message from $P_k$
15:                     numSubmeshesReceived += number of sub meshes from $P_k$
16:                     if numSubmeshesReceived == number of sub meshes need then
17:                         break
18:                     end if
19:                 end if
20:             end while
21:             Stitch sub meshes together and do refinement using worker threads
22:             Send feedback message with mesh refinement information to $P_0$
23:         else if message from $P_0$ with status $WAIT\_IN\_LIST$ then
24:             while $P_i$ is waiting for new tasks do
25:                 Probe for messages
26:                 if message is data (sub mesh) request from a process $P_j$ then
27:                     Receive message from $P_j$
28:                     Send data (sub mesh) to $P_J$
29:                 end if
30:             end while
31:         else if message is a termination message from $P\_0$ then
32:             break
33:         end if
34:     end while
35: end procedure
```

Fig. 30. A high-level description of PDR.PODM worker process workflow.

```
 1: procedure FEEDBACKHANDLER
 2:     Update task queue Q on master $P_0$
 3:     Signal if waiting for feedback
 4: end procedure
 5: procedure MASTERPROCESSPREMA
 6:     Generate an initial mesh in parallel that conforms to $\delta_1$;
 7:     Use a uniform octree to decompose the whole region into subregions based on $g$;
 8:     Find the buffer zones of each subregion of the octree;
 9:     Assign the elements of the initial mesh to octree leaves;
10:     Push all octree leaves to a task queue Q;
11:     Register handler and callback functions
12:     Create mobile objects for each octree leaf
13:     while 1 do
14:         while Q!= $\emptyset$ do
15:             Get one subregion (mobile pointer) from Q;
16:             Create a dependency event for the subregion and its neighbors
17:             Send a $WorkerRefineHandler$ on the mobile pointer
18:         end while
19:         Wait for feedback
20:         if no feedback to wait && Q== $\emptyset$ then
21:             Broadcast a $TerminationHandler$
22:             break
23:         end if
24:     end while
25: end procedure
```

Fig. 31. A high-level description of PDR PODM PREMA master process workflow.

```
1: procedure WorkerRefineHandler(submesh)
2:     Do refinement on sub mesh
3:     Send a $FeedBackHandler$ to $P_0$
4: end procedure

5: procedure TerminationHandler
6:     Signal termination
7: end procedure

8: procedure WorkerProcessPREMA
9:     Register handler and callback functions
10:    Wait for termination message
11: end procedure
```

Fig. 32. A high-level description of PDR PODM PREMA worker process workflow.

demonstrated in Figure 34.

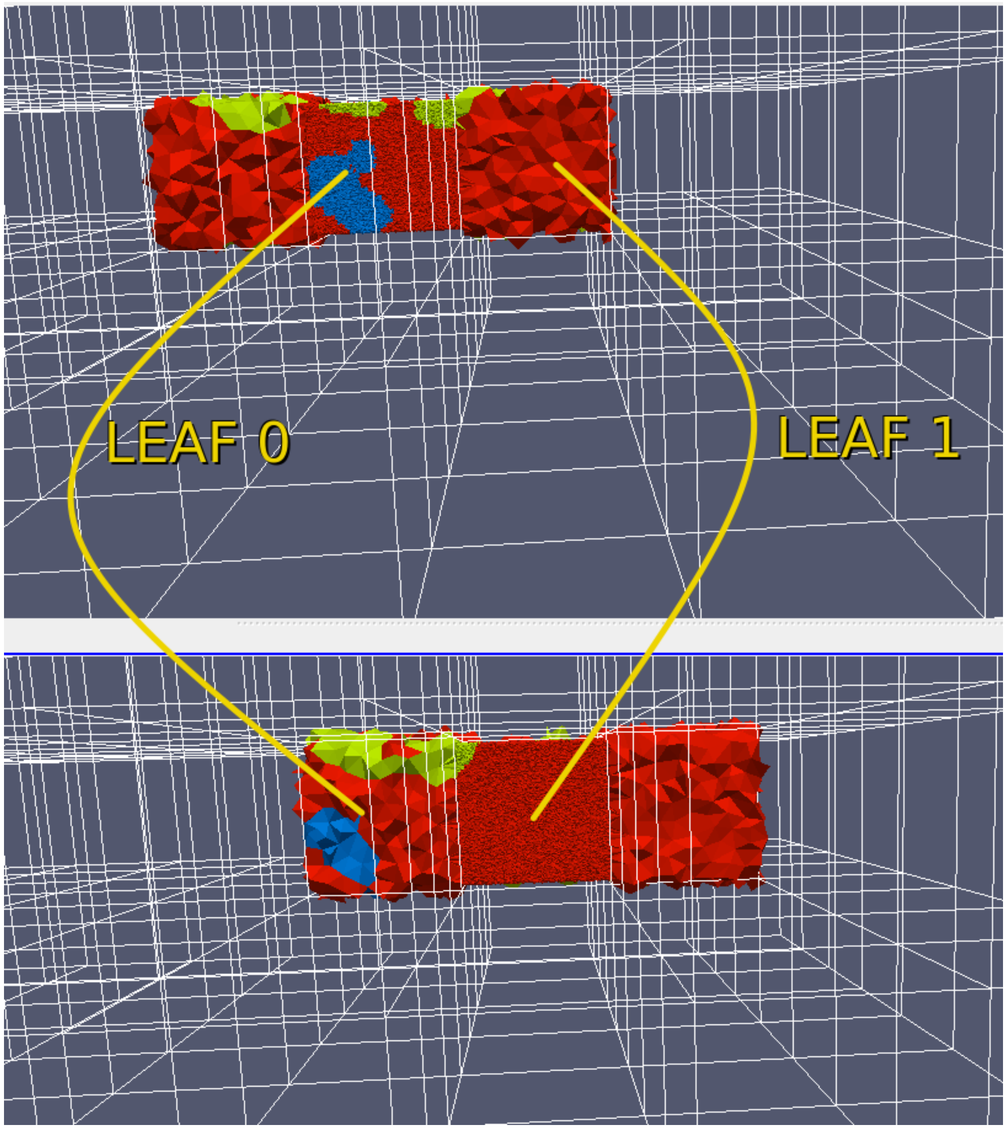


Fig. 33. A demonstration of the bug detected in the original MPI implementation. Even though leaf 0 undergoes refinement as the main subdomain of a task (top), its original, unrefined state is present when leaf 1 is scheduled for its own refinement task (bottom).

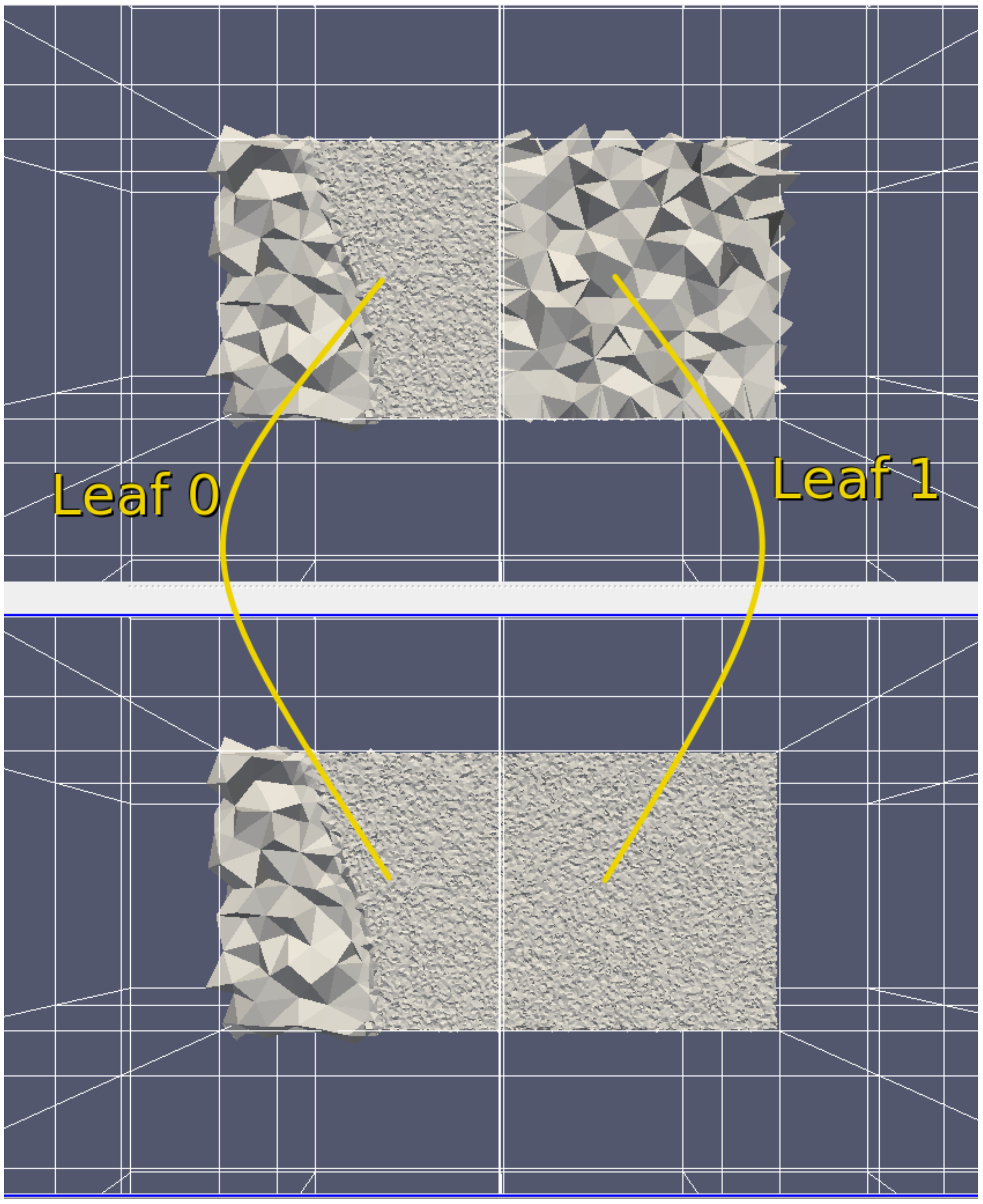


Fig. 34. A demonstration of the fixed version of PDR PODM. After its refinement, leaf 0 is correctly passed to leaf 1 as dependent for its own refinement task.

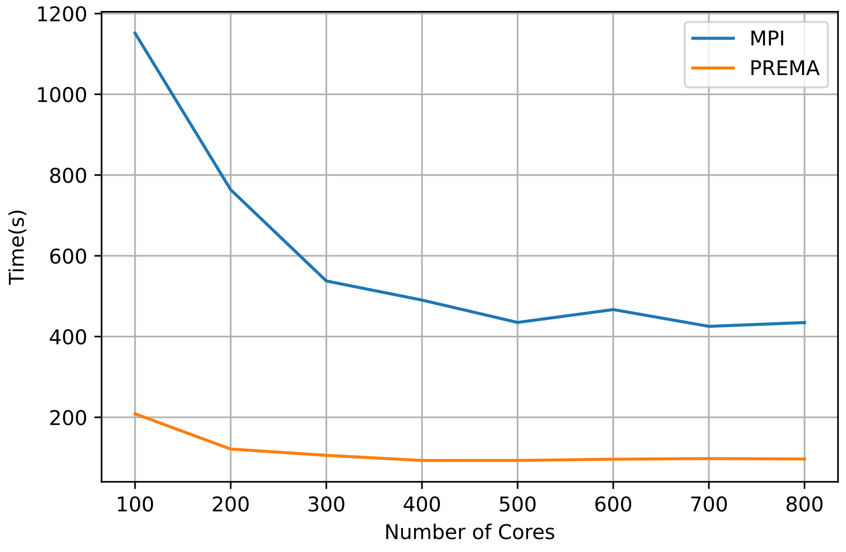


Fig. 35. PDR PODM MPI performance versus PREMA.

Performance Data

In this section, we focus on the code simplification achieved by PREMA; however, the performance gain attained is also very significant, as shown in Fig. 35. This improvement comes primarily from PREMA automatically handling all threads used in the refinement, packing, unpacking, scheduling, and load-balancing processes. Specifically, we noticed that by switching from the application's approach to let the mesher's threads busy waiting while no refining process is running to PREMA's approach of suspending those threads, we get an improvement of up to 80%.

4.6 SUMMARY

We have presented the integration of PREMA with lightweight threads utilizing Argobots. The product of this effort overcomes the limitations previously exhibited by PREMA while incorporating features for effortlessly handling control flow in shared and distributed memory and a tasking framework that allows for fine-grained parallelism inside the execution of remote method invocations. We have experimented with multiple design choices for task creation and message handling, where we observed up to 100% difference in latency. Moreover, we have shown that the lightweight threads remove constraints forced by previous implementations, allowing blocking in remote method invocations while avoiding the overheads stemming from waiting semantics like mutexes. The fine-grained tasking framework achieves performance on par with industrial-strength systems like TBB while outperforming OpenMP. Our experimentation with different combinations of workload decomposition both on the data and task level on the SW4lite benchmark showed up to 12% improvements. Finally, we show that the integration of tasking on top of remote method invocations can have tremendous effects on irregular applications, like 3D mesh refinement, achieving up to 50% improvement through exhibiting multi-grain over-decomposition both on the data and task level. Finally, we have demonstrated the reduction in code complexity achieved by using PREMA versus MPI while not sacrificing performance and even achieving some improvement.

CHAPTER 5

MANAGING HARDWARE HETEROGENEITY

The recent slowdown in Moore's Law is leading to large-scale disruptions in the computing ecosystem. Users and vendors are transitioning from utilizing computing nodes of relatively homogeneous CPU architectures to systems led by multiple GPU devices per node. This trend is expected to continue in the foreseeable future, incorporating many more types of heterogeneous devices, including FPGAs, System-on-Chips (SoCs), and specialized hardware for artificial intelligence[78]. The new computing ecosystem sets the basis to significantly improve performance, energy efficiency, reliability, and security; thus, high-performance computing (HPC) systems are adapted and optimized for traditional and modern workloads.

Exploiting extreme heterogeneity requires new techniques and abstractions that handle the increasing complexity in productivity, portability, and performance. The new methods should allow users to express their applications' workflow uniformly, hiding the peculiarities of the underlying architecture while handling concerns arising from performance portability. One such concern is managing data on various devices. In most cases, data need to be transferred among devices to execute kernels optimized explicitly for an accelerator; thus, a framework needs to allocate the respective memory, find the devices involved in such a transaction, initiate the transfer, and monitor its progress. The coherence of the same data in different devices is also an issue. One needs to guarantee that the application will always use the most recent version of data, no matter which device. Moreover, since these operations are substantial overheads, they should happen asynchronously and overlap with valuable work, further increasing complexity.

Another concern is the orchestration of task (computation) execution. Tasks should only start asynchronous executions after the respective data have been moved to the target device and only when they do not conflict with other tasks, requiring lightweight synchronization. Task scheduling and load balancing should also be a significant concern to keep the available devices saturated and fully utilize them. The schedulers should be aware of each device's load and the data locality of each task to designate where each computation should occur efficiently. Finally, a framework that handles all these concerns should provide a friendly, high-level interface for applications but also expose low-level access that allows experts to optimize their applications for their specific needs. Moreover, having the option of lower-level access is crucial for distributed memory frameworks to use them on multiple nodes efficiently.

Utilizing and orchestrating data movement and task execution on multiple heterogeneous nodes increases the number of issues that need to be tackled. Thus, a complete runtime framework should also facilitate the seamless use of distributed heterogeneous nodes using the same approach of abstractions for data and workload independent of the underlying hardware and have tight integration with the performance portability layer used in a single node. Current trends in HPC follow the programming model of MPI+CUDA, which leads to complicated code, suboptimal performance, or both. Users following this approach must explicitly transfer data between the host and the device before sending/receiving to/from a remote node. Moreover, they will need to use asynchronous operations for both memory transfer operations (host-GPU and network) and overlap them to avoid wasting cycles. This leads to the concern of correctness and synchronization involving the two types of data transfers and asynchronous kernel invocations. And even if one manages to handle all those correctly, they would have an application that only operates efficiently (or at all) for the specific hardware it was developed. Thus, a distributed framework that natively incorporates and abstracts heterogeneous nodes is the only way to create applications that scale independently of the hardware in which they were implemented.

In this work, we extend PREMA to support seamless, efficient, and performance-portable development of distributed applications on heterogeneous nodes. First, we introduce a heterogeneous tasking framework to optimize the parallel execution of heterogeneous tasks on a single node. The tasking framework provides a programming model that automatically leverages heterogeneous devices. In contrast to other systems, our framework does not require the application to choose a device where a task should run; instead, the application only picks a device type, and the framework is responsible for scheduling the task to the optimal computing device. Next, we integrate PREMA with the heterogeneous tasking framework, enabling it to manage and utilize heterogeneous nodes uniformly. Along with the design and implementation of the final product, we present optimizations that contribute to achieving high performance. The evaluation results with microbenchmarks and a proxy application show that our system incurs low overhead with scalable performance.

The major contributions in this chapter are as follows:

- A new design and implementation of a heterogeneous tasking framework for the development of performance portable applications that can scale from single-core to multicore, multi-device (*CPUs, GPUs*) platforms efficiently, *without any code refactoring*.

- A novel integration of a distributed runtime with the heterogeneous tasking framework to provide an end-to-end solution that scales over distributed heterogeneous computing nodes while exposing a high-level and abstract programming model.

- A series of memory, scheduling, and threading performance optimizations that achieve sig-

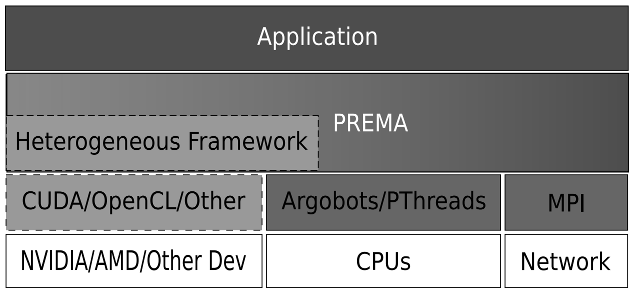


Fig. 36. A high-level representation of the heterogeneity-aware PREMA. The hardware devices/interfaces stand on the lower level and are utilized by integrating PREMA with MPI, PThreads, and Argobots (CPU-only; see [47], [80]), and the heterogeneous tasking framework (in the current work). On top of that stands the application, which leverages these capabilities through a simple but powerful interface.

nificant improvements, up to 300% on a single GPU and linear scalability on a multi-GPU platform, directly applicable to similar systems and applications.

- Demonstration of up to 40% speedup on an end-to-end distributed, heterogeneous proxy application (e.g., Jacobi solver) by utilizing the new runtime framework in combination with widely studied optimizations like over-decomposition[79].

Following the principle of separation of concerns, a new abstract compatibility layer is introduced, allowing PREMA to access different heterogeneous devices uniformly. This layer is implemented as a stand-alone heterogeneous tasking framework that handles all concerns arising from the co-existence of multiple types of devices. PREMA integrates this framework as the preferred way to interact with heterogeneous devices. Thus, it can be easily extended to utilize more device types without needing to modify its implementation, apart from this low-level compatibility layer. Moreover, PREMA exposes some of these capabilities wrapped in a high-level interface, allowing users to utilize such devices in a controlled and safe way. Figure 36 shows a high-level representation of the software stack and how the different layers interact. In the following sections, first, we present the heterogeneous framework layer and its capabilities in detail; then, we focus on its integration with PREMA to provide a distributed, heterogeneity-aware runtime.

5.1 HETEROGENEOUS TASKING FRAMEWORK

The programming model of the heterogeneous tasking framework builds upon two simple abstractions: the heterogeneous objects (*hetero_objects*) and heterogeneous tasks (*hetero_tasks*). A

```
// Device−independent kernel implementation
kernel(dgemm, (device_global double∗ A, device_global double∗ B, device_global double∗ C, long N), {
    parallel_region (
        int ROW = kernel_group_id_y ∗ kernel_local_size_y + kernel_local_id_y ;
        int COL = kernel_group_id_x ∗ kernel_local_size_x + kernel_local_id_x ;
        double local_sum = 0;

        for (int i = 0; i < N; i++) local_sum += A[ROW ∗ N + i] ∗ B[i ∗ N + COL];
        C[ROW ∗ N + COL] = local_sum;
    )
})

int main() {
    const int N = 1024;
    // Allocate NxN matrices A, B, C
    hetero_object<double> m_A(N,N);
    hetero_object<double> m_B(N,N);
    hetero_object<double> m_C(N,N);
    // Get write access to data and populate them
    double ∗A = m_A.request_data(false , true).get();
    double ∗B = m_A.request_data(false , true).get();
    ...

    // Release data access from host
    m_A.release();
    m_B.release();
    hetero_task task;
    // Set the input/output matrices A, B, C
    task.arg(m_A).read();
    task.arg(m_B).read();
    task.arg(m_C).write().dim_x();

    // Set the thread dimensions
    task.set_threads({32, 32, 1}, {32, 32, 1});
    // Set the target device type
    task.device(device::GPU);
    // Set the kernel to execute
    task.submit(dgemm);
}
```

Fig. 37. An example of a DGEMM application using the tasking framework.

hetero_object uniformly represents a user-defined data object residing on one or more computing devices of a heterogeneous compute node (e.g., CPUs, GPUs, FPGAs). Applications treat such objects as opaque containers for data without being aware of their physical location. A *hetero_task* encapsulates a non-preemptive computing kernel that runs to completion and implements a medium-grained parallel computation. Like hetero_objects, hetero_tasks are defined and handled by the application uniformly, independent of the device they will execute on. Fig. 37 shows an example of a DGEMM implementation using the tasking framework. The presented example shows a DGEMM execution request for the GPU; however, by only changing the device target in line 53, one can target a different device without touching the rest of the code. The kernel() macro on the top of the listing will expand to CUDA, OpenCL, and/or other defined backends to fit the vendor-provided implementation.

5.1.1 Heterogeneous Objects

Handling copies of the same data on different heterogeneous devices can lead to error-prone and difficult-to-maintain application code. In general, applications need to manage data transfers among them, use the correct pointer for the respective device, and keep track of their coherence. A hetero_object is an abstraction that automatically handles such concerns, maintaining the different copies of the same data in a single reference. The underlying system controls hetero_objects to guarantee that the most recent version of the data will be available at the target device when needed. For example, accessing an object currently resident on the CPU from a GPU would automatically trigger the underlying data transfer from the host to the respective device. In the same manner, accessing the same object from a different device would initiate a transfer from the GPU to that device, potentially after first staging the data at the host. Finally, the runtime system guarantees data coherence among computing devices, keeping track of up-to-date or stale copies and handling them appropriately.

The memory captured by a hetero_object should be accessed and modified through hetero_tasks for optimal performance. However, the application can also explicitly request access to the underlying data on the host after specifying the type of access requested to maintain coherence. This method will trigger (if needed) an asynchronous transfer from the device with the most recent version of the data and immediately return a future. The future allows for querying the transfer status and providing access to the raw data once the transfer has been completed. In this state, the data of the hetero_object are guaranteed to remain valid on the host side, preventing tasks that would alter them from executing until the user explicitly releases their control back to the runtime system. Since the application has no direct access to the memory allocated to different devices,

```
hetero_task  t0 ,  t1 ;
hetero_object <double> o1,  o2,  o3;
 ...
t0 . add_arg(o2) . read () ;
t0 . add_arg(o1) . write () ;
t0 . device (GPU);
t0 . submit( t0_kernel ) ;
t0 . wait () ;
t1 . add_arg(o1) . read () ;
t1 . add_arg(o3) . read () ;
t1 . device (CPU);
t1 . wait () ;
 ...
```

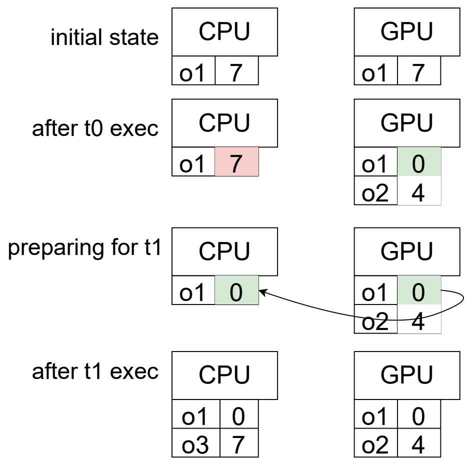


Fig. 38. A simple example where device memory coherency is needed.

our framework monitors the memory usage of each device. When a device's memory is close to being depleted, the runtime system will automatically start offloading some of the user's data to the host or other devices. We currently use a Least Recently Used (LRU) policy to determine which hetero_object should be offloaded to free a device's memory. An application can explicitly request to remove a hetero_object from all devices to help the runtime clean up some memory; otherwise, a hetero_object will be freed when going out of scope. In both cases, the hetero_object will only be removed once no tasks and other operations are referencing it.

Data Coherency

Hetero_objects can maintain multiple copies of their data in different computing units, which allows data reuse and avoids unneeded data transfers and (de)allocations. However, this raises the issue of maintaining coherency between the copies of the same object in different device memories. For example, in Fig. 38, two tasks are submitted to run on two devices (GPU, CPU); however, they both depend on the same hetero_object o1, resident in both the CPU and the GPU. Since the task targeting the GPU is about to write o1, the two tasks must run sequentially. Also, the copy in the CPU memory has to be updated with the value of the GPU copy after the GPU task has run. Therefore, the runtime system implicitly handles this issue for the application.

The MSI cache coherency protocol is implemented to take care of memory coherency. In multicore CPUs, the MSI protocol is used to maintain coherency between the CPU caches; each

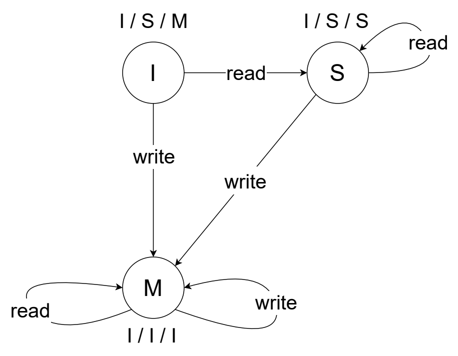


Fig. 39. A Finite State Machine (FSM) of the MSI coherency protocol.

cache line of each CPU is tagged with one of the three states:

- Modified: The block has been modified in the cache. The data in the cache is inconsistent with the memory. When evicted, the block has to be written to memory.
- Shared: This block is unmodified and exists in at least one cache. The cache can evict the data without writing it to memory.
- Invalid: This block is invalid and must be fetched from memory or another cache if the block is to be stored in this cache.

The MSI states are maintained by communicating through the bus and change based on how different CPUs access the respective data blocks. Fig. 39 presents a finite state machine (FSM) with the possible states of a cache (state labels), the transitions between them for each operation (edge labels), and the effect of access to the states of other caches (state annotations). From the FSM, we can see that reading a memory block affects another cache only if the data has been modified in that cache while writing to data will cause all other cached copies to be invalidated. Accessing (reading or writing) invalid blocks will always cause a memory transfer while reading valid blocks does not cause any transfers. For our specific purposes, the MSI protocol is implemented in the context of the hetero_objects. Hetero_objects maintain an array of data states in each device (including the CPU). When submitting a new task, the runtime system will check the access types defined for each input/output hetero_object, trigger any memory transfers required based on the protocol, and modify the states array according to the FSM.

5.1.2 Heterogeneous Tasks

Heterogeneous tasks (hetero_tasks) are opaque structures that consolidate the parameters characterizing a computational task. Through a hetero_task, applications define the kernel to execute, input/output data arguments, processing elements requested (e.g., threads in a CPU), task dependencies, and target device type. Moreover, applications can request the allocation of a temporary shared memory region available only for the duration of the kernel, which maps to the concept of local/shared memory found in other GPU programming APIs (CUDA, OpenCL).

Heterogeneous tasks are independent of the underlying target hardware, allowing a uniform expression of the application workflow whether they target CPUs, GPUs, or other device types. The computational kernel they represent is defined in a dialect similar to an OpenCL kernel that is translated appropriately for each target device. Input/output data arguments of a task are defined as the hetero_objects it needs to access, along with the access type required for each (read, write, read-write). This information is used to issue the appropriate data transfers, maintain coherence, and infer task dependencies. Submitting a task for execution does not immediately execute the respective kernel; instead, the runtime system enqueues the task execution request and immediately returns control to the user. The heterogeneous tasking framework provides methods to query the status of a kernel execution or wait for its completion. Moreover, task dependencies can be defined explicitly by the user or implicitly by the runtime.

Expressing task dependencies is an essential element of task-based programming. Applications can set (explicitly or implicitly) the order inter-dependent tasks should execute, while the runtime system can optimize task scheduling when such dependencies are defined as early as possible. Our runtime provides three ways to express task dependencies: execution flow synchronization in a *fork-join* style, defining *explicit task dependencies*, and *implicit dependencies* through data dependencies. We will use the dependency graph in Fig. 40 to describe the different ways to express dependencies. In this example, tasks T1 and T2 depend on the completion of T0 (i.e., they can only start executing after T0 has been completed), T3 and T4 depend on the completion of T2 but can run in parallel with T1 and each other, while T5 depends on their completion. Finally, T6 depends on T1 and T5, rendering it the last task to run.

Fork-Join Dependencies

Fork-join-style execution dependencies are the simplest and most common way to implement dependencies. For example, an application submits some concurrent tasks and can join them independently to guarantee completion before issuing dependent computations. Global synchro-

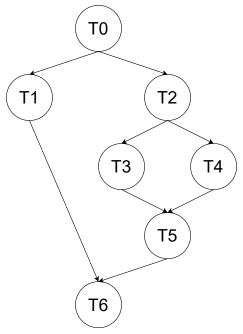


Fig. 40. Task dependencies DAG. The arrows represent a dependency between two tasks.

nization may also be available through a barrier construct. Our runtime system provides such functionality through the wait() method of tasks to join individual tasks and the wait_all() function to issue a barrier-like synchronization.

In Fig. 41, the application explicitly waits for all dependencies of a task to complete before submitting a dependent one (e.g., lines 5-8, 9-12). This approach restricts the discovery of more parallelism and hinders the effort of the scheduling engine to overlap data transfers with computations since it provides only a small subset of the application task graph at a time. Using this way to express dependencies, the application takes full responsibility for managing dependencies. The only support the runtime system provides is to signal when a task is complete. The application has to monitor the execution order by submitting new tasks only when their dependencies have been satisfied.

Explicit Dependencies

Even though fork-join parallelism is easy to implement and widespread among task-based runtime systems, it is not very convenient for complex graphs of dependencies. Moreover, waiting for all dependent tasks to be completed before issuing new ones can impede opportunities for optimizations and deteriorate performance. To overcome the issues arising from fork-join synchronizations, our runtime allows explicitly defining dependencies. Dependencies can be constructed through a task's add_dependency() method before submitting it for execution. This way of defining dependencies allows overlapping task creation, execution, and dependency definitions while providing information to the runtime system as early as possible.

An example of explicitly defining task dependencies is presented in Fig. 42. In contrast to the fork-join dependency representation, an application can define task dependencies a-priory (e.g.,

```
hetero_task t0, t1, t2, t3, t4, t5, t6;
// Define task properties for each task_handle: pe_dims, device, arg, etc ...
...
t0.submit(t0_kernel);
t0.wait(); // T0 completed, can now launch T1, T2
// T1, T2 independent, can run in parallel
t1.submit(t1_kernel);
t2.submit(t2_kernel);
t2.wait(); // T3, T4 depend on T2
// T1, T3, T4 independent, can run in parallel
t3.submit(t3_kernel);
t4.submit(t4_kernel);
t3.wait(); // T5 depends on T3, T4
t4.wait(); // T5 depends on T3, T4
t5.submit(t5_kernel);
wait_all(); // T6 requires the completion of all other tasks before running
t6.submit(t6_kernel);
t6.wait();
...
```

Fig. 41. Forcing task dependencies in the fork-join style.

lines 6-11) and then submit each computing task one after the other without waiting for any of them to complete. Also, dependents (but not extra dependencies) can be added to already submitted or currently running tasks (lines 21-23); the runtime system maintains the task handles as long as at least one reference to them exists. Internally, the runtime system's execution engine can easily infer whether a task's dependencies have been fulfilled since each task contains a list with pointers to its dependencies. By iterating this list and checking each task for completion, it can infer whether all the dependency tasks have been completed and, thus, if the next task is safe to execute.

Implicit Dependencies

Our runtime system supports implicit dependencies discovery to reduce further the effort required to develop applications using tasks. Dependencies are inferred automatically from the runtime system based on the access types defined by the application for the data arguments of each task. For example, tasks accessing different hetero_objects can run concurrently, regardless of access type. Moreover, even when they access common hetero_objects, they can still run concur-

```
hetero_task t0, t1, t2, t3, t4, t5, t6;
// Define task properties for each task_handle: pe_dims, device, arg, etc ...
...

// Define dependencies between tasks
t1.add_dependency(t0); // T1 <— T0
t2.add_dependency(t0); // T2 <— T0
t3.add_dependency(t2); // T3 <— T2
t4.add_dependency(t2); // T4 <— T2
t5.add_dependency(t3); // T5 <— T3
t5.add_dependency(t4); // T5 <— T4

// Tasks with defined dependencies can be submitted without waiting
t0.submit(t0_kernel);
t1.submit(t1_kernel);
t2.submit(t2_kernel);
t3.submit(t3_kernel);
t4.submit(t4_kernel);
t5.submit(t5_kernel);
// We can add dependents on already submitted tasks
t6.add_dependency(t1); // T6 <— T1
t6.add_dependency(t5); // T6 <— T5
t6.submit(t6_kernel);
// If T6 terminates, all other tasks must have terminated too.
t6.wait(); // Could also use wait_all()
...
```

Fig. 42. Building the task graph with explicit task dependencies.

rently if the access type is read-only. However, when accessing a common set of hetero_objects with even one accessed for write purposes, the tasks are dependent and should be serialized. Therefore, the order in which they should run is the order of submission.

As a result of the implicit dependencies, an application can submit tasks in a valid order, and the runtime automatically generates the dependency graph that guarantees parallel correctness. Using this graph, the runtime system can execute independent tasks in parallel (their order is arbitrary) while ensuring that dependent tasks run sequentially and in order. Fig. 43 presents how the code used in the other two methods is simplified when implicit dependencies are used. Indeed, the

```
1   hetero_task t0, t1, t2, t3, t4, t5, t6;
2   // Define task properties for each task_handle: pe_dims, device, arg, etc ...
3   ...
4
5   // Submitting tasks in a sequentially valid order is enough
6   t0.submit(t0_kernel);
7   t1.submit(t1_kernel);
8   t2.submit(t2_kernel);
9   t3.submit(t3_kernel);
10  t4.submit(t4_kernel);
11  t5.submit(t5_kernel);
12  t6.submit(t6_kernel);
13  // If T6 terminates, all other tasks must have terminated too.
14  t6.wait(); // Could also use wait_all()
15  ...
```

Fig. 43. Building the task graph with implicit task dependencies.

application only needs to define the access type of each data argument for each task, which is required either way to achieve data coherency between the computing devices and submit tasks in a sequentially valid order (lines 6-12).

Internally, the implicit dependencies change to explicit ones before passing each task to the execution engine. The task submission function automatically keeps track of tasks submitted and their access types per requested hetero_object to form dependencies. Submitting a writing task creates an implicit synchronization point for all tasks. When a writing task is submitted, all previously submitted reading tasks for the same hetero_object are added as their dependencies. Moreover, reading tasks submitted after a writing task add the writer as their dependency. Thus, the runtime system needs to track the reader tasks between two synchronization points(between two writing tasks) at each time.

All of the above ways to express dependencies can function simultaneously to facilitate the implementation of any algorithm in an expressive and flexible manner. An application can also deactivate the implicit dependencies globally or on a task-by-task basis. Such a feature could be desirable in cases where the application can guarantee data consistency even when two or more tasks simultaneously modify the same hetero_object(s), e.g., using atomic operations to access a variable.

5.1.3 Execution Model

Heterogeneous tasks are executed asynchronously by the tasking framework. A task submitted for execution is appended to a list of task execution requests. The control is then immediately returned to the application, which can continue to issue more tasks or execute other work. A separate runtime component (optionally running in a separate thread) examines task execution requests and eventually schedules them for execution after performing the necessary steps to guarantee correctness.

The first step towards executing a task is to infer its dependencies with other tasks based on their data arguments. The runtime maintains a list of the currently submitted or running tasks that target each hetero_object; new tasks that access these hetero_objects with a conflicting access type have their dependencies set accordingly. Those with at least one incomplete dependence are pushed to another queue of blocked tasks; otherwise, they are appended directly to the scheduler's runnable work pool. Blocked tasks are periodically checked for resolved dependencies, and those with all their dependencies resolved are moved to the scheduler's runnable tasks pool.

Once all blocked tasks have been examined, the scheduler is ready to schedule the runnable tasks. At this stage, the scheduler decides the order in which the tasks should be executed and the device where they should run based on the user's preference (i.e., the scheduler chooses the specific device ID while the user only gives a selection for a device type). The runtime will then reserve device resources and issue the data transfer requests of the input/output hetero_objects to be accessed on the chosen device. Moreover, if possible, it will automatically try to overlap the different operations by utilizing the features provided by the target device's API (e.g., CUDA streams or OpenCL command queues). When all outstanding data transfers of a task have been completed, the computational kernel will be submitted to the target's work pool. Submitted tasks are periodically checked for completion by the runtime to update the status of pending dependent tasks.

5.1.4 Scheduler

With the introduction of more heterogeneous computing devices and workloads, it is expected that scheduling and load balancing will only become more complicated. To provide flexibility for different use cases, the actual implementation of the scheduler is designed to be modular and separate from the rest of the heterogeneous tasking framework. We provide the scheduler as an abstract class that only requires two operations to be implemented. The *push()* operation adds a new runnable task into the scheduler's work pool while the *pop()* operation returns the next task to be executed as well as the device it should run on. The abstract scheduler class allows the development of as simple or complex custom data structures and policies as the user might need.

5.1.5 Abstracting Low-Level Implementations

As mentioned before, this runtime system can target different low-level computing libraries, including CUDA and OpenCL for NVIDIA GPUs, OpenCL for AMD GPUs, and OpenCL and Argobots for CPUs. Theoretically, by targeting the OpenCL backend, one could target other types of devices that implement the OpenCL standard, like SoC and FPGA, but this is beyond the focus of this work. To simplify the development of the runtime system, we have developed a set of platform-agnostic functions and class definitions that need to be implemented for the different operations of the runtime system. Building those on top of the low-level implementations of the different backends exposes their functionality in a uniform, abstract way.

Writing Kernels Using a Platform Agnostic Syntax

CUDA and OpenCL define their kernel codes mostly in C code but enrich the language with extra semantics. Moreover, there is a difference in when and how the kernel codes are compiled. For example, CUDA kernels are compiled when the rest of the code is compiled and require the nvcc compiler. On the other hand, OpenCL kernels are strings passed to OpenCL and compiled at runtime utilizing Just In Time (JIT) compilation.

The difference in how the two backends define their kernels (static code versus string) is why kernels in our runtime system are defined like in Fig. 37. The kernel keyword is a macro that generates two versions of the given kernel, one for each backend. Fig. 44 shows the macro definition of the keyword (top) and the code generated for the two underlying backends (bottom). In the dgemm example, when the submit() method is called for the dgemm kernel, the respective struct is passed to the runtime system, which will use the appropriate field when issuing the task to a device.

The different semantics added to the C language by each backend are handled using macros. Table 4 shows the available semantics and how they map to backend-specific features. Redefining the uniform semantics macros to backend-specific implementations is possible because the OpenCL version uses a separate compiling step from the rest of the application. By passing a different implementation for the macro at the OpenCL compilation step, we get two definitions for the same macro.

Another feature is the dynamic local/shared memory, which both OpenCL and CUDA support, to request quick-access, temporary memory only shared in a thread group/block. The two computing libraries use different approaches to implement this feature. OpenCL provides access to shared

```
#define kernel(name, args, definition ) __global__ void name_##cuda_fn args definition \
 struct { \
      void* cuda_fn; \
      const char*; \
} name = {.cuda_fn = (void*)name_##cuda_fn, \
                .ocl_fn = "__kernel void "#name"_fn"#args# definition };
```

```
// CUDA kernel function
__global__
void dgemm_cuda_fn(device_global double* A, device_global double* B, device_global double* C, int N)
{ ... }

// OpenCL kernel string
"__kernel
void dgemm_cuda_fn(device_global double* A, device_global double* B, device_global double* C, int N)
{ ... }"
```

Fig. 44. The kernel macro (top) and the generated code (bottom) for the kernel in Figure 37.

memory through "special" arguments (tagged as local_mem) to the kernel function. On the other hand, CUDA implements this functionality by defining a unique per kernel extern __shared__ array. To provide a uniform semantic for this feature, the application is restricted to requesting only a single piece of such memory for each kernel (like CUDA). For OpenCL, the runtime system macro will pass this memory as an extra, hidden parameter to the OpenCL version of a kernel definition (the CUDA version remains untouched). To provide access to this hidden parameter, a new semantic is defined, kernel_dynamic_local(mem), which sets the address of the hidden parameter to a variable mem. For CUDA, this semantic implements the definition of an extern shared array. In both cases, the application can manipulate this memory by pointer arithmetic and typecasting, similar to plain CUDA.

Choosing to implement computation kernels in this way prevents the user from using some of the features supported by CUDA for kernels, such as using C++ templates for the kernel code. However, since OpenCL does not support this feature, it is a reasonable compromise to support a common interface.

TABLE 4

Mapping of uniform kernel semantics to backend-specific features.

Uniform Semantics	CUDA	OpenCL
kernel_local_id_{x/y/z}	threadIdx.{x/y/z}	get_local_id({0/1/2})
kernel_local_size_{x/y/z}	blockDim.{x/y/z}	get_local_size({0/1/2})
kernel_group_id_{x/y/z}	blockIdx.{x/y/z}	get_group_id({0/1/2})
kernel_num_groups_{x/y/z}	gridDim.{x/y/z}	get_num_groups({0/1/2})
kernel_barrier()	__syncthreads	barrier(...)
kernel_global	-	__global
kernel_static_local	__shared__	__local

Exposing Platform-Specific Operations

While a uniform kernel syntax is required to efficiently develop the application code, a set of required, common operations also have to be exposed for the internal implementation of the run-time system. Such operations consist of issuing data transfer requests between host and device or device to device, kernel execution requests, memory (de)allocations, and querying for completion. Other operations include initializing and finalizing the computing library and querying the hardware available.

We decided to introduce an API very close to the one provided by OpenCL as it is more generic and can easily express most of CUDA's operations. The datatypes provided are the same for both backends and are implemented as classes that wrap around backend-specific datatypes. The wrapper classes enclose backend-specific datatypes in unions to save space since the same piece of data will never be used by more than one backend at a time. Conversely, the functions provided have identical signatures, but the backend they implement is appended to their name. Thus, when higher-level abstractions are implemented, the same functionality can be used for different device types by choosing the version with the desirable backend in its name; these operations will then use the correct union fields depending on the backend they implement. An example of this design is presented in Fig. 45 for the kernel execution operation in CUDA and OpenCL. The two versions have the same arguments but have slightly different names since the backend they implement is appended.

This design allows us to keep the higher-level classes' member variables platform-agnostic and specify a platform-specific implementation of an operation as late as possible. Moreover, keeping

```
inline void device_kernel_launch_cuda(device_id id, kernel_id kernel, device_thread_dimensions_t
    dims, void** args, size_t* args_sizes, size_t args_size, size_t smem_size, device_queue_t
    queue, device_event_t event)
{
    checkError(cudaSetDevice(id.cu));
    checkError(cudaLaunchKernel(kernel.cu, dims.g, dims.b, args, args_size, queue.cu));
    checkError(cudaEventRecord(event.cu, queue.cu));
}

inline void device_kernel_launch_ocl(device_id id, kernel_id kernel, device_thread_dimensions_t
    dims, void** args, size_t* args_sizes, size_t args_size, size_t smem_size, device_queue_t
    queue, device_event_t event)
{
     size_t i = 0;
    for(i = 0; i < args_size; i++)
    {
      checkError(clSetKernelArg(kernel.ocl, i, args_sizes[i], args[i]));
    }

    checkError(clSetKernelArg(kernel.ocl, i, smem_size, NULL));
    checkError(clEnqueueNDRangeKernel(queue.ocl, kernel.ocl, 3, NULL, dims.g, dims.b, 0, NULL,
        &event.ocl));
    checkError(clFlush(queue.ocl));

}
```

Fig. 45. The low-level implementation of the kernel execution request for CUDA and OpenCL.

```
1 #define kernel_args (...) (__VA_ARGS__, index_info* __ids, char* __smem) // Appends one argument to
      a list of arguments
2 #define kernel(name, args, definition ) void name_##abt_fn kernel_args args definition
3
4 // The function generated from the above macro
5 void dgemm_abt_fn(device_global double* A, device_global double* B, device_global double* C, int N,
      index_info* __ids /* hidden */, char* __smem /* hidden */)
6 {
7     for(ids->local[0].id = 0; ids->local[0].id < ids->local[0].size; ids->local[0].id ++)
8     {...}
9 }
```

Fig. 46. A computing kernel implementing DGEMM for Argobots.

the same signatures for all common operations makes it easy to unify the design on higher-level functions and methods and hide implementation-specific data in the wrapper datatypes.

Argobots as a Backend for CPUs

Since we intend to integrate this heterogeneity-driven runtime system with PREMA, we decided to support the Argobots-based fine-grained tasking framework as a backend for CPUs to provide better and more efficient interoperability between the two systems. This section describes the implementation of the Argobots-powered fine-grained parallelism backend.

Kernel Generation

The work distribution of a kernel between different threads is done automatically for both CUDA and OpenCL, based on the directions provided by the application; however, this is not the case for the Argobots backend, which requires explicit work distribution. Moreover, the two computing libraries provide interfaces that let the application retrieve the ID of the running thread, which also needs to be implemented explicitly for the Argobots backend. To provide the capabilities mentioned above, we need to generate a different kernel for the Argobots backend and modify some of the macros defined previously. Starting with the macros that retrieve the ID of a thread, we need to explicitly pass information about the local ID and size, the number of groups, and the group ID and size. We pass such information along a pointer to potentially requested shared memory as a hidden input parameter to the kernel, as presented in Fig. 46 line 5.

The distribution of kernel local IDs is performed through a new semantic, the parallel_region semantic (Fig. 37, line 4); applications that choose to target the Argobots CPU backend need to insert code that is to be executed in parallel into this semantic, which generates a series of nested loops around the code provided. Currently, the range provided in the local dimensions is executed by a single thread. Instruction Level Parallelism (ILP) is the only parallelism provided through compiler optimizations; however, groups run on multiple threads as expected. A simplified illustration of the macro and the code generated for the DGEMM kernel of Fig. 37 is presented in Fig. 46. Lines 2-3 show how the kernel macro is defined with the added index_info* hidden argument and the loop surrounding the kernel code inserted in the parallel_region semantic; for the sake of space, we only show the loop of a single dimension. Lines 5-9 show the code generated when applying the macro on the DGEMM kernel.

Having access to the information about the threads in the Argobots kernel, the *kernel_thread_id*, *kernel_thread_size*, etc., macros need to be modified to expose this information to the application. Since the macros are used by both the CUDA and the Argobots backend (the OpenCL backend can redefine them because it is compiled separately, as already mentioned), we need to have a way to infer if they are called from a CUDA or Argobots kernel. CUDA provides a way to define functions that can run by both the host and the kernel through the __device and __host attributes. Functions with these attributes are compiled separately for each target. One can distinguish between the two at compile time through a macro defined appropriately based on the target. We use this capability to implement the kernel-specific semantics for CUDA and the Argobots backend by using a unique set of functions that internally retrieve the needed information differently based on the compilation target. For CUDA, they return the respective built-in variable (e.g., threadIdx.x), while for Argobots, they use the hidden index_info structure.

The kernel_barrier() semantic is not directly supported in the Argobots backend; however, the parallel_region semantic denotes the synchronization points in the kernel. Since the loop iterations finish just after the last instruction in the enclosed code, any code after the parallel_region will run once all local ids have run the enclosed block. For CUDA and OpenCL, the parallel_region semantic translates to a kernel_barrier() at the end of the block for this purpose. Dynamic local memory is implemented in the same way as OpenCL, a hidden pointer to the allocated memory is passed through the kernel parameters (Fig. 46, line 5), and the respective call kernel_dynamic_local(mem) provides access to this dynamic local memory.

TABLE 5
Mapping of uniform kernel semantic to backend-specific features for Argobots.

Uniform Semantics	Argobots
kernel_local_id_{x/y/z}	ids→local[0/1/2].id
kernel_local_size_{x/y/z}	ids→local[0/1/2].size
kernel_group_id_{x/y/z}	ids→group[0/1/2].id
kernel_num_groups_{x/y/z}	ids→group[0/1/2].size
kernel_global	-
kernel_static_local	static

Other Operations

Since Argobots only implements operations for CPUs, memory transfer requests use the implementations of the other two backends depending on the target. Memory (de)allocations can use C calls to malloc and free, and kernel submissions the respective API provided by the library. The only thing missing is handling the group and local dimensions of threads, as requested by the task submission API and the dynamic local memory (kernel_local_dynamic). The distribution of the kernel work to different cores is implemented by submitting one Argobots task for each index in the range of the group dimensions. As explained previously, the range in the local dimension is implemented through a loop for each task. For example, submitting the task in Fig. 37 with the group and local dimensions 32x32 would submit 32x32 Argobots tasks where each one would run the kernel in a loop of size 32x32.

Argobots tasks are submitted recursively so that all available hardware threads can theoretically push the same amount of tasks to their work queues. Before a task is submitted, its index_info structure is populated with the IDs and sizes of local and group dimensions. Dynamic shared memory is also allocated at this step by simply using a call to malloc() and setting the respective field of index_info; the memory is deallocated when a task completes its execution. Fig. 47 presents how kernel tasks are submitted to the Argobots backend. Empty nodes represent tasks created to parallelize the initialization process required by the kernel executions; filled nodes represent tasks that execute the kernel enclosing it in a loop.

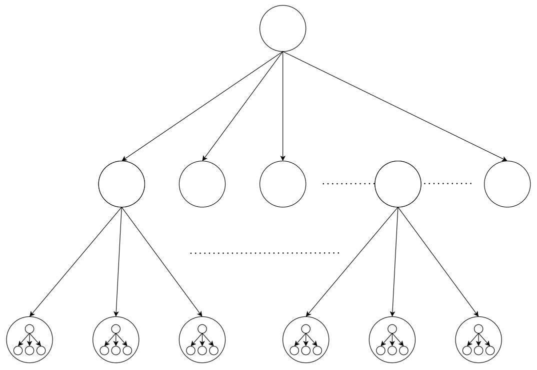


Fig. 47. The Argobots backend task submission paradigm.

5.1.6 Design Abstractions

The heterogeneous tasking framework is implemented in C++, leveraging its performance and object-oriented design. It is developed in three software layers to allow easy integration with new device types, programming APIs, and scheduling policies (see Fig. 48).

The **Device API** is the bottom layer, encapsulating the different operations provided by a heterogeneous device vendor. It consists of abstract C++ classes that expose virtual methods for operations required to (a)synchronously issue tasks and manage data in such devices, query their hardware specifications, and methods to retrieve the status of an asynchronous operation. Currently, we provide native support with CUDA and OpenCL for GPUs. The Device API provides the low-level, vendor-specific implementations of all abstractions of the tasking framework like mapping hetero_objects to memory locations, hetero_tasks, and task execution requests to actual kernel invocations, memory transfer requests, etc.

The next layer is the **Core Runtime** layer, which provides the underlying implementation of the hetero_objects and hetero_tasks, monitors the coherence of the different copies of the data and detects and enforces task dependencies. It utilizes the Device API to coordinate data transfers, guide the correct execution of tasks and signal the completion of different operations. This layer acts as the "glue" between the application preferences, the scheduler and load balancing policies, and the Device API.

At the top stands the **Application Layer**, which consists of a thin API that exposes the capabilities of the tasking framework in a high-level interface. In the current implementation, kernels are defined in a dialect similar to OpenCL through the use of macros which are expanded to implement a kernel version for each available target.

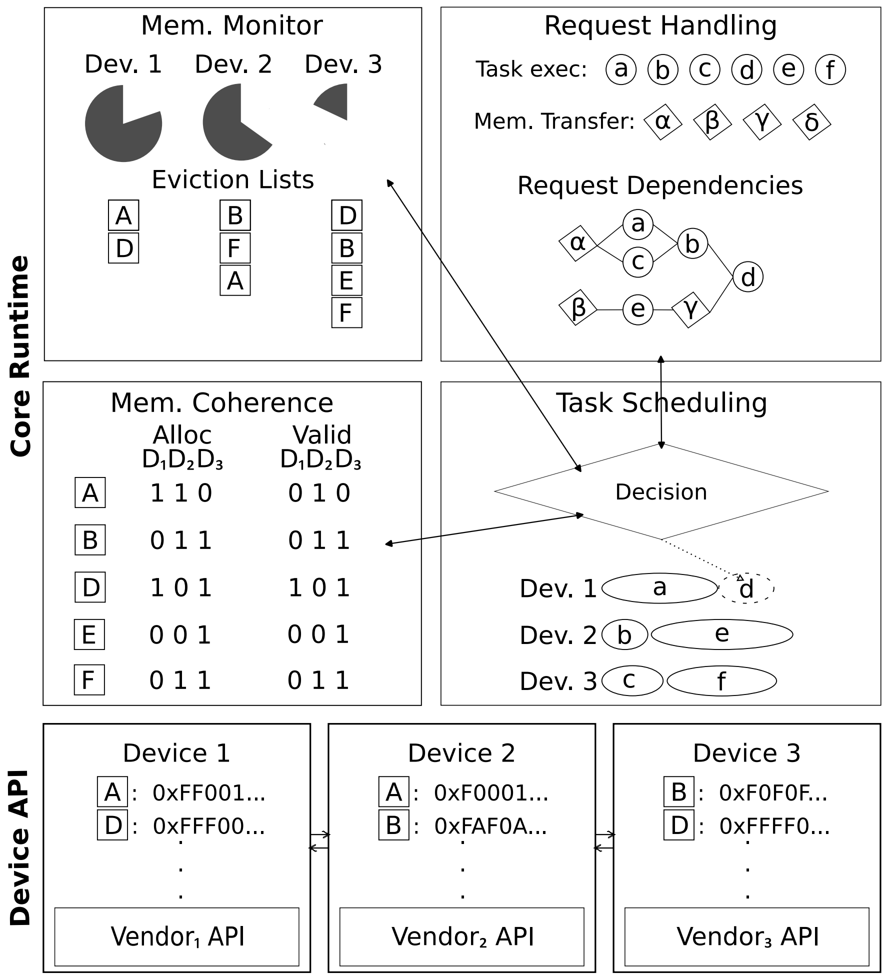


Fig. 48. A representation of the heterogeneous tasking framework software stack and operations. The operations performed by the Core Runtime include (a) Memory monitoring to keep track of the available device memory and deallocate unused objects when running low on resources, (b) Memory transfer and task execution request handling that dispatches such requests when it is safe, (c) Memory coherence among different copies of the same hetero_object in multiple devices, (d) Task scheduling to optimize for a reduction in memory transfers and optimize overall execution time. The Device API exposes an abstract “device class” that encapsulates the implementation of different vendor interfaces uniformly. The device API maps high-level abstractions like hetero_objects and hetero_tasks to actual device-specific constructs.

5.2 HETEROGENEITY WITHIN PREMA

Integrating heterogeneity in PREMA is a crucial requirement to handle the load of exascale-era machines. Applications should be able to use and transfer device memory in the context of remote handler executions without much hassle. A step towards this direction is to allow PREMA to send and receive buffers located in a GPU device either explicitly (currently CUDA only) or through the abstractions of the heterogeneous tasking framework we have introduced. The explicit approach allows users to utilize GPUs without confining them to use our heterogeneous tasking framework, facilitating interoperability with legacy CUDA codes. It also provides a barebone approach to integrate heterogeneity on top of the distributed system, which can act as the base case for our performance evaluation since it introduces the least possible overhead.

5.2.1 Explicitly Handling Devices

In the explicit approach, the application can directly call the different GPU operations of the CUDA API to allocate/free memory, initiate transfers and execute kernels. PREMA provides a function to invoke remote handlers that include a GPU buffer as an argument; the function requests the ID of the remote process, the buffer to transfer, its size, the IDs of the source and target devices, and the handler (host function) to be invoked at the receiver. PREMA will transfer the buffer between the remote GPUs and invoke the handler when it has been completed. The handler can then invoke any GPU-related operation that targets this buffer safely. However, the application needs to guarantee that the handler does not return before the completion of the kernel since any buffers transferred through a handler will be freed at its return, including the GPU buffer. Waiting for all the device operations to complete (e.g., through *cudaDeviceSynchronize()*) is enough to guarantee correctness; however, this approach will harm PREMA's time-slicing abilities, preventing it from switching to other tasks while GPU operations are in progress. Thus, the user should follow a more complicated approach, querying the status of the operations without blocking (e.g., through *cudaEvents*) and periodically yielding control of the thread for PREMA to run background jobs.

5.2.2 Utilizing the Heterogeneous Tasking Framework

To facilitate a higher-level interaction of PREMA with heterogeneous devices, we introduced a set of extensions allowing direct utilization of the abstractions provided by the heterogeneous tasking framework. Compared to the explicit remote handler invocation API, the user only needs to provide the handler to be executed, the target process ID, and the hetero_object passed as an

```
DEFINE_MP_HANDLER(execute_second_dgemm_handler) { // PREMA mobile object handler
    hetero_object <double> m_B = get_hetero_object<double>();
    hetero_task  task;
    // Set the input/output matrices A, B, C
    task.arg(this->m_A).read();
    task.arg(m_B).read();
    task.arg(this->m_C).write().dim_x();
    task.set_threads({32, 32, 1}, {32, 32, 1});
    task.device(device::GPU);
    task.submit(dgemm);
}
DEFINE_MP_HANDLER(execute_first_dgemm_handler) { // PREMA mobile object handler
    hetero_task  task;
    hetero_object  m_C;
    task.arg(this->m_A).read();
    task.arg(this->m_B).read();
    task.arg(m_C).write().dim_x();
    task.set_threads({32, 32, 1}, {32, 32, 1});
    task.device(device::GPU);
    task.submit(dgemm);

    prema::mp_send(this->other_mp, execute_second_dgemm_handler, m_C);
}

int main() {
    const int N = 1024;
    // Allocate NxN matrices A, B, C
    hetero_object <double> m_A(N,N);
    hetero_object <double> m_B(N,N);
    hetero_object <double> m_C(N,N);
    // Populate A, B ...
    mobile_object_data  my_data(m_A, m_B, m_C);
    prema::mobile_ptr  my_mp(my_data); //reference to the remote object
    prema::mobile_ptr  other_mp = /*get remote mobile_ptr*/;
    my_data.other_mp = other_mp;

    prema::mp_send(my_mp, execute_first_dgemm_handler);
}
```

Fig. 49. An example of a series of two DGEMM invocations. The results of the first execution are passed on a remote node to invoke the second, using the heterogeneity aware PREMA.

argument (transferred). Since the hetero_objects handle the location of the underlying data, the user does not need to specify their location. The framework automatically decides the device to store the received buffer on the target process. Once a hetero_object of a remote method invocation has been transferred, the designated handler is invoked on the target. The application can invoke tasks that utilize it on any available device type. Moreover, the application is guaranteed that any hetero_object that is the target of any hetero_task execution or messaging operation will live long enough for all such operations to complete, even if the handler returns earlier. In addition, the tasking framework will make sure that no other task can start executing on a hetero_object that is in the process of network transfer. A code example is shown in Fig. 49 where a series of two distributed DGEMM invocations is implemented in heterogeneous PREMA. Two mobile objects create matrices A, B, and C, then create mobile pointers out of their data and exchange them with each other. Next, each mobile object invokes the first DGEMM on itself and sends the result to the remote object invoking the second DGEMM. Note that there is no need to explicitly handle the data buffers for the network transfer (lines 29, 55). Also, the application does not need to explicitly wait for the completion of the DGEMM task (line 27) before sending the results to the remote mobile object (line 29). Finally, even though the result of the first DGEMM is stored in a local variable (lines 20, 24), and the handler can return before the asynchronous send has completed, the data will be transferred correctly. PREMA and the tasking framework will make sure that data are consistent and updated in the correct order.

Another desirable requirement provided through the hetero_objects on top of PREMA is the ability to "put" and "get" data between potentially distributed devices. A new extension allows users to create global pointers for hetero_objects, i.e., unique identifiers referenceable from all processes in the distributed system. When an application needs to store/retrieve data to/from a remote hetero_object, it just needs to provide its global pointer and the location (hetero_object or pointer) of the data to be read/written, along with a callback that is triggered on the target, signaling the completion of the operation.

5.2.3 Implementation

Depending on the capabilities of the underlying communication library and the device(s) hardware, the actual implementation of the memory transfers differs to leverage heterogeneity-aware communication substrates. When the application utilizes heterogeneous objects, and the communication substrate is not heterogeneity-aware, the implementation of the memory transfers includes the following steps (also see Fig. 50):

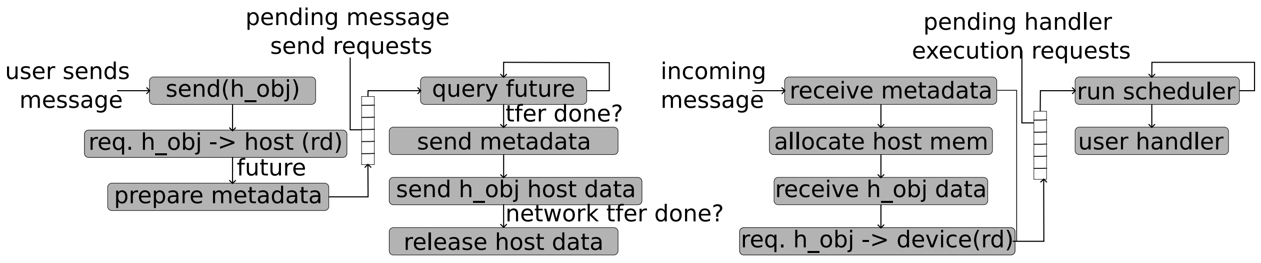


Fig. 50. Heterogeneous data send and receive in PREMA with no GPU-aware interconnect.

Sender:

1. Asynchronous memory read from a device to the host returning a future.
2. Push a message, including the future and handler metadata, to the outgoing message pending queue.
3. Periodically query the outgoing message queue.
4. When the future of a message is complete, send two messages: the metadata and the hetero_object.
5. Once the message transmission is complete, release access to the host data.

Receiver:

1. Receive metadata.
2. Use the information in metadata to prepare for the hetero_object data.
3. Receive the hetero_object in the data prepared.
4. Request allocation of the hetero_object in a device.
5. Execute the user handler.

PREMA will automatically request an asynchronous read of the hetero_object data from the device to the host. The tasking framework will guarantee that the device-to-host transfer will start once all previously submitted, conflicting tasks have finished and prevent any new ones from running before PREMA has finished its network transfer. Next, a header message encapsulating the handler's metadata (e.g., data pointer, target, etc.) is prepared and pushed to a queue of pending

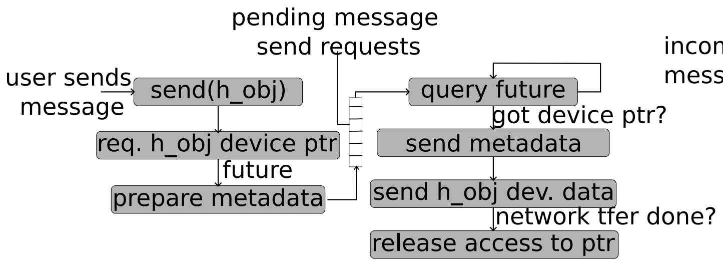
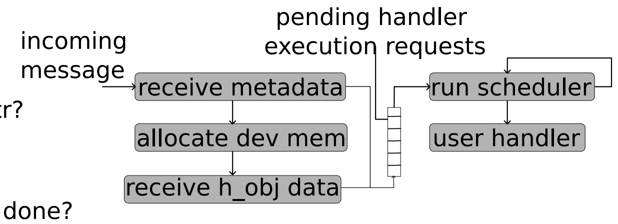


Fig. 51. Heterogeneous data send and receive in PREMA with GPU-aware interconnect.

send message requests. The header will also incorporate a future returned by the previous step that allows checking for the transfer's progress. The outgoing message requests queue is checked periodically by PREMA. When the future of a message signifies the completion of a device-to-host transfer, the message is ready to be sent through the network. PREMA will asynchronously send two messages, one for the metadata and one for the hetero_object data, utilizing the MPI as the communication substrate. Finally, when the two messages are transmitted, PREMA will release its read request from the hetero_object, notifying the tasking framework that the object can be safely modified or deleted by another task.

On the receiving side, once the first metadata message is detected and received, the information it carries is used to allocate the required memory to store the actual data of the hetero_object. Next, the second message with the actual hetero_object data is received in the newly allocated buffer. A request to allocate the received data in a device is sent to the tasking framework, and the metadata message along with this request is enqueued into a pending handler execution request queue. Finally, the scheduler will pick one of the pending handler execution requests and run the respective user handler. In this case, PREMA does not wait for the host-to-device transfer to complete before starting the handler, allowing code independent of the hetero_object itself to run, overlapping the transfer. Furthermore, since the data of the hetero_object will be used through a hetero_task, the tasking framework will ensure that any task execution will be delayed until the transfer has been completed.

The "put" operation is almost identical, with the difference that the receiver does not need to allocate new host memory. Since the hetero_object already exists on the target, PREMA will request write access on the host side, and once access is granted, it will receive the data directly in the hetero_object's host memory. Note that no transfer from device to host is performed since the data will be overwritten from the receive. The request will just guarantee that the network transfer will not conflict with any other task running on this hetero_object.

The host-staging step can be skipped if the communication library/hardware and compute devices support direct transfers between distributed devices (currently only tested for CUDA-OpenMPI). Thus, in an implementation for the case where the user uses hetero_objects, the process is the following (also see Fig. 51):

Sender:

1. Asynchronous request access to device memory returning a future.
2. Push a message, including the future and handler metadata, to the outgoing message pending queue.
3. Periodically query the outgoing message queue.
4. When the future of a message is complete, send two messages: the metadata from the host and the hetero_object directly from the device.
5. Once the message transmission is complete, release access to the device data.

Receiver:

1. Receive metadata.
2. Use the information in metadata to prepare for the hetero_object data in the device.
3. Receive the hetero_object in the data prepared.
4. Request allocation of the hetero_object in a device.
5. Execute the user handler.

PREMA will automatically asynchronously request read access to the hetero_objects pointer in the current device. Again, the tasking framework will ensure that no conflicts with active tasks will be possible, but no transfer will occur. Same as the non-GPU-aware case, a metadata message is prepared that includes a future generated from the previous step's request and is appended to the message send requests. Once it is safe for PREMA to access the device memory, the message is ready to be sent through the network. The metadata message is sent first, followed by the data in the device. Under specific conditions, some MPI implementations can directly target CUDA device memory which is used in this case. When the message transmissions have been completed, PREMA will release its read access to the hetero_object. This will allow other task/messaging operations to modify the hetero_object safely.

On the receiving side, once the metadata message is received, the tasking framework is requested to create a dummy hetero_object on the device that can accommodate the incoming data. Then the metadata message along with the dummy hetero_object are enqueued to the pending handlers queue for execution. When the hetero_object allocation is complete, PREMA will receive the actual data directly in the memory allocated in the device and notify the tasking framework that it no longer needs to keep access to the device pointer. Finally, the scheduler will pick one of the pending handler execution requests and run it. Again, the handler can start before the data have been received in the hetero_object, leaving the tasking framework to guarantee no conflicts will occur by delaying tasks as needed.

Like in the previous case, the "put" operation is the same, with the only difference being that the receiver can directly receive incoming data on the current location of the hetero_object. Since the hetero_object already exists on the target, PREMA will request write access on its current device location. Once access is granted, it will receive the data directly in the hetero_object's device memory. Again, data will be overwritten from the receive; thus, the request will guarantee that the network transfer will not conflict with any other task running on this hetero_object.

5.2.4 Hybrid Programming Model

The integration of the two systems is provided to the application through a new, hybrid programming model where features of both systems are put into use. Specifically, the abstractions of mobile_objects and hetero_objects are used together by reintroducing mobile_objects as amalgamations of hetero_objects. In this new model, mobile objects are used to handle coarse-grained data transfers in response to distributed load balancing, while hetero_objects are used to handle in-node memory transfers between hosts and accelerators for resource utilization. This separation is maintained in order to collect high-overhead functions in a single, high-impact operation instead of issuing multiple low-impact operations with high overhead. We made this decision because all of the aforementioned functions incur high latencies to start the process while operating faster when bigger data blocks are processed at a time.

Mobile Objects as Amalgamations of Hetero Objects

To their greatest extent, user data are now enclosed into hetero objects in order to benefit from the implicit support for data allocations, transfers, and coherency between different devices. To use the advantages of the mobile objects, hetero objects can be defined as part of a mobile object in the context of PREMA. Thus, each hetero object is "owned" by a mobile object, and the location of a hetero object is determined by the location of its parent mobile object. Fig. 52 shows an example

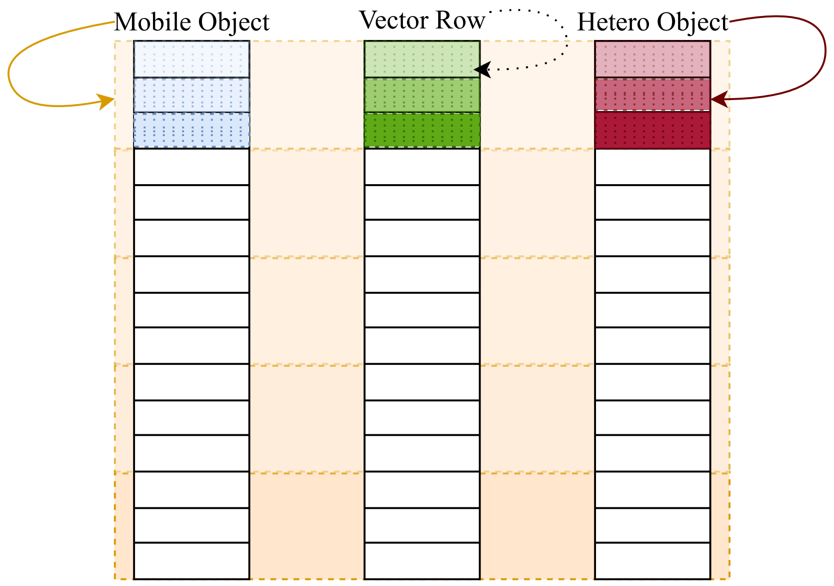


Fig. 52. An example decomposition of a vector addition application in mobile and hetero objects.

of such decomposition for a simple vector addition application. Dotted lines represent a vector row; multiple such rows constitute a hetero object, depicted as a solid rectangular, while multiple hetero objects form a mobile object, as shown by the dashed rectangulars.

In general, an application should try to form mobile objects from hetero objects that are accessed (from tasks) together to minimize inter-node communication when this is possible while assigning hetero objects as chunks of independent data that are big enough to saturate computing devices. In our example, each mobile object includes rows from each array to allow the partial results to be computed independently without any communications, while each computing node can inject big enough pieces of work to local devices by separating several rows of an array into a single hetero object.

In the homogeneous version of PREMA, mobile objects were created using a pointer to user data along with a set of callbacks to serialize and deserialize this data. With the introduction of hetero_objects, this process is separated into a collection of hetero_object that define their own methods to (de)serialize data which are utilized implicitly through a call to the respective pack()/unpack() functions. By utilizing the individual (de)serialization methods along with the mobile object level callbacks, the runtime system can (de)serialize mobile objects consisting of hetero objects and implicitly transfer them to other nodes.

An application is encouraged to follow an object-oriented design where the different data members are abstracted as hetero objects (other datatypes can also be included), and methods use the hetero task interface. An example of the vector addition application is presented in Fig. 53 where the hetero objects representing the partial vectors x, y, and z are members of the class that implements a mobile object (line 19), and the vector addition task is issued through the vec_add method. Following this design, it is easy to implement the application logic around the provided abstractions.

```
 1
 2  class vec_add_mo
 3  {
 4  public:
 5      vec_add()
 6      {
 7          for( int i = 0; i < N; i++)
 8          {
 9              hetero_task add_task;
10              ...
11              add_task.arg(m_x[i]).read();
12              add_task.arg(m_y[i]).read();
13              add_task.arg(m_z[i]).write();
14              add_task.device(GPU);
15
16              add_task.submit(vec_add_kernel);
17          }
18      }
19      ...
20
21      hetero_object <double> m_x[N], m_y[N], m_z[N];
22  }
```

Fig. 53. A possible design for the mobile objects of the vector addition example.

Mobile Handlers as Sets of Heterogeneous Tasks

Another change introduced in the hybrid model is found in the mobile handlers' context. Mobile handlers now comprise a set of hetero tasks that access and manipulate hetero objects through the interface presented in the previous section. Thus, a mobile handler can issue separate, asynchronous tasks on different hardware devices on the same node where data transfers and task dependencies are handled automatically. Moreover, PREMA ensures that the data and hetero_objects encapsulated by the mobile object remain "locked" in the node while the hetero_tasks run even if the handler has returned.

By adhering to the hetero task and hetero objects model, mobile handler executions are becoming heterogeneity-aware. Mobile handlers are, by definition, executing -mostly- on data present on a single node, encapsulated in a mobile object, though they can involve data passed by the handler-caller (which can be remote) as arguments of the handler. PREMA is responsible for implicitly transferring the data encapsulated by a hetero object that is an argument of a mobile handler and providing a handle to it at the execution target. Thus, the application is provided with a higher-level construct that abstracts the low-level implementations while also allowing the runtime system to optimize inter-node communications by issuing transfers directly from the device they reside in instead of moving them to the host first.

5.3 PERFORMANCE EVALUATION

This section investigates optimizations that can improve the performance of different operations provided by the heterogeneity-aware PREMA and the tasking framework and presents the performance of the optimized implementation on a proxy application. We used a small 16-node cluster, each consisting of two Intel Xeon Gold 6130 20-core CPUs (2.1 GHz) and four NVIDIA Tesla V100 GPUs.

5.3.1 Heterogeneous Tasking Framework

We evaluate different performance optimization techniques on this framework for CUDA GPUs using a simple, double precision matrix-matrix multiply benchmark and compare the throughput achieved in 100 iterations with a pure CUDA implementation. In each iteration, the three matrices are allocated in the device, input data are transferred, and the compute kernel is executed. Note that the results are not copied back to the host to prevent the pure CUDA implementation from blocking. We incrementally apply optimizations and show their effect. As shown in Fig. 54, the baseline bar indicates that the framework adds some overhead on top of the naive CUDA imple-

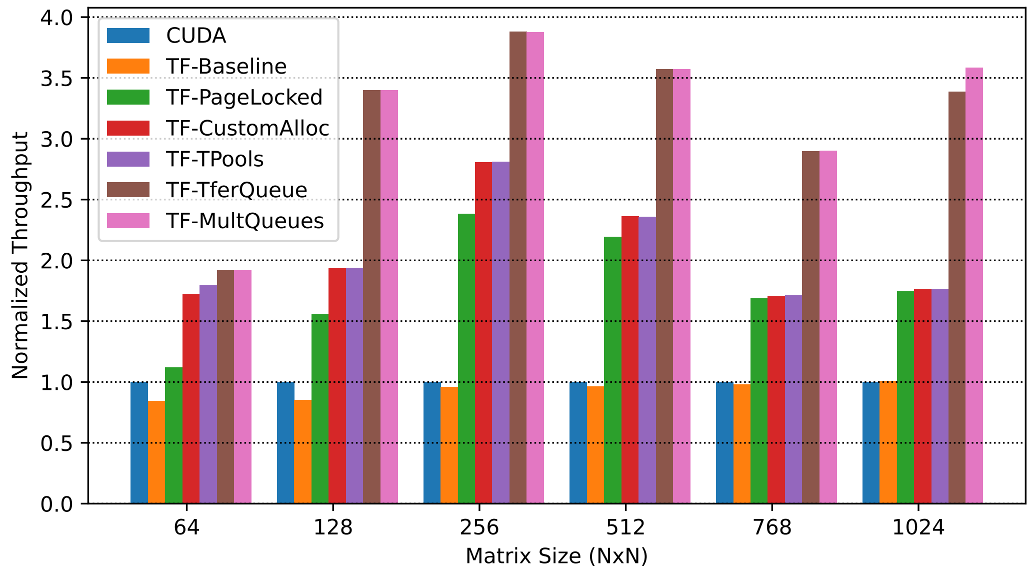


Fig. 54. Performance optimizations on the tasking framework. The effects of incrementally applying optimizations on the heterogeneous tasking framework's performance are evaluated using a matrix multiply benchmark on a single GPU. The performance improvements are evaluated on multiple matrix sizes and normalized by a CUDA implementation's performance. **CUDA**: The baseline CUDA implementation. **TF-Baseline**: The tasking framework (TF) without any optimizations. **TF-Pagelocked**: TF utilizing Page-locked host memory. **TF-CustomAlloc**: TF after applying the custom memory allocator optimization. **TF-TPools**: TF utilizing request pools. **TF-TferQueue**: TF after introducing a separate queue for memory transfers. **TF-MultQueue**: Final version of TF, utilizing multiple queues for kernel invocations.

mentation, which is significant, especially for smaller matrices. PREMA overcomes some of these overheads with the optimizations presented. Note that many optimizations applied automatically by the heterogeneous framework could also have been implemented directly in the CUDA implementation. However, it would require much more effort from the user and a relatively more extensive and more complex application code. Moreover, the code would need to be rewritten to run on non-NVIDIA devices.

Page-locked Host Memory

To fully utilize the bandwidth capabilities of the respective hardware, NVIDIA GPUs require that host data to be transferred to the GPU should reside in a page-locked memory region. Moreover, this is the only way that device-to-host transfers can be asynchronous with respect to the host. Thus, applications need to explicitly (de)allocate memory in a unique way, which increases code complexity and induces an overhead much higher compared to regular memory allocations. We incorporated this optimization into the framework to relieve applications and PREMA from explicitly handling this burden. For this purpose, the framework allocates a large chunk of page-locked memory at the initialization step that is later used as a memory pool for host memory allocations and prevents further expensive requests for page-locked host memory allocations.

The optimization gives a significant boost that ranges between 30% and 145% for different matrix sizes (Fig. 54; TF-PageLocked). Specifically, the smallest improvement is observed in smaller matrices (64x64) with 30%, followed by the larger ones (768x768 and 1024x1024) with about 70%. In the 128x128 case, it attains 90%, while the most notable improvement with more than 100% is achieved when 256x256 (120%) and 512x512 matrices (145%) are used.

Custom Device Memory Pools

Allocating and freeing device memory are expensive operations that may require synchronization between the host and the device. Moreover, the two functions might require the completion of previously issued asynchronous operations before running. The proposed runtime system uses a custom memory allocator per device to avoid overheads from constantly requesting new memory (de)allocations. During the initialization of the runtime system, a request to allocate most of the available memory of each device (except the host) is issued, and the custom memory allocator handles the returned memory. When memory needs to be allocated, the custom allocator is used instead of the one provided by the device library.

The most significant improvement of this optimization is manifested for the smallest matrix case (64x64) with another 60%. As the size of the matrices increases, the improvement observed declines with 25%, 12%, and 6%, respectively, for matrices ranging from 128x128 to 512x512. For larger matrices, the improvement is negligible (Fig.54; TF-CustomAlloc).

Enabling Concurrent GPU Operations

So far, all the optimizations implemented were focused on improving the overlap of the CPU and GPU operations; however, processes that run in the GPU are still serialized by default. We enable implicit concurrency between the different GPU operations by utilizing multiple execution streams provided by the device implementations (OpenCL command queues or CUDA streams for NVIDIA GPUs). Our implementation uses multiple streams for submitting computation kernels and two for memory transfer operations (one for each direction). We found that five streams for

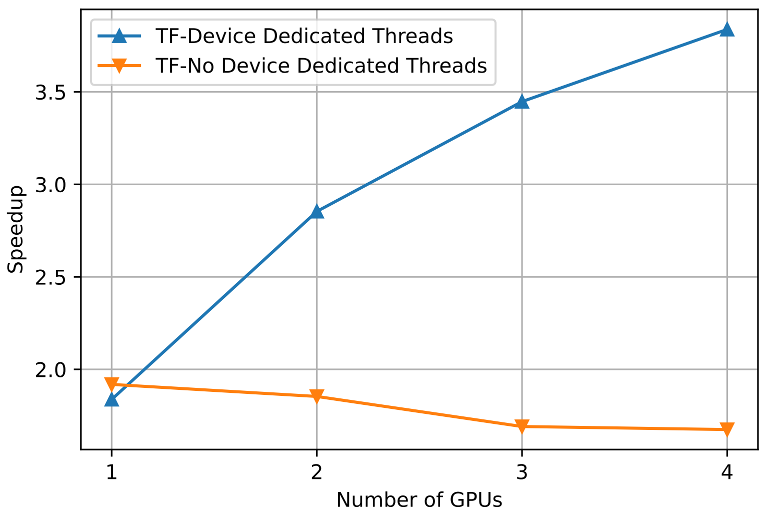


Fig. 55. Performance of the heterogeneous tasking framework on multiple device. Evaluated on a 64x64 matrix multiply benchmark utilizing multiple GPU devices with and without dedicated threads per device. The performance is shown as speedup against a simple CUDA implementation.

computation kernels are generally enough to saturate the device capabilities for concurrent kernel executions. Still, we also provide an environmental variable that allows the application to change this without recompilation.

The results of this optimization are substantial in most matrix sizes when the overlap between memory transfers and kernel executions can be large enough. This is the case for all matrix sizes larger than 64x64. Since the overhead of transferring data to the device is substantial, a great percentage of that can be overlapped with the kernel execution of the previous iteration. Thus, the results show improvements of 75%, 50%, 50%, 85%, and 100%, respectively, for matrices of size 128x128-1024x1024. On the other hand, the improvement in the case of 64x64 is 10% due to the small amount of data that needs to be transferred (Fig.54; TF-TferQueue, TF-MultQueues). The final optimization applied in this aspect was to use more than one stream per data transfer direction. However, this did not improve performance further because the device hardware only supports two copy engines.

Other optimizations

We have introduced request memory pools to mitigate the effects of system calls and thread synchronizations. Request pools are maintained per active thread, and the memory of a request is recycled in the pool once the respective operations have been completed. Another optimization regarding requests was implemented on the queues used to submit a request to a device. Initially, we used structures provided by the C++ STL, protected by a mutex, to implement such queues. These queues were substituted with custom lists that avoid allocating nodes to store new elements. Moreover, the mutex locking step was moved after the queue's size was checked to eliminate unneeded locking operations. This optimization is less important than the ones discussed above, about 2% improvement, but helps to attain a more consistent latency, especially when the dedicated thread is used (Fig.54; TF-Tpools).

Putting it all together

To summarize, we have applied a series of optimizations that affected the performance of the tasking framework on the specific benchmark in different percentages depending on the size of the matrices. The most important optimizations include the automatic use of page-locked host memory and the introduction of multiple streams. However, the custom device memory pool also substantially improved the cases of smaller matrices. The overall performance improvement achieved from this series of optimizations can exceed 300%, depending on the size of the matrices. Our framework offers all these optimizations with minimal user involvement and will continue to improve without any modifications required in the application code. Note that the performance improvements will increase as the number of memory transfers per computation decreases.

Multi-Device Platforms

The above results show the performance of the heterogeneous tasking framework on a single device. However, our framework is able to utilize multiple devices automatically. This is where the importance of using dedicated threads per device becomes apparent. While dedicated threads do not help when only a single GPU is in use (as shown in Fig. 55), they are crucial to scale beyond a single device. Fig. 55 shows that the framework can scale almost linearly as we add more devices, achieving up to 3.8 speedup on 4 GPUs. The superlinear speedup observed in the case of one and two GPU devices is an effect of the optimizations presented so far for a single device. As we add more devices, the effect of these optimizations declines, and the speedup becomes linear. We must note that the hardware available to us constraints the framework's capabilities. All the devices in the specific machine share a single bus with the host. Thus, the maximum memory transfer throughput is constant whether one or four devices are utilized. For the specific benchmark, a maximum of 2 64x64 matrices can be moved simultaneously to any device. Given more capable hardware, we expect the framework to perform much better in a multi-device context.

5.3.2 Heterogeneous PREMA

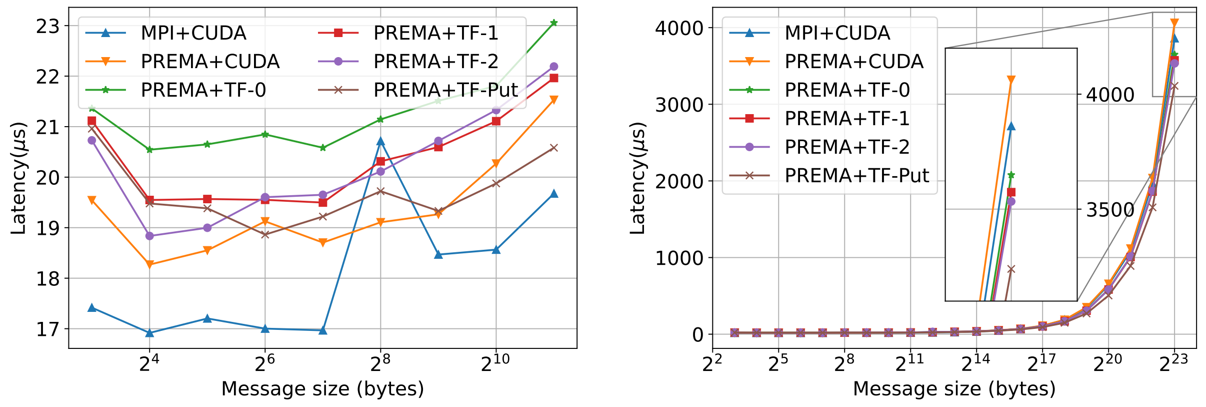
To optimize the performance of the new heterogeneity-aware version of PREMA, we experiment with optimizations that can help us mitigate the overheads following the implementation of remote handler invocations that include heterogeneous memory both without and with the tasking framework (without a dedicated thread). We evaluate our optimizations on a simple ping-pong benchmark for inter-node communications and compare it with an MPI+CUDA implementation. The benchmark runs 100 ping-pong iterations with message sizes ranging between 8 bytes to 8 MBs, and the average latency and bandwidth observed per message size are reported.

Device Message Receiving Cache

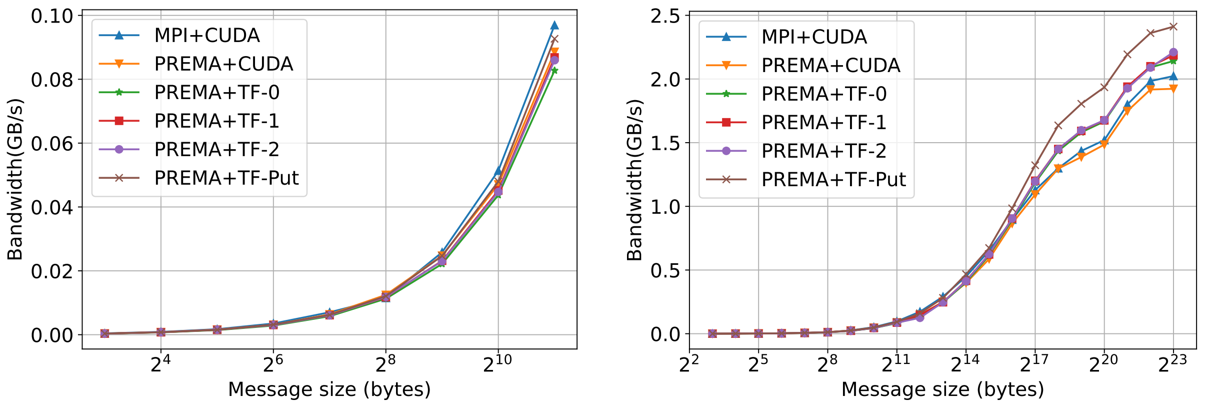
In PREMA, messages are one-sided and asynchronous and are received implicitly to invoke a designated task on their target. Thus, the receiver cannot specify a memory region where the message buffer shall be stored (like MPI). PREMA has to dynamically allocate memory to receive the incoming message buffer and provide it to the application handler invocation. In our initial implementation, without the tasking framework, simply allocating new device memory for each incoming message resulted in poor performance, increasing the latency experienced up to ten times compared to the respective MPI implementation. We avoided this behavior by allocating a cache in the device specifically for the buffers of received messages. When a new message buffer is about to be received, memory is requested from the cache instead of the device API if possible. The cache allowed us to attain performance within 10% overhead of that achieved by the MPI (Fig. 56; PREMA+CUDA). It is also important to note the consistent “spike” observed in the MPI performance for messages of size 256B, which does not seem to affect our implementation when using the device cache.

Preallocating hetero_objects

Hetero_objects automatically utilize memory pools for device memory, thus, implicitly overcoming the issue faced in the case where device memory is handled explicitly and achieving performance within 25% of the MPI+CUDA implementation (Fig. 56; PREMA+TF-0). However, we



(a)



(b)

Fig. 56. Performance evaluation of different optimization techniques. Evaluations performed in terms of (a) latency and (b) bandwidth on a ping-pong benchmark without direct network device-to-device transfers. Optimizations are compared against the baseline performance of an MPI+CUDA implementation. **PREMA+CUDA**: Integration of PREMA with CUDA directly, without involving the heterogeneous tasking framework. **PREMA+TF-0**: The baseline integration of PREMA and the heterogeneous tasking framework. **PREMA+TF-1**: PREMA+TF-0 plus the introduction of memory pools. **PREMA+TF-2**: PREMA+TF-1 plus the optimization of avoiding message buffer copies for very small messages. **PREMA+TF-Put**: The remote put operation after the optimizations of PREMA+TF-2.

can still improve some latencies caused by the constant allocation and deallocation of temporary hetero_objects that wrap message buffers targeting device memory. Specifically, we found that the data structures allocated for a hetero_object for bookkeeping different operations targeting the object in various devices can significantly affect communication performance. Since these structures can be allocated in advance, we use a pool of preallocated semi-initialized hetero_objects to mitigate this effect. This optimization improved the latency experienced for small messages by about 10%, bringing the performance of PREMA within 15% overhead of MPI+CUDA, as can be seen in Fig. 56 (PREMA-TF-1). Moreover, for larger messages, the performance achieved is better from the MPI+CUDA by almost 10%, as can be seen when zooming in the latency of 8MB messages.

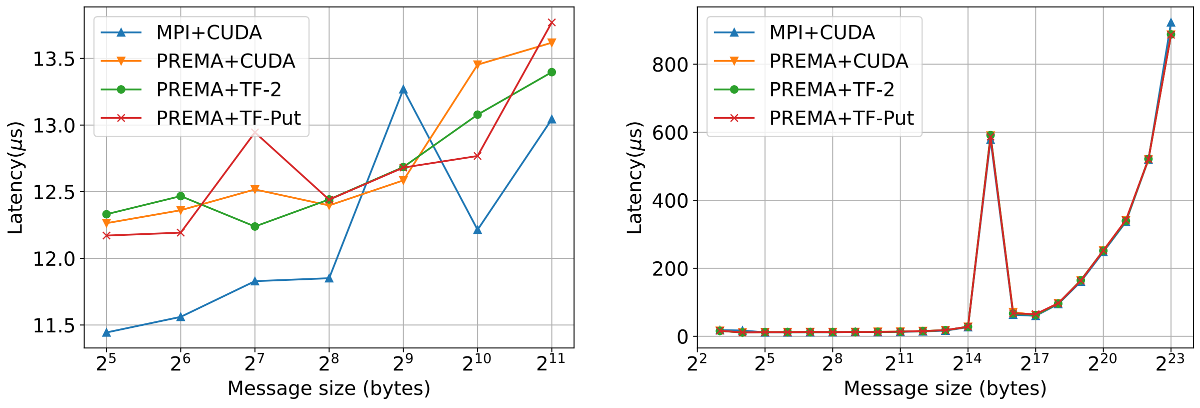
Avoiding Message Buffer Copies

The process of inter-node message transfers presented in 5.2.3 with host-staging includes an optimization for small buffer sizes to only send one message. Small messages with a size up to 512 bytes (header + buffer size) will consist of the actual application data/buffer appended at the end of the message. This helps optimize the performance of small messages where even negligible overheads are noticeable. However, when hetero_objects are used, this introduces an extra copy. As explained before, requesting access to the underlying data of a hetero_object will implicitly copy the data to the host in a buffer maintained by the framework. To append the data at the end of the message header, PREMA needed to copy the data from the framework's location to the message header.

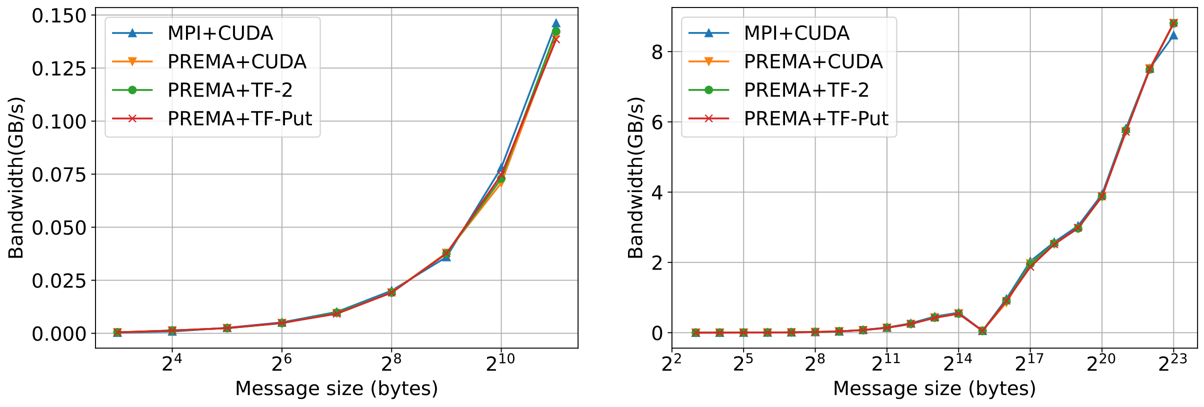
A new method is introduced in the hetero_object API that allows the user to request a copy of its underlying data to a designated memory region at the host. PREMA utilizes this feature to request from the framework to directly transfer the device data at the end of the message header buffer. This change slightly improves the communication critical path and attains up to 5% lower latency for smaller messages (Fig. 56; PREMA-TF-2).

Put Operation

Another operation introduced to further increase the performance of inter-device communication over the network is the put operation. As mentioned earlier, the put operation allows PREMA to utilize the existing memory of heterogeneous objects that is also page-locked. By leveraging these optimizations, the put operation outperforms all previous optimizations that use the tasking framework since the transfer from the host to the device is much faster on the receiver side. Fig. 56(PREMA-TF-Put) shows its performance reaching and overcoming the one of the

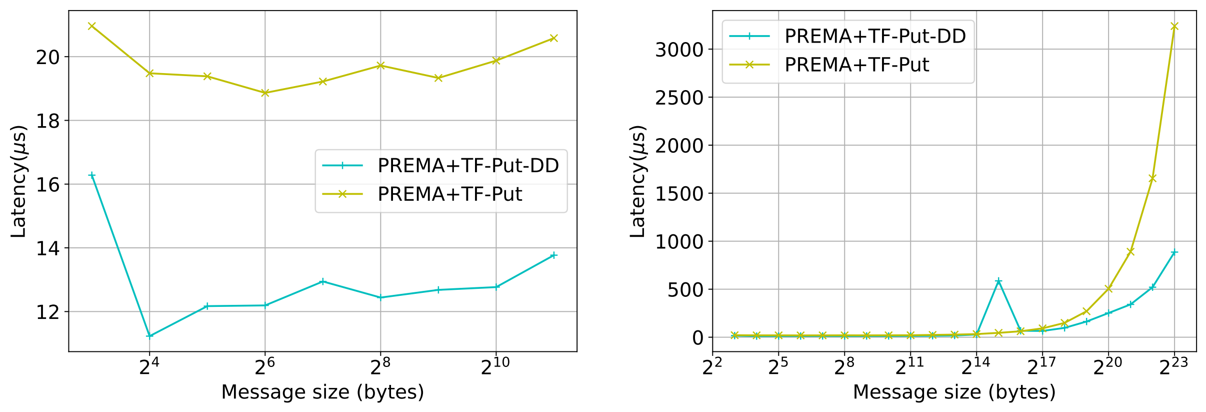


(a)

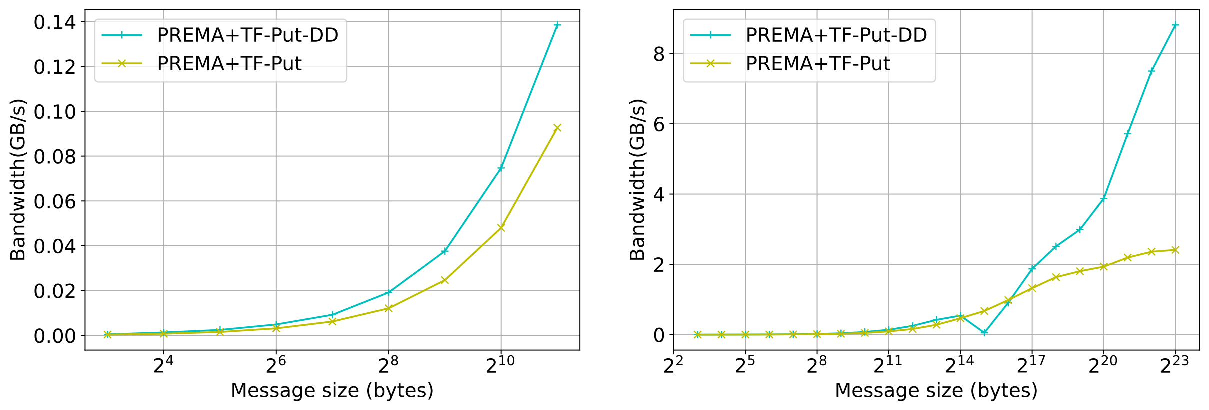


(b)

Fig. 57. Performance evaluation of different optimization techniques. Evaluations performed in terms of (a) latency and (b) bandwidth on a ping-pong benchmark with direct network device-to-device transfers. Optimizations are compared against the baseline performance of an MPI+CUDA implementation. **PREMA+CUDA**: Integration of PREMA with CUDA directly, without involving the heterogeneous tasking framework. **PREMA+TF-2**: Integration of PREMA and the heterogeneous tasking framework incorporating all the optimizations presented in this section. **PREMA+TF-Put**: The remote put operation after the optimizations of PREMA+TF-2.



(a)



(b)

Fig. 58. Comparison of the optimal performance. Evaluations performed in terms of (a) latency and (b) bandwidth using host-staging or Device-to-Device (DD) network transfers for the ping-pong benchmark.

PREMA+CUDA implementation for messages larger than 64 bytes and even the MPI+CUDA for messages larger than 8KB achieving up to 20% better performance for messages of 8MB.

Direct to Device Transfers

The optimizations presented so far target the generic implementation of heterogeneity on top of PREMA, where the communication library/hardware is not heterogeneity-aware. However, as mentioned in detail in 5.2.3, PREMA can leverage the capabilities of hardware/libraries that have been integrated with support for direct device-to-device communication. Following the previously presented procedure, the latencies observed for heterogeneity-aware hardware can be significantly mitigated. Fig. 57 shows the performance of the optimized version of each operation and the MPI+CUDA when direct-to-device communication is possible. The attained performance, in this case, is up to 100% better than the host-staging case for small messages and up to 200% for large messages, as shown in Fig. 58.

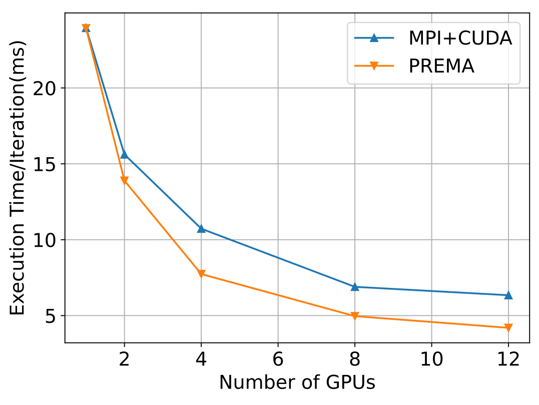
Putting it all together

Overall, all three operations introduced in PREMA to handle heterogeneity, including the transfer of CUDA buffers, hetero_objects, and the remote put operation, significantly simplify application development, saving hundreds of lines of boilerplate code while maintaining reasonable overhead (10-15%) compared to MPI+CUDA regarding latency and bandwidth. Moreover, it is interesting to notice that some of these operations, like the put operation, outperform MPI+CUDA (by up to 20%) for large messages (Fig. 56, 57) by implicitly leveraging from the page-locked memory of hetero_objects. Page-locked memory forces the operating system to lock a virtual memory address to a specific physical address, allowing CUDA GPUs to use DMA and significantly improve the throughput of read and write operations.

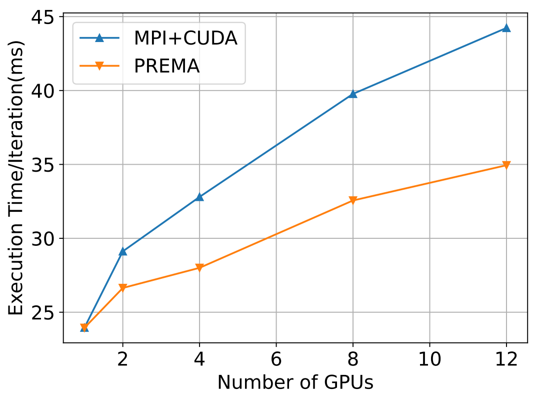
5.3.3 Proxy Application: Jacobi3D

To evaluate the performance of the distributed memory framework, we have adapted a proxy Jacobi3D application [81] on PREMA. The proxy performs a fixed number of iterations of the Jacobi method on GPUs in a 3D domain decomposed into cuboids and wrapped into mobile objects. In each iteration, the mobile objects exchange halo data, packing the GPU data and transferring them to their neighbors. On the receiving side, the data are unpacked into the GPU, and once all halos have been received, the Jacobi update is executed.

Fig.59 (left) shows the execution time of the heterogeneous PREMA versus the MPI + CUDA



(a)



(b)

Fig. 59. (a) Strong and (b) weak scaling performance of the Jacboid3D proxy application.

counterpart for a 1024x1024x768 domain (strong scaling). The implementation with PREMA achieves up to 23% better performance. For weak scaling, the domain's size is increased according to the number of distributed GPUs. The performance achieved is up to 25% (Fig.59; right) better than the MPI+CUDA implementation. The improvements observed stem from automatically overlapping message passing, host-device memory transfers, and kernel invocations.

5.3.4 Over-Decomposition

A common practice that PREMA applications utilize for performance improvement is over-decomposition. Over-decomposition is used to decompose the data domain into more chunks than the number of PEs, allowing PREMA more flexibility to load balance workload and overlap latencies. The effectiveness of this approach has already been demonstrated in previous work for heterogeneous platforms [47, 80]. In the context of heterogeneity, host-to-device and device-to-host memory transfers are broken into pipelined pieces and can be overlapped much more easily with the following kernel invocations. An example of the execution timeline that is achieved can be seen in Fig. 60. The effects of over-decomposition are shown in Fig. 61 for the same Jacobi3D proxy application. Different levels of over-decomposition are attempted in this benchmark, with each level attaining improvements over the MPI implementation as well as the PREMA implementation without over-decomposition (OD1). The best performance is observed with an over-decomposition of two, which achieves improvements of up to 40% versus the MPI implementation and about 20% over the initial PREMA implementation.

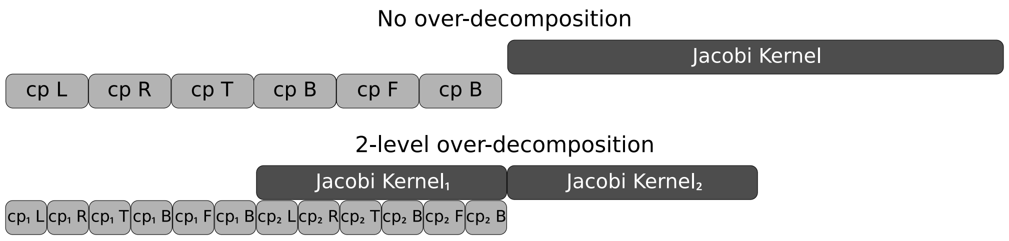


Fig. 60. An example of the Jacobi3D execution timeline with no and 2-level over decomposition. Each kernel invocation requires the copy of the six halo buffers, left (L), right (R), top (T), bottom (B), forward (F), and back (B), into the GPU. By decomposing the data domain (and the kernel invocation) into two pieces, PREMA can overlap the memory transfers of the second with the kernel invocation of the first, saving a significant amount of time.

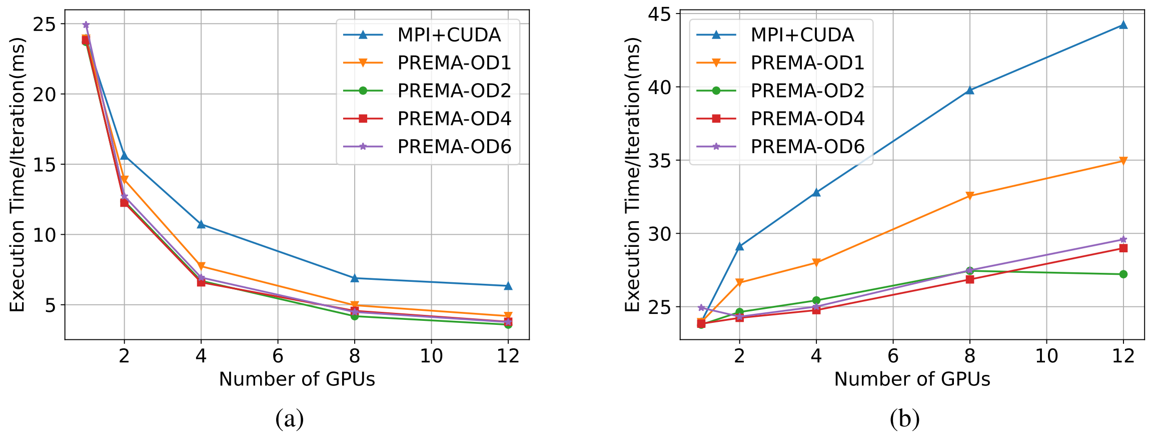


Fig. 61. (a) Strong and (b) weak scaling performance when utilizing over-decomposition.

5.4 SUMMARY

This chapter introduced an efficient tasking framework that handles performance portability for multi-device heterogeneous nodes. This framework automatically scales applications to multiple devices while managing efficient scheduling, load balancing, task dependencies, memory transfers, and overlapping latencies, attaining a performance improvement of up to 300%. Integrating this framework into a distributed runtime system resulted in a complete library leveraging exascale HPC systems consisting of multiple heterogeneous nodes. In addition, this chapter presents a series of optimizations and their evaluation which indicates mitigated overheads to within 10% of the MPI. Evaluation results on a proxy application show that the end product of this work incurs low latency and scalable performance (up to 40% versus the MPI+CUDA) while providing a simple and uniform interface independent of the target hardware.

CHAPTER 6

UTILIZING MACHINE LEARNING IN NUCLEAR PHYSICS ACCELERATORS

In nuclear physics, experiments measuring scattered particle parameters are the most computationally intensive process. This process relies on measurements of particle tracking detectors to construct a particle trajectory by combining the detected hits and resolving the particle momentum via fitting the trajectory points (using Kalman Filter[1]). In high luminosity experiments (where multiple particles are produced as a result of an interaction, and noise is present in particle tracking detectors), the process of isolating detector hits for each particle trajectory relies on considering each combination of hits that can potentially form a track and then fitting each hypothesis to determine which one represents a valid trajectory. This process can be time-consuming, amounting to about 94% of the total data post-processing time. Recent advances in artificial intelligence and machine learning create the opportunity for substituting some of the existing algorithms with predictions from machine learning models. This substitution reduces the code complexity needed to select the correct track hit combinations by providing only the most likely track trajectory candidates and then using the identified tracks to extract the track parameters. This work focuses on the track-candidate identification process and hit-based track parameter extraction for the CLAS12[2] detector at Jefferson Laboratory (JLab), Newport News, Virginia. We split this process into three steps. First, we investigate the usage of Convolutional Neural Network Auto-Encoders (CAE) for denoising raw hits from drift chambers and removing as many uncorrelated signals as possible to simplify the process of valid track identification (track denoising). Next, we study different machine learning models and construct a model that can identify the candidate with the highest probability of representing a real track out of candidates formed by combining various particle detections (track classification). Finally, a third machine learning model extracts the track parameters (e.g., momentum, polar and azimuthal angles) required for physics analysis.

The major contributions in this chapter are as follows:

- A Convolutional Autoencoder (CAE), able to denoise the detector observations used for track reconstruction. The algorithm improves tracking efficiency by more than 15% in real data production procedures with nominal conditions and up to 200% in synthetically generated data with high luminosity conditions (90 - 110 nA).

- An MLP network classifier, implemented as part of the CLAS12 reconstruction software, providing tracking code with recommended track candidates. The classifier achieves accuracy greater than 99% and results in an end-to-end speedup of 35% compared to conventional algorithms.

- A machine learning regressor network capable of reconstructing particle track parameters (particle momentum, and polar and azimuthal angles) with accuracy similar to conventional methods but up to 150 times faster.

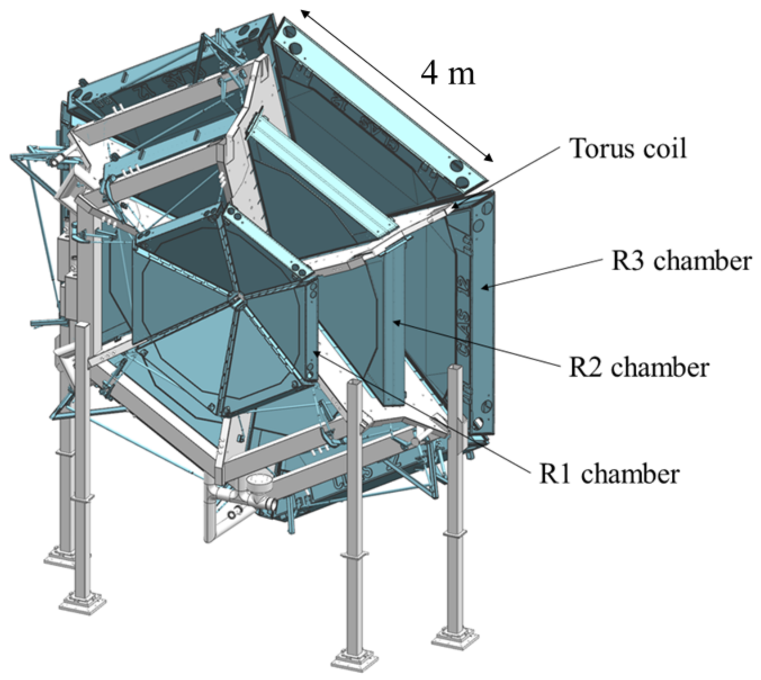
6.1 BACKGROUND

6.1.1 CLAS12 Detector

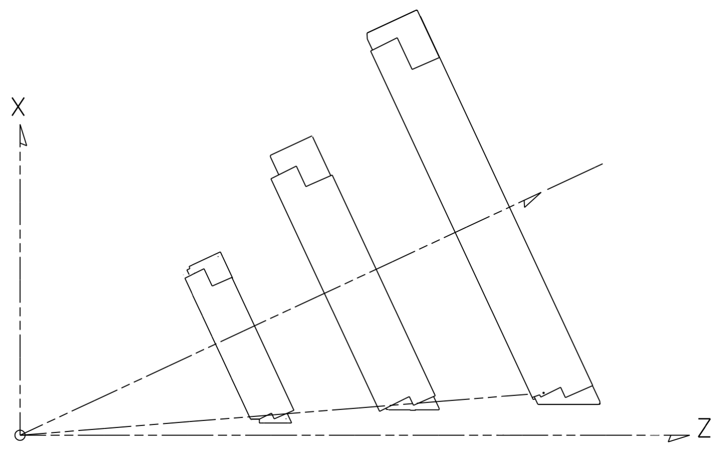
The CEBAF Large Acceptance Spectrometer at 12 GeV (CLAS12 [2]) is located in Hall B, one of the experimental halls at the Jefferson Lab in Newport News, VA, serving a variety of physics experiments with different running conditions.

The forward part of CLAS12 is built around a superconducting toroidal magnet (Figure 62a). The six coils of the toroid divide the detector azimuthally into six sectors. Each sector contains three multi-layer drift chambers [82] (DC) for reconstructing the trajectories of charged particles originating from a fixed target. Figure 62b depicts the cross-section of one of the sectors of drift chambers covering an azimuthal angular range of 60°. One sector comprises three drift chambers (called “regions”), each consisting of two sections (called “super-layers”).

Particles originating from the target within a polar angular range from 5° to 40° leave signals on all six super-layers and can then be reconstructed by a tracking algorithm. The track reconstruction algorithm for CLAS12 [83] works in two stages, hit-based tracking and time-based tracking. In the hit-based tracking stage, the Simple Noise Removal (SNR) algorithm identifies and removes uncorrelated noise hits in Drift Chambers. The remaining hits are then grouped into clusters within the wire layers of a given DC super-layer. To reduce tracking inefficiencies attributed to wire malfunctions or intrinsic inefficiencies, it is acceptable for two layers to be missing within a super-layer when forming a cluster; the remaining wire layers are sufficient to determine the cluster’s shape and find the track trajectory. After identifying clusters in each super-layer and forming track candidates with combinations of all of them (one from each super-layer), an initial fit to the track candidates is performed to determine if the cluster combination forms a “good” track and then to extract track parameters such as momentum and polar and azimuthal angles, based only on the positions of the hits (clusters); a process called hit-based tracking. Tracks identified as good continue to the next stage (called time-based tracking), where timing information from hits is applied to refine track fitting. The initial determination of track parameters based on hits



(a)



(b)

Fig. 62. View of the CLAS12 detector. (a)The Drift Chambers and Toroidal Magnet and (b) a side view of one of the sectors of Drift Chambers, consisting of three “regions” and covering 60° azimuthal range.

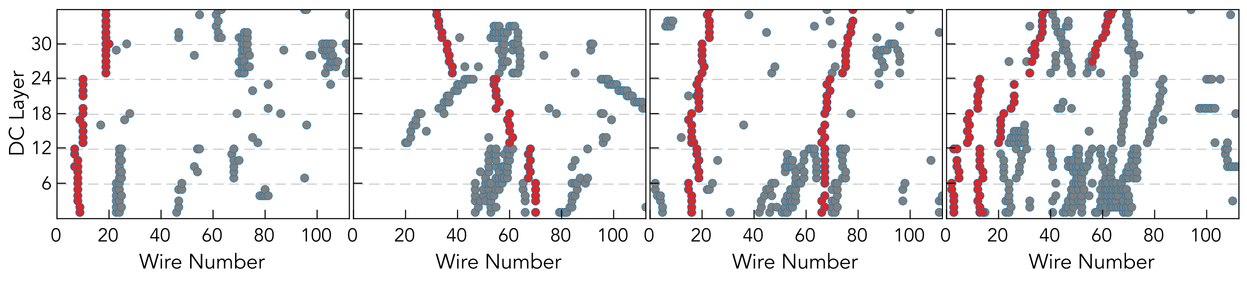


Fig. 63. Example of Drift Chamber data. Each circle represents a wire with hits identified as belonging to a track by the tracking algorithm annotated with red. Each plot presents data from one sector from different events. Cases with one and two tracks in one sector are shown.

is essential for determining event start time at the target since it leads to more refined time-based tracking.

Drift Chambers

The Drift Chambers in CLAS12 are used for tracking charged particles. Each sector comprises three drift chambers (called "regions"), each consisting of two sections (called "super-layers"). Each super-layer has six layers of wires (12 wire planes in each "region"), with those residing on the two adjacent super-layers oriented at $\pm 6^\circ$ stereo angles. Each layer of wires has 112 hexagonal cells spanning a range from about 5° to 40° in polar angle.

Figure 63 presents some example events for one sector at a time. The gray dots depict raw hits detected on a sector, while the red ones depict only those belonging to identified tracks. Hits other than the red ones comprise the uncorrelated hits of the event. Particles passing through each super-layer of chambers leave a signal in one or two wires in each layer. Wires with signals on each super-layer close to each other are grouped into segments, and a track candidate is formed from 6 segments (one per super-layer) in each sector and then further validated by fitting. The efficiency of track reconstruction relies on cleanly identifying segments in each super-layer. With increased noise arising from running with high beam intensity, detecting valid tracks with conventional algorithms becomes less efficient, and with the loss of segment, the tracking efficiency suffers.

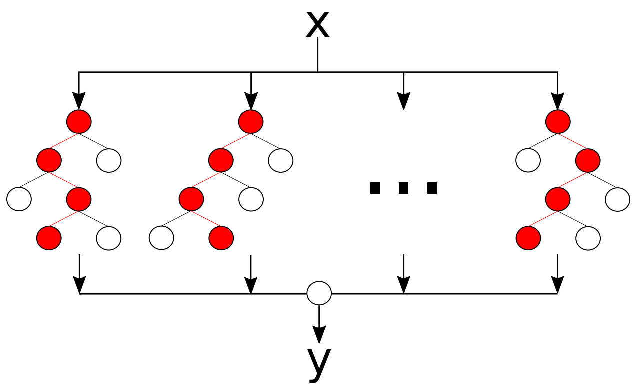


Fig. 64. An illustration of the extremely randomized trees decision-making algorithm.

6.1.2 Machine Learning

Extremely Randomized Trees

ERT [84] is a supervised learning algorithm comprising an ensemble of randomly generated decision trees [85]. A decision tree is formed by recursively dividing the dataset into subsets. The splitting criteria are formed by the algorithm based on the classification features such that the derived subsets are optimal. The recursive splitting process completes when the derived subset consists only of the same class elements or when splitting does not add any extra value. However, this algorithm introduces a high probability of overfitting the training data, in other words, creating a model that cannot generalize to new data. For example, it could generate a tree with a leaf for each instance in the training set, which would not be reusable when used on a different dataset. To mitigate such issues, one can use multiple decision trees formed by random subsets of the dataset. The randomness introduced by generating new decision trees dramatically improves the model's prediction power. A method that follows this approach by picking random subsets of the input dataset with replacement is called random forest [86]. ERT is an extension of the random forest method that incorporates more randomness by randomly selecting the splitting criteria instead of choosing the best split. Moreover, this method forms the different random trees using subsets of the input dataset without replacement. The final prediction of this model is produced by taking the average of the predictions of all random trees. Figure 64 shows an example of the ERT decision algorithm.

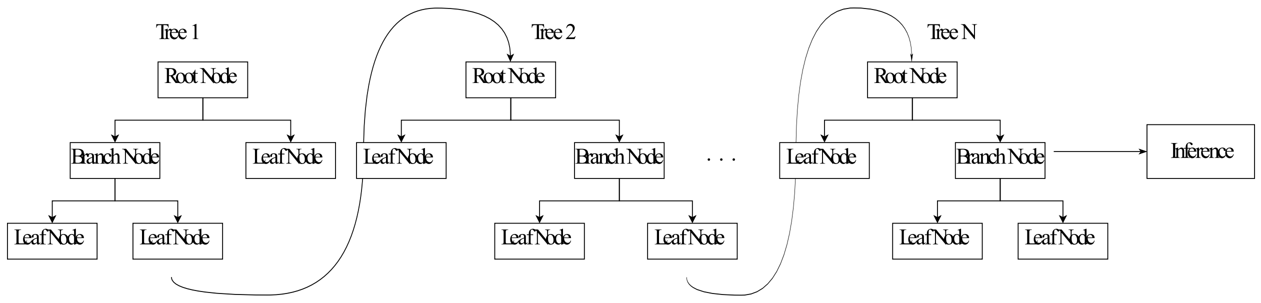


Fig. 65. An example of the GBT algorithm. An ensemble of trees is formed sequentially, with each new tree optimizing the residuals of the previous one. Inference is performed by incrementally combining the results of individual trees in the order they were created.

Gradient Boosting Trees

Gradient Boosting Trees (GBT) [87] is a supervised learning algorithm that, like Extremely Randomized Trees, constitutes an ensemble of decision trees. The main difference between the two models is how they combine decision trees. In contrast to ERT which builds decision trees in parallel by fitting different subsamples of the dataset, GBT builds new trees sequentially and incrementally (Fig. 65). The goal of each new tree is to minimize the error of the previous tree; thus, each tree fits the residuals from the previous one (hence the name gradient boosting). Another difference is regarding how inference is run. While ERT outputs a decision by aggregating the predictions of all trees at the end of the process (by averaging or majority vote), GBT combines results incrementally (e.g., by iterating the created trees sequentially and aggregating). Finally, because new trees are fit to improve the errors of previous ones, GBT is more prone to overfitting when excessively increasing the number of trees. On the other hand, trees in ERT are independent; thus, increasing their number does not lead to overfitting.

Multi-Layer Perceptron

Multi-Layer Perceptron (MLP) [88] is a supervised learning algorithm that learns a function f that, given an input in space R^m, produces an output in space R^n by training on a given dataset. It consists of an input layer, an output layer, and one or more layers in-between called hidden layers. Each layer contains a number of neurons representing the parameters of the network. Specifically, for the input and output layers, those parameters represent the input $(\vec{x} : x_1, x_2, ..., x_m)$ and output values $(\vec{y} : y_1, y_2, ..., y_n)$ of function f. The neurons between two adjacent layers (l_{i-1}, l_i) are fully connected through weighted links. The values of the neurons of layer l_i are formed as a weighted linear summation of the neurons on layer l_{i-1} plus some bias b_i. Before these values are fed forward from l_i to l_{i+1}, a non-linear function, called the activation function, can be applied (e.g., hyperbolic tangent) to introduce non-linearity to the model.

The number of layers and neurons are referred to as hyperparameters of a neural network which need to be tuned for optimal results. Cross-validation techniques can be used to find the ideal values. Figure 66 shows an MLP with a single hidden layer, R^3 input, and R^1 output. The calculations that take place for the output(s) of each layer are presented below, where W is the matrix of the link weights, $\vec{x}$ the input vector of the layer, b the bias, $\vec{z}$ the output before applying an activation function ϕ and $\vec{a}$ the output after applying the activation function:

$$W\vec{x} = \begin{bmatrix} w_{11} & w_{12} & \cdots & w_{1m} \\ w_{21} & w_{22} & \cdots & w_{2m} \\ \vdots & \vdots & \ddots & \vdots \\ w_{k1} & w_{k2} & \cdots & w_{km} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_m \end{bmatrix} + b = \begin{bmatrix} z_1 \\ z_2 \\ \vdots \\ z_k \end{bmatrix}$$

$$\phi(\begin{bmatrix} z_1 \\ z_2 \\ \vdots \\ z_k \end{bmatrix}) = \begin{bmatrix} a_1 \\ a_2 \\ \vdots \\ a_k \end{bmatrix}$$

Once all layers have received and fed forward their respective values, the output layer produces an output evaluated against the ground truth using an error function. Training is then performed by applying backpropagation and optimization. In this stage, an optimization algorithm, like gradient descent, minimizes the error function between the predicted and ground truth values. This is achieved by back-propagating the output error to the hidden layers and adjusting the values of the weighted links between the neurons. In this way, the next forward pass should produce better results since the weights have been adjusted according to the expected output. By applying multiple iterations of feed-forward and backpropagation, the neural network optimizes its weights to "learn" the function f that will produce outputs with minimal error.

Convolutional Neural Network

Convolutional Neural Networks (CNN) is a type of neural network that performs better on data where spatial locality is important (e.g., images). A common CNN consists of Convolutional,

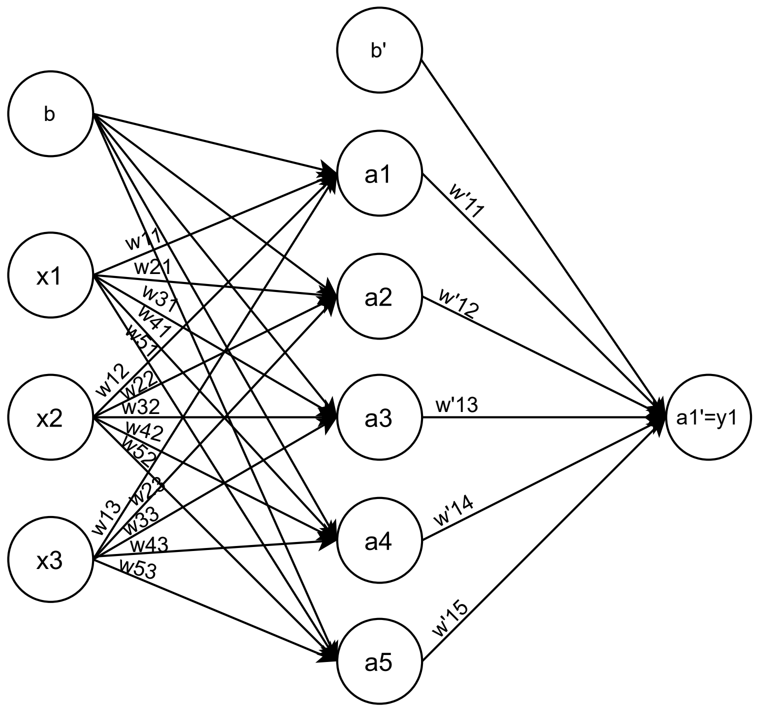


Fig. 66. A MLP with a single hidden layer of 5 neurons, three inputs, and one output.

Pooling, and Fully Connected Layers. Convolutional layers are the most important layers in a CNN, including an input, a number of kernels, and an output (feature map).

- The input to a CNN layer is an N-dimensional array (for example, a colored image with height, width, and RGB values) given as input to the network or produced by a previous layer.
- A kernel is a 2-dimensional array of weights that applies convolution on the input. The kernel is shifted on the input based on a stride, and on each shift, a dot product is applied between the respective area of the input and the kernel to produce a single element of the output (also known as a feature map). The process continues until the whole input has been processed and the entire matrix of the feature map has been produced. The kernel weights are the parameters the neural network learns and are adjusted between iterations through backpropagation and gradient descent.
- The result produced by applying convolution on the input using the provided kernel is the layer's output (feature map). Then, an activation function is finally applied to the output array (e.g., Rectified Linear Unit or ReLU [89]) to introduce non-linearity in the model.

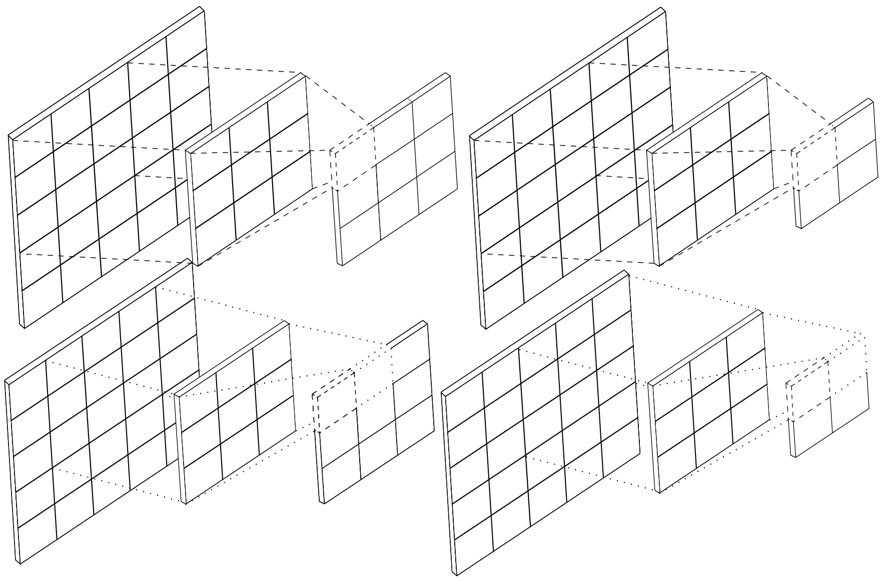
The size of the output is affected by three parameters: the *number of kernels*, the *stride* used during convolution, and the type of *padding*. The *number of kernels* determines the output depth (e.g., n kernels would generate n outputs, thus an output of depth n). Increasing the output depth can extract more features by training extra weights. As mentioned, the *stride* determines the step the kernel moves over the input to apply the dot product. A stride larger than one would generate an output smaller than the input. *Padding* can be either “valid” or “same”. Valid padding adds no padding to the input, resulting in an output of a smaller size for kernels greater than 1x1. This occurs because the kernel will be shifted on the input and apply the dot product fewer times than the input size. In other words, when a row or column of the kernel exceeds the input size, the dot product will be aborted. In same padding, the input is padded by zeros to ensure that the output will be the same size as the input.

Examples of the convolution process between a 5x5 array and 3x3 kernel for valid and same padding and strides of 1x1 and 2x2 are shown in Figure 67. When valid padding is used (Fig. 67a), the kernel can only be applied three times horizontally with a stride of 1x1 before moving to the next row. This process is repeated until no more steps are left to be taken horizontally and vertically for nine applications of the dot product (3x3 output). When using a stride of 2x2, the kernel is shifted by two elements at a time horizontally and two vertically when a row is completed, allowing the application of only four dot products in total (2x2 output). The same process is followed when same padding (fig. 67b) is applied, with the only difference being the introduction of more elements on the edges of the input array to ensure that the output of the convolution (assuming stride 1) will be the same size as the input. In both cases, the number of kernels equals the output depth.

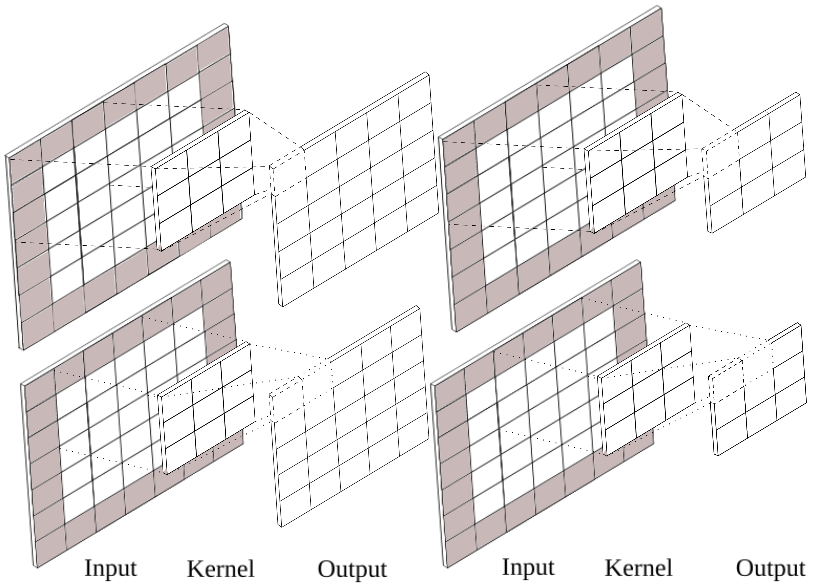
Pooling layers perform downsampling by sweeping a 2-dimensional filter over their input and applying some aggregation function to generate the output. The process is very similar to convolution, with the difference being that instead of a dot product, an aggregation function is applied (same as fig. 67a but with the kernel only applying an aggregation). The aggregation can be the maximum or average values in the area under the filter; hence, the names max pooling or average pooling, respectively. This process reduces the number of parameters to be learned to prevent over-fitting and reduce the training time. The fully connected layer is an MLP residing at the end of the CNN.

Recurrent Neural Network

Another approach that we investigate is the use of RNN[90]. RNN is a neural network that predicts dataset features that present a sequence. For example, given a sequence of temperature



(a) First two steps of convolution with valid padding and a stride of 1 (left) or 2 (right).



(b) First two steps of convolution with same padding and a stride of 1 (left) or 2 (right). The gray area in the input array represents the padding of zeros added to the input.

Fig. 67. Examples of convolution between with (a) valid and (b) same padding. The figures show the convolution process with a stride of 1x1 (left column) and a stride of 2x2 (right column).

data for the days in the previous month, an RNN can infer the temperatures of the next few days in the future. RNNs incorporate the connection between sequential data to infer the next data points in the sequence. To employ them for the needs of identification, we use the following observation: applying a well-trained RNN on a sequence that does not follow the same pattern as the training data (i.e., providing a false sequence) will result in predictions with a high degree of error.

In particle trajectories, a track can be presented as a sequence of hit detections on sequential layers of wires. Given a subset of a track, we used this observation to train an RNN using Gated Recurrent Units[91] (GRU) layers that can predict its missing parts. Specifically, our model is trained to infer the detections of the last 12 layers given information of the first 24. The RNN is trained on the same dataset as the models for trajectory classification, except that only data for valid particle tracks are utilized. This allows the RNN to predict valid particle tracks based on partial previous sensor activation patterns. Since the RNN is trained on only valid particle tracks, it will give incorrect predictions for invalid particle tracks. Therefore, by passing all particle candidates through the RNN, we produce a new set of inferred candidate tracks. We can extract the invalid and valid tracks by measuring the spatial distance of the inferred tracks and the actual track in the dataset. An inferred track far from the real candidate track in the dataset is considered invalid. Thus, the candidate used to generate it is also classified as invalid. On the other hand, when the candidate track in the dataset has a small distance from the inferred one, it likely means that the inferred track is valid, and so is the track that generated it. This process allows us to eliminate most invalid tracks and identify possible valid ones.

Auto Encoders

To our knowledge, no prior work has utilized AI to denoise data from hits coming from drift chambers. However, prior work in other fields has used auto-encoders for different applications, including image denoising, which inspired our work. Auto-encoders[92] are a type of Artificial Neural Network employed to learn efficient data encodings. In the simplest case, an auto-encoder is trained to recreate input data to its output. This process is performed to extract a representation of the data (*code*) in an unsupervised manner. This representation can then be used to recreate the input on the output using the decoder.

Auto-encoders consist of three components (see Fig. 68):

1. The encoder that encodes the input data, X

2. The decoder that reconstructs the encoded data in the output Y

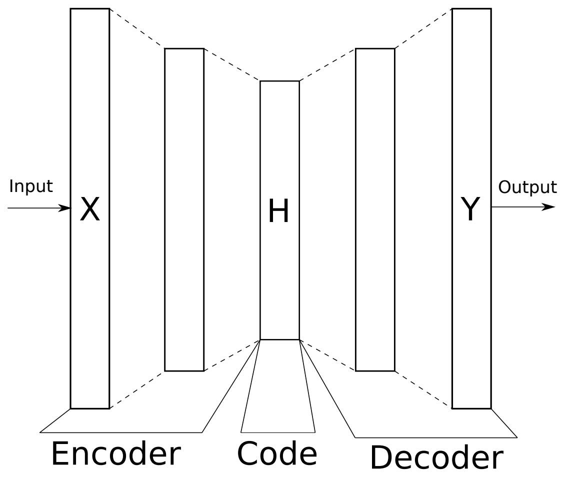


Fig. 68. Schematic representation of an encoder with three hidden layers.

3. The hidden layer H learns the encoding of the input data as defined by the encoder and is used by the decoder to reconstruct the input

The process followed by an auto-encoder is defined as the following transition:

$$H = enc(X)$$

$$Y = dec(enc(X))$$

$$Y = dec(H)$$

where *enc* and *dec* are the functions applied by the encoder and the decoder, respectively.

Using an auto-encoder where the hidden layer is the same or larger size as the input and output layers could cause the network to over-fit the data and learn the identity function. To prevent such a case, an auto-encoder must be augmented with some technique to avoid over-fitting (i.e., losing generalizability). The applied techniques must help the network find the optimum trade-off between bias and variance. In other words, the auto-encoder needs to be capable of reconstructing each specific input efficiently while also generalizing well enough to represent the key characteristics of the data as a whole. Some of the techniques that can be employed to prevent over-fitting while allowing the model to generalize are:

1. Employing regularization *(Regularized Auto-encoders)*

2. Using a network architecture that creates a hidden layer of lower dimensions compared to the input and output *(Under-complete Auto-encoders)*

Regularization in the context of auto-encoders has been implemented in different ways. Some of the regularized auto-encoders that have been employed extensively in the past include:

1. *Sparse* auto-encoders[93], where an additional sparsity penalty is imposed, forcing the network to deactivate several neurons of the hidden layer based on the input data. As a result, the active code generated is still lower in dimension even though its number of neurons is equal to or larger than those of the input/output.

2. *Contractive* auto-encoders[94], where the goal is to make the reconstruction process less sensitive to minor variations in the data. This is achieved by imposing a penalty that helps carve a representation that better captures the local directions of variations dictated by the data, corresponding to a lower-dimensional non-linear manifold while being more invariant to the vast majority of directions orthogonal to the manifold.

3. *Denoising* auto-encoders[95], where the network is given a corrupted input and is trained to reconstruct the uncorrupted input. During this process, the network learns the important aspects of the data while filtering out the noise.

4. *Variational* auto-encoders[96] are different from other auto-encoders, providing a statistical manner to describe the samples of the data set in latent space. Therefore, the encoder outputs a probability distribution in the code instead of a single value in a variational auto-encoder.

For the under-complete auto-encoders, the network encodes data in a lower dimension by transitioning from layers with a high to a low number of neurons. The reverse sequence of layers is then applied to recreate the input layer on the output. Throughout this sequence of layers, the model has to maintain its ability to reconstruct an accurate copy of the input. To achieve that, it has to learn and store information that better characterizes the data in a smaller set of neurons. Thus, the code learned in this process comprises a compressed data representation.

If an under-complete auto-encoder consists of a single hidden layer and only linear activation functions, it produces the same representations as Principal Component Analysis[97]. In other words, an auto-encoder generalizes PCA to non-linear data, one of its first applications in the literature. Other applications where auto-encoders have been applied extensively are Dimensionality Reduction[98], Information Retrieval[99], Anomaly Detection[100] and Image Processing[95].

Auto-encoders can either be implemented as Multi-Layer Perceptrons (MLP) [101] by setting the number of neurons per layer appropriately or as convolutional neural networks (CNN) [102]. An under-complete CNN auto-encoder is very similar to an under-complete MLP auto-encoder. Its hidden layers' dimensionality gradually decreases to a point where the code is generated. Then the dimensionality gradually increases until it matches that of the input. However, in CNN, the dimensionality of each hidden layer is not explicitly set (like the number of neurons of each layer in MLPs). Instead, it is derived by the kernels' sizes, padding, strides, and pooling layers (if any).

6.2 DENOISING DRIFT CHAMBERS

During the denoising phase, removing as much noise as possible while retaining the valid hits is essential to avoid losing crucial information about the experiment. We show that using Convolutional Auto Encoders (CAE), it is possible to remove noise hits while retaining up to 94% of valid tracks for a beam current of 110 *nA*. For lower beam currents ($45-55$ *nA*), we get up to 98% efficiency. Studies on experimental conditions with increasing noise show that CAE performs better than conventional tracking algorithms in isolating hits belonging to tracks.

6.2.1 Method

Data Representation

Before experimenting with machine learning for our needs, we must define a representation for the detector data. As described in Section 6.1, each sector of the CLAS12 detector has three regions, each with two super-layers that consist of six layers of 112 hexagonal cells, for a total of 4032 ($3 \times 2 \times 6 \times 112$) cells. Each cell reports a value of 1 if a hit was detected or 0 otherwise. For our initial attempt with Multilayer Perceptron Autoencoders, each data point consisted of these 4032 values as features without pre-processing. For the CAE network, we reshaped the data point features into a 36×112 structure that better represents the spatial locality of the cells. This representation casts our problem to an image-denoising application where convolutional autoencoders have been shown to perform well [95]. Specifically, the data generated by the detector were treated as a 2D black and white image, where a black pixel (value 1) indicates a sensor activation and a white pixel (value 0) implies no sensor activation. For training the neural network, we used experimental data. The network input comprised an image containing all hits in one sector and an output image created from hits that belong to the reconstructed track by conventional tracking algorithms. The training and validation sample had a mixture of events of one, two, or three tracks in the output sample.

Neural Network Architectures

As shown in Table 6, several different CAE architectures were considered under-complete. The set of models presented in this table are variations of models we found to work relatively well for our problem after more extensive experimentation with different neural network architectures. The models exhibit variations in the number of layers, the type of pooling layers, and the approach to modify the dimensionality of the hidden layers (e.g., through kernel strides versus pooling).

Model 0 is a regular CNN autoencoder. The encoder of model 0 consists of two convolutional layers, each followed by a max-pooling layer. The decoder, which follows just after the max-pooling layer, consists of two convolutional layers followed by two upsampling layers (upsampling layers perform the reverse operation of pooling layers, they increase the size of their input by duplicating rows/columns). The shrinking in dimensions, in this case, is performed by the max-pooling layers, while the convolution kernels use same padding to keep the dimensionality of their outputs intact. Models 0a - 0f follow similar architectures with variations, including stacked convolutional layers (0a-0f) instead of single (i.e., two back-to-back convolutional layers without any pooling layer between them), average pooling instead of max pooling (0c), and different numbers of layers (0e). Models 1 & 2 decrease the dimensions of the hidden layers by using only the convolutional layers. They achieve that by employing strides with a dimensionality larger than 1x1. The upscaling (an increase of the input size) performed by the upsampling layers in the previous models is achieved here by deconvolution layers (deconvolution kernels perform a transpose convolution operation, the specifics about this operation are outside the scope of this work).

All models presented use Rectified Linear Unit [89] activation function for all layers except the last layer where Sigmoid[103] activation is used. The sigmoid layer, in the end, restricts the network outputs in the range of [0,1]. To assign a pixel to a class of black or white, we initially set a cut-off threshold at 0.50. In section 6.2.3, we experiment with different values for this threshold and present its impact on our results. We use Nesterov momentum Adam (Nadam) [104] as the optimizer and binary cross-entropy as the loss function for all models.

The architecture of one of the models (0b) is shown in Figure 69. The input to the network is a single image of dimension 36x112. The model then creates feature maps of dimensions 36x112 and 18x56 and finally encodes the input in feature maps of dimensions 6x28. This down-sampling process is then reversed to produce an output of dimension 36x112 that represents the input data without the noise. Two consecutive convolution layers with 4x3 kernels are used before max pooling and up-sampling layers.

TABLE 6

Architectures of the tested machine learning models. “C#” stands for “Convolution 2D with # feature maps”, “DC#” stands for “Deconvolution 2D with # feature maps”, “MP” stands for “Max Pooling 2D”, “AP” stands for “Average Pooling 2D”, and “US” stands for “Up-Sampling 2D”. For convolutions with stride (models 1,2), “k” stands for kernel size and “s” for stride size.

Model	Architecture
0	C48(4x6), MP(2,2), C48(4,6), MP(2,2),
	C48(4,6) ,US(2,2), C48(4,6), US(2,2), C1(4x6)
0a	C48(5x4), MP(2,2), 2*C48(4x3), MP(3,2)
	2*C48(4,3), US(3,2), 2*C48(5x4) , US(2,2), C1(5x4)
0b	2*C54(4x3), MP(2,2), 2*C54(4x3), MP(3,2)
	2*C54(4x3), US(3,2), 2*C54(4,3), US(2,2), C1(4x3)
0c	C48(5x4), AP(2,2), C48(4x3), AP(3,2)
	C48(4,3), US(3,2), C48(5x4), US(2,2), C1(5x4)
0d	2*C54(4x3), MP(2,2), 2*C54(4x3), MP(3,2)
	2*C54(4x3), US(3,2), 2*C54(4,3), US(2,2), 2*C54(4x3), C1(4x3)
0e	2*C128(4x3), MP(3,2), 2*C128(4x3), US(3,2)
	2*C128(4x3), C1(4x3)
0f	2*C28(4x3), MP(2,2), 2*C28(3x3), MP(3,2)
	2*C28(3x3), US(3,2), 2*C28(3x3), US(2,2), 2*C28(4x3), C1(4x3)
1	C64(k:6x6,s:6x1) , C64(k:2x2,s:2x1), DC64(k:2x3,s:2x1), DC1(k:6x6,s:6x1)
2	C64(k:3x3,s:3x1), C64(k:2x2,s:2x1), MP(1,2), C64(k:2x2)
	US(1,2), DC64(k:2x2,s:2x1), DC1(k:3x3,s:3x1)

Data Post-processing

The denoising operation may remove some valid hits, misidentifying them as noise, which can cause small gaps in the particle tracks; to mitigate this issue, a simple hit reconstruction algorithm can be employed. The algorithm iterates each hit present in the denoised data and marks any hits within a radius of one sensor around it in the raw(noisy) data. All marked hits are then added to the denoised data. This process recovers many missing hits but may also introduce additional noise. However, its improved efficiency far outweighs the introduced noise’s impact. We plan to

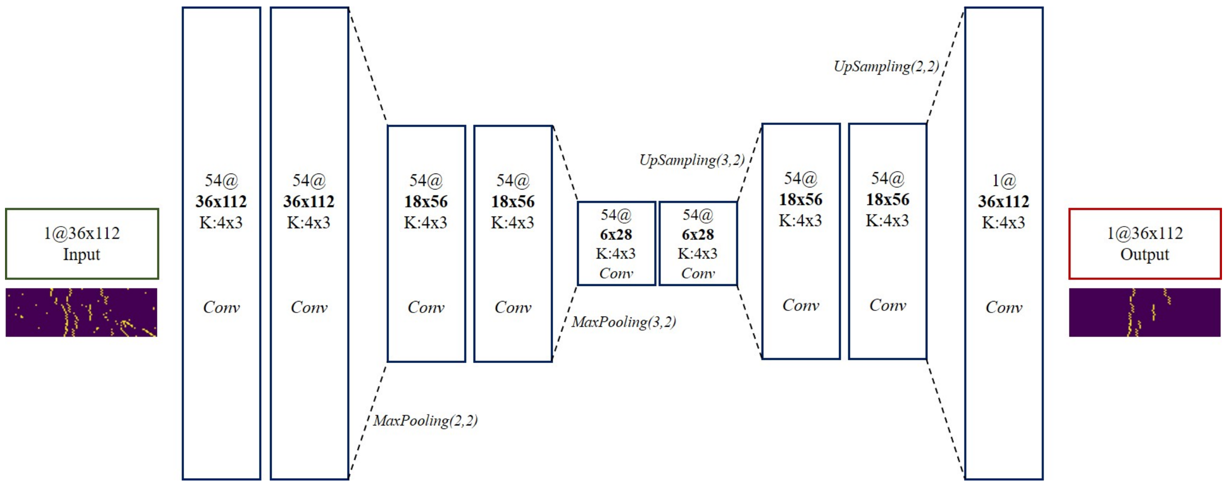


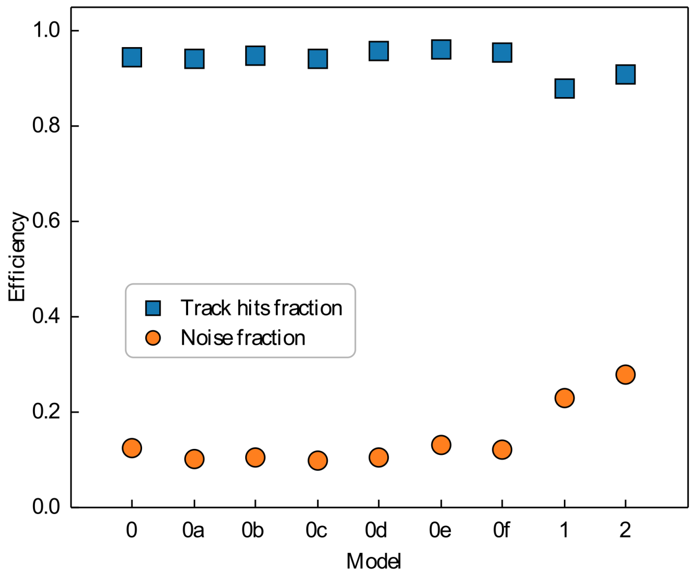
Fig. 69. Schematic representation of model 0b from Table 6.

introduce such a step in the future; however, in this work, all results presented do not include it. The reason for this is that timing cuts are necessary to reinstate a hit, and at the hit-based level here, we do not have access to the time of the hit.

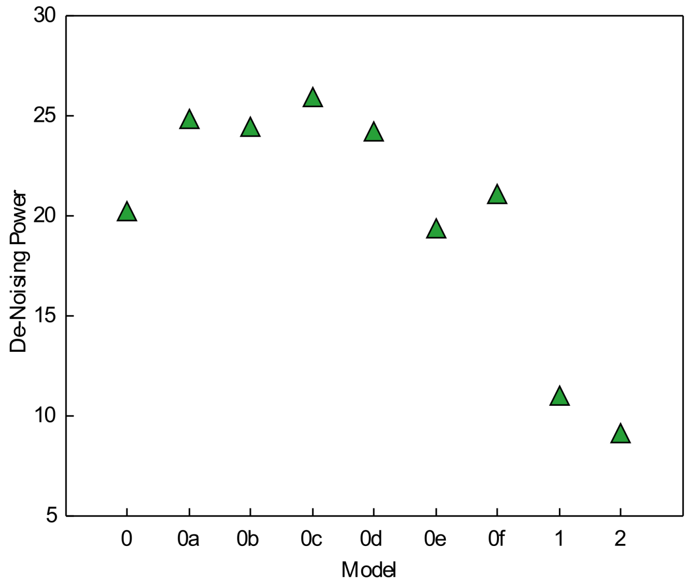
6.2.2 Model Comparison Results

This section presents the training and testing process and performance results of the machine learning models described in section 6.2.1. For each model shown in Table 6, we experimented using actual data generated by the CLAS12 drift chambers running on nominal conditions (i.e., 45nA of beam current). All models were trained on the same dataset of 60000 samples, consisting of events with single or multiple tracks. Training data used is from experimental reconstructed data, where the input sample is simply an image of all hits in the drift chamber sector, and the output image is all hits belonging to track (or tracks) that were reconstructed by conventional tracking algorithm (after denoising the hits with standard SNR algorithm). Testing results presented are gathered by running the generated models on a different dataset of 60000 samples.

Figure 70 shows the evaluation results for all nine models. The distribution of hit reconstruction efficiency (blue rectangles) presents the fraction of valid hits retained in the inferred image after applying the denoising CAE over those belonging to a track in the raw input (i.e., it shows the effectiveness of each model in retaining hits belonging to a track). The noise level (orange circles, left panel) is the hits in the reconstructed image that do not belong to a track as a fraction of the hits belonging to a track. In the original data sample, the average noise level is 250%. Applying the denoiser reduced the noise level to 10%-27%,10 to 25 times less noise. Table 7 presents more



(a)



(b)

Fig. 70. Model efficiency comparison. (a) The efficiency of track trajectory hits reconstruction (blue squares) and the remaining noise fraction after denoising (orange circles) are plotted as a function of the model. (b) The noise reduction power (green triangle) is the fraction of noise hits in the raw sample divided by the fraction in the reconstructed image.

TABLE 7

Evaluation results for all models' denoise efficiency. The noise level in the first four rows is the hits in the denoised image that do not belong to a track as a fraction of the hits belonging to a track. The last row presents the percentage of noise that the autoencoder models removed.

Metric (%)	0	0a	0b	0c	0d	0e	0f	1	2
Noise Mean Before Denoising	252	252	252	252	252	252	252	252	252
Noise RMS Before Denoising	163	163	163	163	163	163	163	163	163
Noise Mean After Denoising	12	10.1	10.3	9.9	10.4	13	12	23	27.5
Noise RMS After Denoising	24	22	21	21	22	25	23	30	32
Noise Removed After Denoising	95	96	96	96.1	95.7	94.7	95.2	90.9	89

detailed performance results of the models with respect to noise removal. Based on these results, we define the noise reduction power metric as the noise ratio before and after applying denoising. Figure 70 (b) (green triangles) shows the denoising power as a function of the model.

These results show that CAEs can accurately denoise and reconstruct given input, including when multiple tracks are in a single input. The best model, considering a combination of low noise fraction, high track hits fraction, and high denoising power (0b, see Figure 69), achieved a mean valid hit detection accuracy of over 95.5% with the mean amount of noise not exceeding 11.05%. 0b achieved very similar denoising efficiency to 0a and 0c; however, it performs better in track hits accuracy, so we chose it as our best model. We should note, though, that since the performance of the models is very close to each other, the ranking can change between different training sessions due to the randomness introduced from the learning process. Models 1 & 2 produced the worst results, which could be attributed to the fact that whole input columns are skipped when using a stride of 2. Since our features are primarily presented in the vertical dimension, many could be lost. The training process for the best model requires approximately 17 minutes to complete on our hardware setup using the dataset above 60000 samples. Denoising a single event requires, on average, 250 μs in our experimental setup; however, when running as part of the reconstruction process on JLab's computing facility, which only consists of CPUs, it takes approximately 90ms per event. The tracking code takes 520 ms per event (but this is background dependent); however, we noticed that denoised files run faster, meaning we will gain some of the lost time back from 90 ms.

6.2.3 Systematic Studies

Luminosity Studies

In section 6.2.2, we showed that CAEs efficiently denoise data from CLAS12 drift chambers, removing up to 96% of the initial background hits while retaining more than 95% of track-related ones. Next, we will study how increased noise (a by-product of high luminosity) can affect the accuracy of our denoising auto-encoder. We used experimental data with 45 *nA* to train the best network we developed (model 0b), and all our studies are performed using this model. Higher luminosity testing samples were produced using a standard background merging software [105] developed for CLAS12. The generation of these testing samples is performed in the following steps:

1. An initial set of events is taken from low luminosity (low beam current) experimental data where the noise level is minimal.

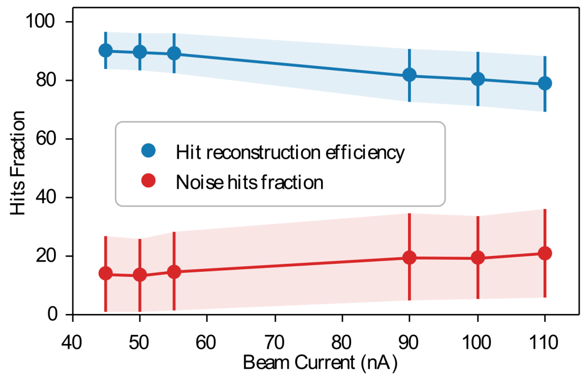
2. The low luminosity run is then merged with the background generated from experimental data on different beam current runs.

As a result we produced data samples corresponding to 45 *nA*, 50 *nA*, 55 *nA*, 90 *nA*, 100 *nA* and 110 *nA* for our studies.

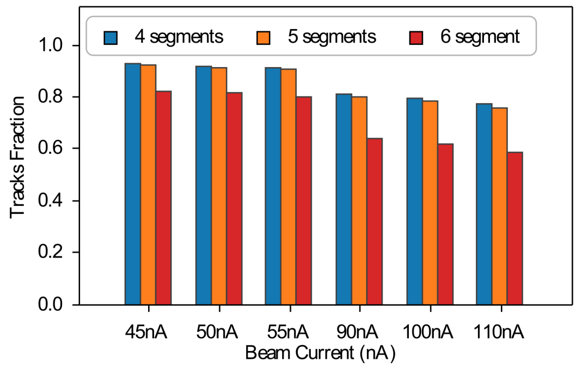
The testing samples were then analyzed with CAE to extract track hits reconstruction efficiency and background hits removal efficiency.

The results are shown in Figure 71. The blue dots of the figure on the left panel depict the fraction of valid hits reconstructed as a function of beam current. The red dots show the fraction of noise hits in the denoised data as a function of beam current. As shown in the figure, network performance drops slowly with the luminosity resulting in average efficiency of about 76% for 110 *nA* beam current. Samples for different luminosity can be seen in Figure 72. One sample from each luminosity (background) setting is presented in each row, along with the clean event (the middle column) that represents only hits that belong to a reconstructed track and the auto-encoder reconstructed image (right column). As seen in the example, there are a few traces of background still present in the reconstructed image; however, these will be eliminated by the tracking algorithm during track candidate selection. It is worth noting that more than 90% of the initial background is removed by the neural network, simplifying the clustering process significantly.

Even though hit reconstruction efficiency indicates the reconstruction efficiency of our method, it does not fully describe its capability to reconstruct full tracks. To quantify its track reconstruction



(a)



(b)

Fig. 71. Reconstruction performance. (a) Reconstructed hits fraction for real signal and background hits as a function of initial beam current. (b) The fraction of tracks reconstructed with 4,5 and 6 segments fully recovered from the denoised data sample.

efficiency, we must measure the network's ability to reconstruct enough hits leading to full track reconstruction. In the reconstruction procedure of CLAS12, a track can be fully reconstructed if segments of the track are identified in any five or four consecutive super-layers. A separate neural network can predict the missing segment positions based on the existing 5 (or 4) segments and recover from the raw input data. This procedure is detailed in [106] and [107].

Given that tracks can be recovered from incomplete segments reconstructed, we will determine the fraction of events that have retained 4, 5, or 6 segments by the CAE to evaluate its track reconstruction efficiency. Before that, we should note that the segment recovery efficiency is also subject to other considerations. Segments ideally consist of hits in all six layers of a super-layer. However, suppose hits in 2 or more layers are recovered. In that case, there is enough information about the position and direction (or angle relative to wire planes) of the segment to recover the rest of the hits from raw hits data. Considering these, we analyzed the output of the CAE to measure how many segments are reconstructed for each event. The results are shown in Figure 71 (right), where track reconstruction efficiency is presented as a function of beam current. Finally, tracks where at least four of their segments are recovered, can be fully reconstructed, resulting in a track reconstruction efficiency of more than 75% under running conditions of 110 nA.

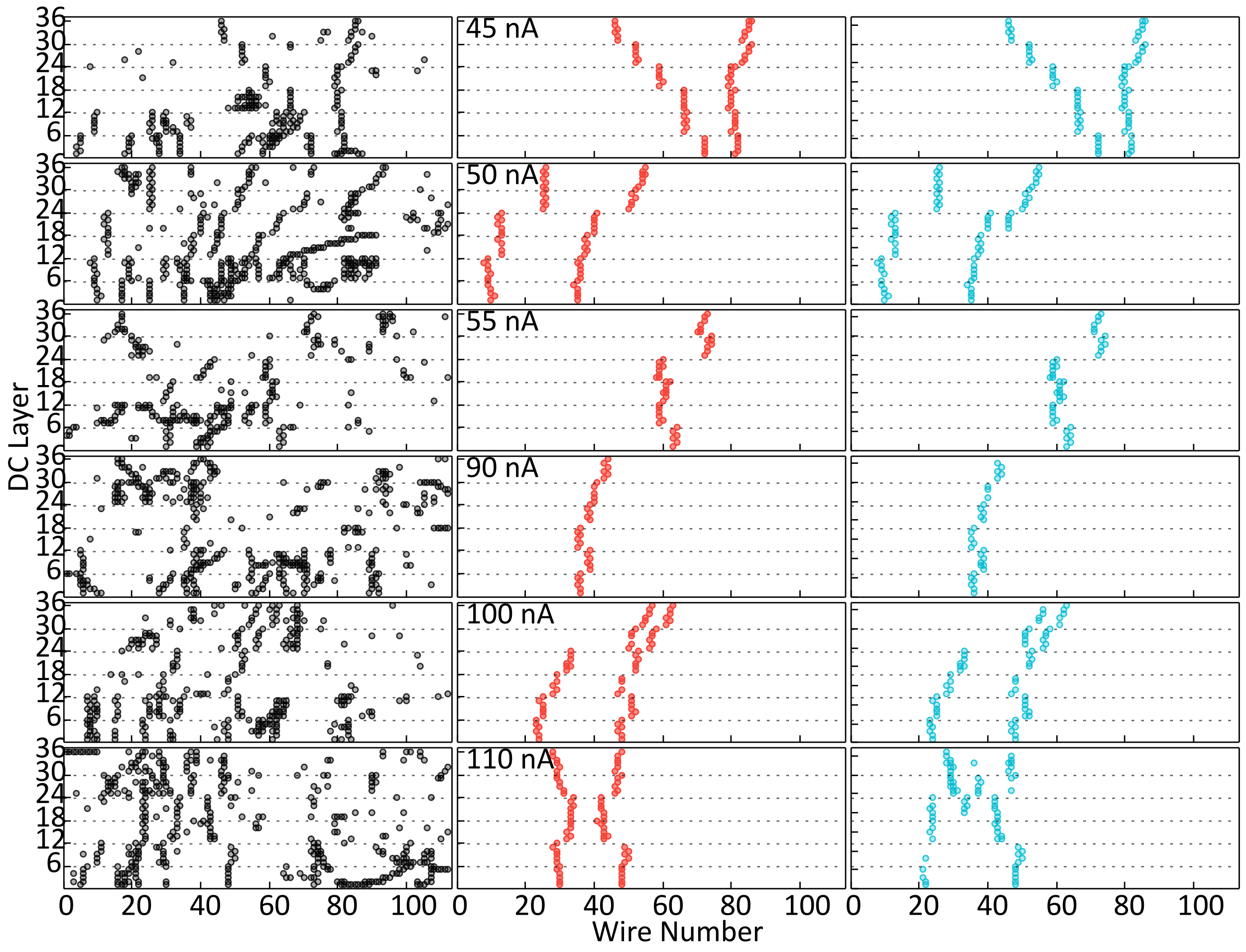


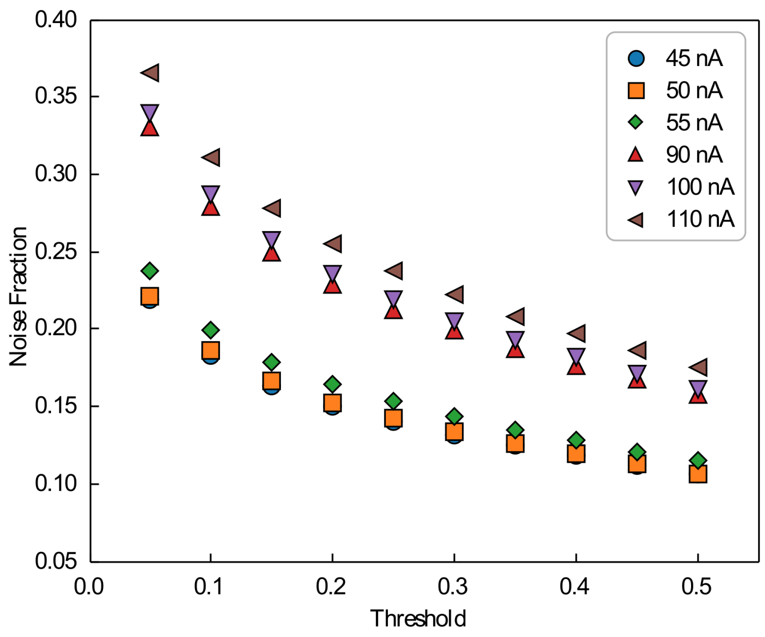
Fig. 72. Examples of the network applying denoising. The raw data are presented on the left column, the ground truth image in the middle, and the reconstructed image on the right. The rows represent one example from each background setting corresponding to 45, 50, 55, 90, 100, and 110 nA, respectively. Dashed lines represent the boundaries of the drift chamber's super-layers.

Reconstruction Threshold Studies

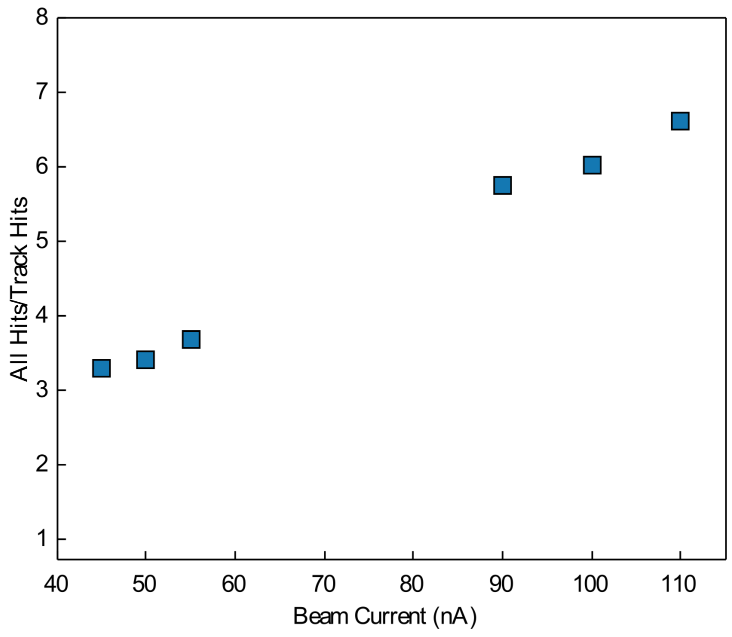
As mentioned in section 6.2.1, we used a cut-off threshold of 0.5 to assign each of the output values of the CAE to a class of no-hit detection (values < 0.5) or hit detection (values ≥ 0.5).

The following studies investigate whether modifying this threshold improves denoising and tracks reconstruction efficiency. We rerun our luminosity studies with varying cut-off thresholds ranging from 0.05 to 0.5 with a step of 0.05. The resulting distributions can be seen in Figure 73a, where the noise level is presented as a function of the cut-off threshold for different beam currents.

As expected, lowering the cut-off threshold for pixel reconstruction does increase the noise level, though the noise is still significantly lower than in the original noisy data. For comparison, Figure 73b shows the ratio of all hits to those that are part of a track for the initial noisy data under high luminosity. For 110 nA, the percentage of noise in the raw data sample is about 6.5, while using the CAE with the lowest threshold of 0.05 yields a noise level of 0.37. Even though this change in the threshold affects the noise level slightly, it significantly improves hit and track reconstruction efficiency. Figure 74a shows the hit efficiency improvement as exhibited by lowering the cut-off threshold. For the highest beam current of 110 *nA*, the hit efficiency improves from 0.76 to 0.89. The same analysis measures the final track reconstruction efficiency after denoising. The results are shown in Figure 74b. The same trend as with hit reconstruction efficiency is observed in track reconstruction efficiency. The improvement is even more significant in this case, with the track efficiency for 110 *nA* increasing from 76% to 94%. Figure 75 presents the Receiver Operating Characteristic Curve (ROC)[108] curve produced by running model 0b on the 110 *nA* dataset where true positive rate corresponds to the ratio of hits correctly identified and false positive rate to the noise that was wrongly maintained in the output. The curve confirms that our model accurately classifies hits and noise. The following section shows that the results obtained for high beam currents are significantly better than the standard tracking procedure for similar data.

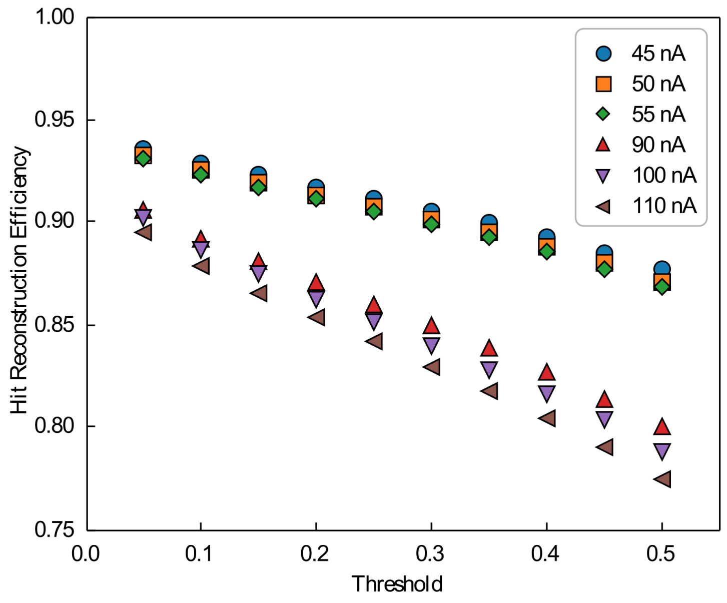


(a)

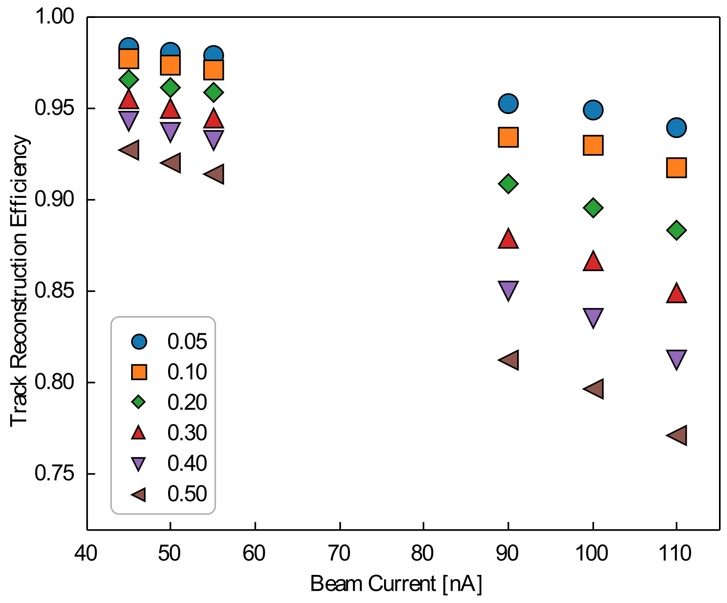


(b)

Fig. 73. (a) Noise reduction efficiency for all beam currents as a function of cut-off threshold. (b) The ratio of all hits in the event to the hits belonging to a reconstructed track as a function of beam current before applying denoising.



(a)



(b)

Fig. 74. Reconstruction efficiency. (a) Hit reconstruction efficiency for all beam current data samples as a function of cut-off threshold, (b) track segment reconstruction efficiency as a function of beam current for different cut-off thresholds.

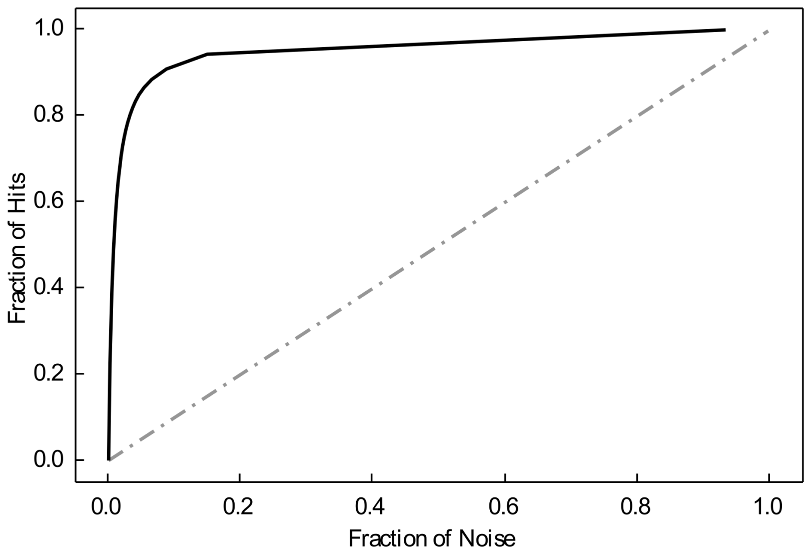


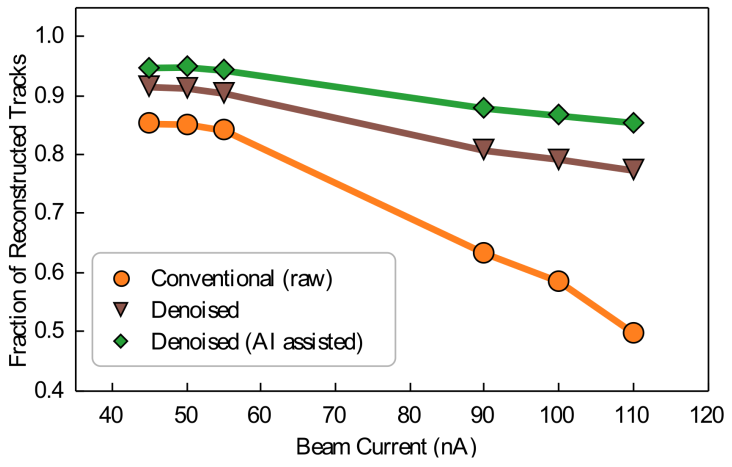
Fig. 75. ROC curve for model 0b running on the 110 *nA* dataset.

6.2.4 Studies with Experimental Data

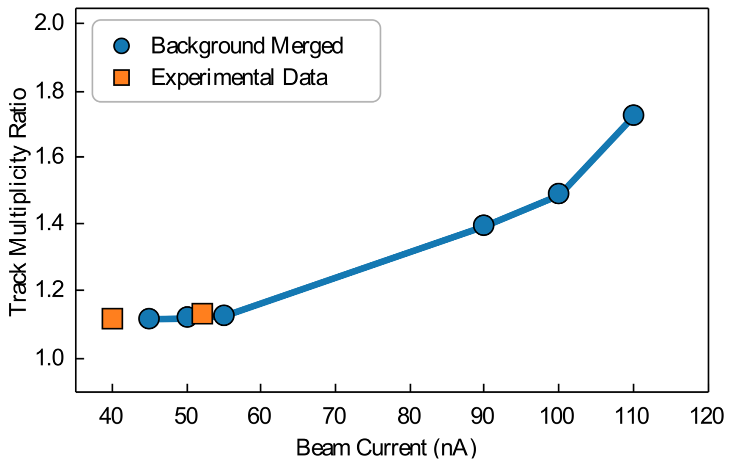
Our systematic studies established that using a threshold of 0.05 for hit reconstruction efficiently removes a significant fraction of noise hits while significantly improving the track hits reconstruction efficiency. The next step of our study was to use the presented CAE in the data reconstruction workflow of the CLAS12 detector. We used our CAE as a stand-alone module to denoise raw data from drift chambers before passing them to the reconstruction process. We used two data sets to measure the effect of denoising on the reconstruction software. One data set was created by merging background to experimental data with a beam current of 5 *nA* and producing six data sets with experimental conditions corresponding to 45 *nA*, 50 *nA*, 55 *nA*, 90 *nA*, 110 *nA* and 110 *nA*. The second data set was from experimental data with beam currents 40 *nA* and 50 *nA*. The CAE was used to produce a denoised data set for all data sets. Both data sets (original and denoised data) were then processed with CLAS12 reconstruction software.

The resulting files were analyzed event-by-event to measure the track reconstruction efficiency after denoising. We isolated tracks from the original 5 *nA* data (reconstructed with six superlayers) and examined whether the same tracks were reconstructed from the background merged file (raw and denoised version). Figure 76 (a) shows the fraction of tracks reconstructed from background-merged files for both data sets (raw and denoised) as a function of beam current. The three different tests represent reconstruction using the CLAS12 tracking algorithm on raw (unaltered) data (orange circles), data pre-processed with the denoising CAE (brown triangles), and finally, results from using the CLAS12 reconstruction, augmented with AI-assisted track identification software [107] from Section 6.3, on denoised. As shown in the figure, passing data through the denoising CAE significantly improves the fraction of tracks that can be recovered in high background conditions compared to conventional methods. The main reason for this improvement is that in high background conditions, the traditional algorithm fails to isolate clusters in each superlayer to have all track candidates considered for track composition. Using CAE helps to reduce noise, thus enabling the standard tracking algorithm to reconstruct clusters more efficiently and, as a result, reconstruct more tracks.

The improvements of the CAE-assisted tracking are shown in Figure 76 (b), which presents the ratio between tracks reconstructed with and without applying the CAE denoising step. The track ratio presented is tracks reconstructed by conventional algorithm only, without AI-assisted track candidate classification, to emphasize that with the traditional algorithm, the gain from denoising alone is very significant. As can be seen from the figure, CAE improves the track yield in normal experimental conditions (i.e., 40 *nA*-50*nA*) by about $12\% - 15\%$ while achieving a much more significant improvement at higher background settings (about 75% at 110 *nA*). The improvement



(a)



(b)

Fig. 76. Track reconstruction efficiency. (a) As a function of beam current for raw data (orange circles), denoised data (brown triangles), and denoised data running through AI-assisted track reconstruction [107] (green diamonds). (b) Track reconstruction gain with denoising algorithm, the ratio of reconstructed six super-layer tracks with CAE to conventional raw data reconstruction for different beam current settings for background merged files (blue circles) and real data (orange squares).

is also apparent in experimental data (without background merging), where we extract the ratio of reconstructed 6-super-layer tracks of CAE-denoised data over conventional raw reconstruction (Figure 76, b) for experimental data with 40 *nA* and 50 *nA*. The experimental data with different beam currents show similar (consistent) improvements in track multiplicity.

6.2.5 Discussion

In this work, we developed methods for denoising drift chamber data for the CLAS12 detector. The proposed Convolutional Auto-Encoders were trained on six-segment tracks, along with whole raw hits from experimental data, to remove noise hits and retain those belonging to a track. We developed this method as a stand-alone software that has to be run on raw data files before they are processed with standard CLAS12 reconstruction software. Applying this method as a pre-processing step presents some limitations in discovering its full potential in track reconstruction. One such restriction comes from the fact that not all pixels in each segment are reconstructed by CAE, leading to a potential loss of efficiency. CLAS12 tracking software relies on having at least four layers (out of six) activated in each super-layer to form a cluster. If the CAE preserves three hits out of six, this cluster will not be reconstructed by CLAS12's software because our stand-alone program removes the hits from the list of potential signals from the raw data set. To overcome this limitation, the CLAS12 tracking software has to be modified to consider suggested hits from CAE and then try to perform clustering by retrieving nearby hits from raw data; such a modification will improve the segment reconstruction efficiency even further.

The results presented are preliminary and do not include all the abovementioned improvements. However, after applying denoising to the data set, they already show significant improvement in the number of tracks reconstructed by the standard CLAS12 tracking code. In normal experimental conditions (at 45nA), we get a gain of $12-15\%$ in the number of tracks reconstructed, while in high luminosity conditions (above $100nA$), we observe improvements of more than 75%. These results have a significant impact on how experiments run. The length of an experiment (time allocated to run with the accelerator-provided beam) is determined by the integrated luminosity requirements needed to produce physics results. The running conditions, i.e., target length and beam current, are chosen to reconstruct tracks efficiently. As can be seen from our results, track reconstruction efficiency with our current (not optimized) implementation at 110 *nA* is higher than the standard reconstruction efficiency (without denoising) at current experimental running conditions ($45nA$ beam). By employing denoising for track reconstruction algorithms, one can run experiments in a much shorter time to achieve the exact statistical yield for the same physics outcome.

This project took about 4-5 months to complete; this includes finding the suitable Machine

Learning algorithm to apply (CNN) inspired by image denoising applications, gathering the training and testing datasets, experimenting with different architectures, activation functions, and error estimators, and finally experimenting with different thresholds. Finding the appropriate architecture was the most time-consuming process as it required manually adjusting the model parameters and then training and testing until satisfactory results were obtained. We intend to continue our efforts to improve our model further as it is integrated into JLab's data processing pipeline.

6.2.6 Summary

This work used Convolutional Auto Encoder neural networks to denoise raw data from CLAS12 drift chambers. The Convolutional Auto Encoder showed excellent performance in removing noise hits, reducing them by more than 90% while retaining 76% of the hits belonging to track trajectories. Further studies with threshold adjustment improved average track trajectory hit reconstruction to 94% with an insignificant increase in noise hits. The developed software was used to denoise several data sets, which then ran through CLAS12's standard reconstruction software. We compared the reconstructed tracks' yield to standard (non-denoised) data track yield, showing that denoising significantly improves track reconstruction efficiency for six-segment tracks. For high background conditions (such as for data $> 90nA$) the gains in track reconstruction are much higher, up to 1.8 times improvement for $110nA$. This software will be rigorously tested in the CLAS12 data processing environment in parallel with standard reconstruction packages and will be implemented in the data processing workflow to assist the tracking algorithm in high background running conditions.

6.3 PARTICLE TRACK CLASSIFICATION

In this section, we describe the development of four machine learning (ML) models that assist the tracking algorithm by identifying valid track candidates from the measurements in drift chambers. Several types of machine learning models were tested, including Convolutional Neural Networks (CNN), Multi-Layer Perceptrons (MLP), Extremely Randomized Trees (ERT), and Recurrent Neural Networks (RNN). As a result of this work, an MLP network classifier was implemented as part of the CLAS12 reconstruction software to provide the tracking code with recommended track candidates. The resulting software achieved an accuracy of greater than 99% and resulted in an end-to-end speedup of 35% compared to existing algorithms.

6.3.1 Data Selection and Performance Metrics

Data Selection

Due to inefficiencies that develop over time, not all six layers in a given super-layer have hits in each cluster. Therefore, we used different data representations (features) for each model. For the CNN, an image of size 36x112 was used, representing all the wires in one sector of the drift chamber since the inefficiencies get smoothed out by convolutional and pooling layers. For the ERT and MLP models, we used six-feature input for each track, representing the average wire number of the cluster in each super-layer. For the RNN, we pre-processed the data to produce a sequence of 36 numbers that represent the track trajectory. For this purpose, each cluster in all six super-layers was fitted with a linear function to determine its intercept and slope. Then, the signals in missing layers were completed with a pseudo hit with the wire number on the line representing the cluster. After this procedure, we had a track candidate with 36 input features, one for each wire in the drift chamber layers.

To generate the training datasets, we extract good and bad tracks from data events already processed by the conventional algorithm and write out their hit patterns. Figure 63 shows an example of one event where all hit clusters are presented along with the clusters that form a valid track.

The cluster combinations that form a real track are labeled as a positive sample in all representations. In contrast, those not identified as valid tracks are labeled negative samples. Since there are multiple negative samples per event alongside the positive one, we need to balance the training dataset between negative and positive samples. To achieve this, we only provide a single negative sample for each positive sample of an event. In the next section, we evaluate how different options for negative samples affect our results.

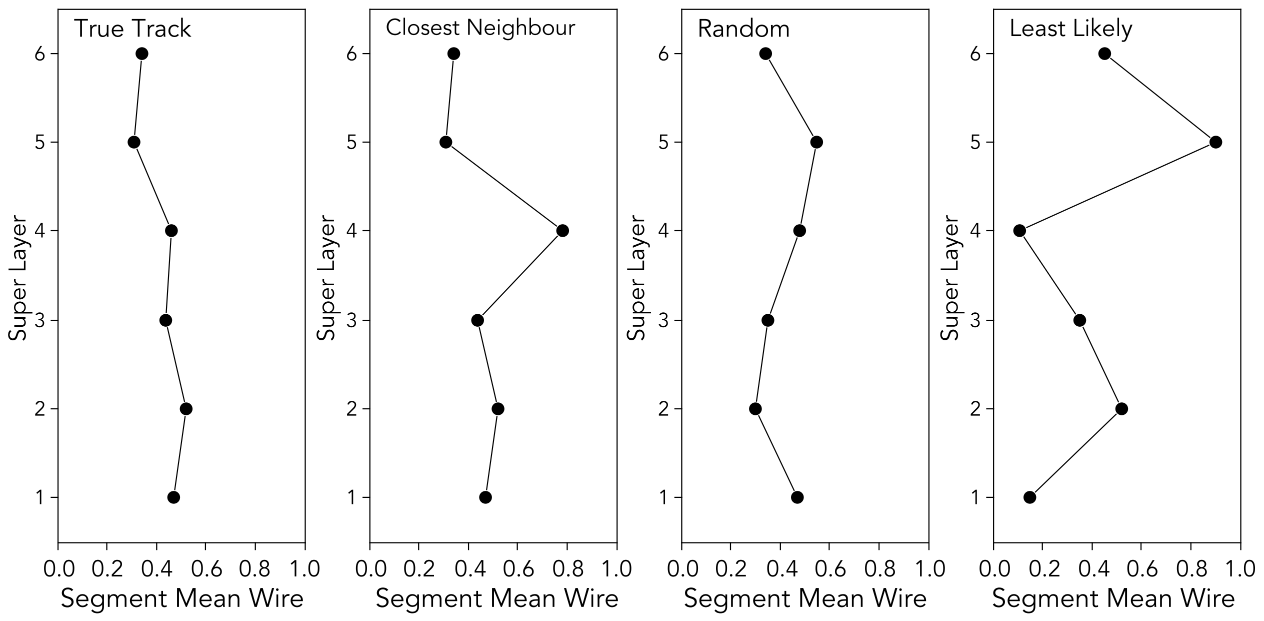


Fig. 77. Training data selection for the inference accuracy study. Examples of data samples for "Least likely", "Random" and "Closest Neighbor". (Wire numbers are normalized to 112)

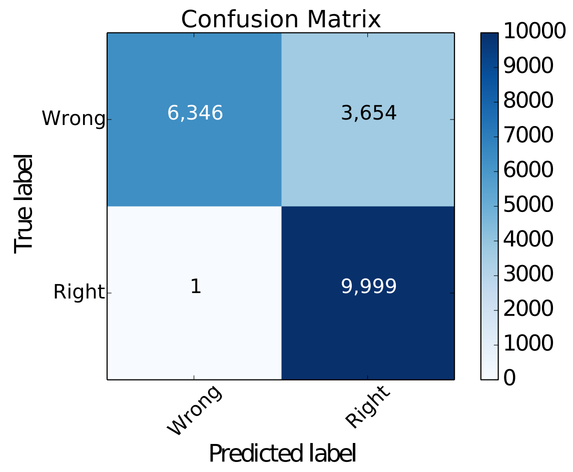
Training Data Tuning

We discovered that negative sample selection could affect the model accuracy when selecting training data. In each reconstructed event with a valid track, there are many track candidates that we have to choose as a false sample. In the training dataset, we balance the number of "true" and "false" samples to prevent the machine learning model from over-fitting to "false cases".

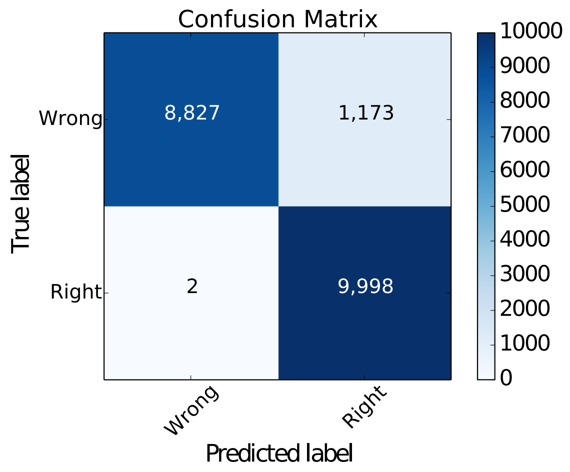
We create three training datasets (shown on Firgure 77) and test the network trained on different input data. The samples and choices for the "false" track candidate are shown on Figure 77 are the following:

- track candidate that looks least like the real track in that event
- track candidate randomly chosen from all combinations
- track candidate that is closest to the real track; most of the time, they differ only by one cluster

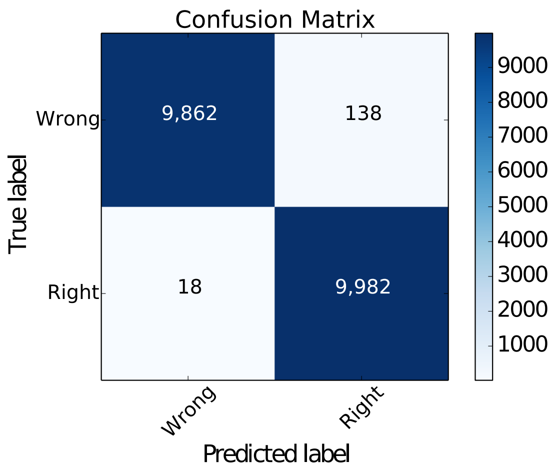
Three models are trained using three datasets, and then model evaluation is done on a subset of "closest neighbor" data.



(a)



(b)



(c)

Fig. 78. Confusion matrix (number of events) for different models. (a) trained with data set with least likely track candidates, (b) trained with a sample with randomly chosen “false” candidates, (c) trained with sample closest neighbor “false” candidates.

As can be seen from the confusion matrices in Figure 78, the model trained on random and "least likely" datasets can accurately predict the "true" track but fails in classifying the "false" track. Because the model has not shown a false candidate very similar to a true candidate, it cannot distinguish false candidates very similar to the true. The tests show that model accuracy depends on training sample composition. The best results are achieved using the "closest neighbor" training dataset. This method is used for all model evaluations and the final implementation of training data extraction from experimental data.

Performance Metrics

We devised and utilized several metrics besides the standard accuracy metrics to determine our models' accuracy. The need for these new metrics stems from the fact that the performance of the machine learning models should be evaluated in the context of event samples rather than the overall performance of the whole dataset. Moreover, it is crucial to identify the valid track in each sample; however, it is less harmful to misclassify an invalid track since it will be eliminated by the following stage of the tracking algorithm where Kalman-Filter fitting is applied. Of course, keeping false positive numbers low is essential since Kalman-Filter fitting is the slowest part of the reconstruction process. Also, examining only the model's accuracy does not give a good indication of its overall performance due to the heavily imbalanced amount of invalid tracks in the testing sample. For example, a model that classifies all tracks as invalid may achieve relatively high accuracy; however, it is useless because it cannot detect valid tracks.

Our custom metrics provide a better indication of the real performance of the machine learning methods for the needs of our application. The accuracy metrics consist of the following:

1. **A-det**: The ratio of samples where the valid particle track was correctly detected.

2. **A-det|conf**: The percentage of A-det for which there were invalid tracks confused (misidentified) as valid ones (false positives).

3. **A-det|high**: The percentage of A-det for which the valid particle track had the highest probability of being valid out of all tracks in a sample.

4. **A-notdet**: The ratio of samples where the valid track was not detected (false negatives); this metric was critical to minimize, as we do not want to miss valid particle tracks.

6.3.2 Models Description and Performance Evaluation

Evaluation Settings

The following evaluations were performed on Old Dominion University's Wahab High Performance Computing Cluster. The training set consists of 3.4 million tracks spread equally between valid and invalid tracks. The evaluation dataset consists of 14,760 events totaling 606,223 tracks, both generated from real data in CLAS12 after being classified using the conventional algorithm. Classifications for models outputting probabilities (ERT, MLP, CNN) are made around a threshold of 0.5; tracks with inferred probability higher or equal to that are considered valid. The ERT and MLP models were executed on a 40-core Intel Xeon Gold 6148 CPU @ 2.40GHz, utilizing the 6-feature data format and implemented using the scikit-learn library[109]. The CNN and RNN were executed on an NVIDIA Tesla V100 GPU. The CNN utilized the 36x112-feature data format while RNN the 36-feature format. Both networks were implemented in TensorFlow 2 [110] with Python 3 on a Linux Ubuntu machine.

Extremely Randomized Trees

Architecture Our application uses a model with three hundred estimators (decision trees), with no limit on the number of features to be considered when splitting. In addition, we use the information gain (entropy) split quality criterion. The rest of the parameters are kept at their default values. The input for the model is a 6-feature dataset.

Results The ERT model has 99.96% accuracy in identifying the valid particle track in a sample (A-det metric). In 40.72% of the samples where the valid track is identified, there are also some false positives (A-det|conf metric); however, in most cases ($> 97\%$ of the samples), the valid track is given the highest probability of being valid (A-det|high metric). Finally, there are almost no samples where the valid track is undetected (A-notdet metric). These results are compiled in Table 8. Table 9 shows the confusion matrix generated by evaluating the model on the testing dataset, indicating very few false negatives and an acceptable amount of false positives.

Multi-Layer Perceptron

Architecture The MLP used for our application consists of three hidden layers with sixty-four neurons each (see Figure 79). The optimizer chosen was *Adam* [111]. We used a batch size of thirty-two, and an adaptive learning rate, meaning that the learning rate decreases if there is no

TABLE 8

Results of the ERT model evaluation.

Metric	Result
A-det	99.96%
A-det|conf	40.72%
A-det|high	97.09%
A-notdet	0.04%
Time to Train	502 sec
Time to Predict/sample	306 μs

TABLE 9

The confusion matrix generated by evaluating the ERT model on the testing dataset.

	Predicted: Invalid	Predicted: Valid
Actual: Invalid	560,019	31,444
Actual: Valid	6	14,754

training loss decrease in two consecutive epochs. The rest of the parameters were kept as their default values.

Results The MLP model achieves 99.95% accuracy in identifying the valid particle track in a sample (A-det metric). In 38.32% of the samples where the valid track is identified, there are also some false positives (A-det|conf metric). In 92.92% of the samples, the valid track is given the highest probability of being valid (A-det|high metric). Finally, the valid track is not detected in only 0.05% of samples (A-notdet metric). These results are compiled in Table 10. Table 11 shows the confusion matrix generated by the results from evaluating the model with the testing dataset.

Convolutional Neural Network

Architecture The convolutional neural network consists of three convolutional layers of 32, 64, and 128 filters, respectively, with a 3x3 kernel size, each followed by a 2x2 max-pooling layer with same padding and dropout of 0.25, 0.25, and 0.4 (see Fig. 80). Dropout[112] is a technique

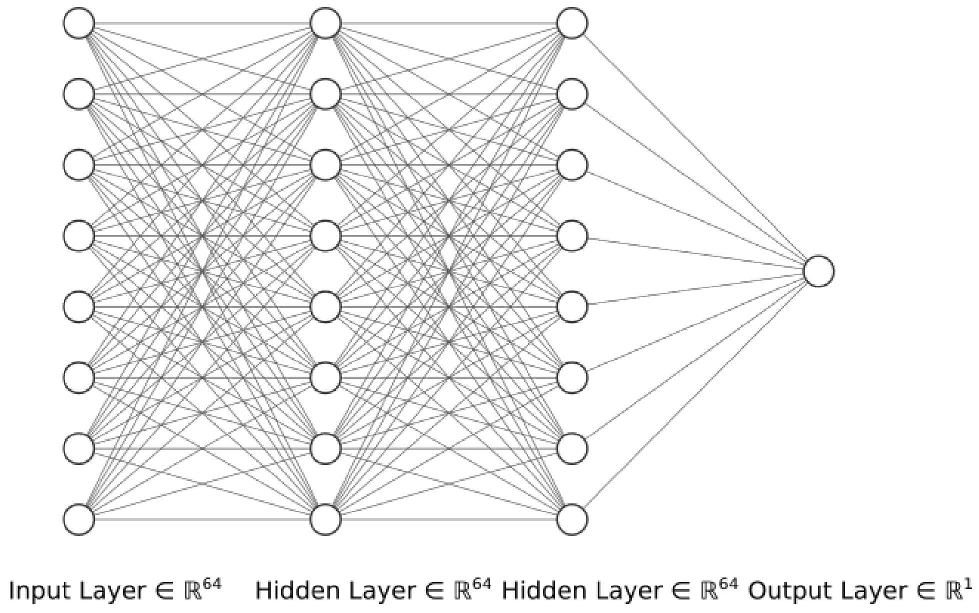

Input Layer $\in \mathbb{R}^{64}$
Hidden Layer $\in \mathbb{R}^{64}$
Hidden Layer $\in \mathbb{R}^{64}$
Output Layer $\in \mathbb{R}^{1}$

Fig. 79. Architecture of the Multi-Layer Perceptron network.

where a percentage of the available neurons of the network are randomly selected and deactivated during training in an attempt to prevent over-fitting. Two dense layers of 128 and two neurons follow the convolutional layers with a dropout of 0.3 between them. The activation function for all layers except the last one is Leaky ReLU[113], while softmax is used for the last layer. Adam was used as the optimizer and categorical cross-entropy as the loss function. We trained the model for twenty epochs using a batch size of 32.

Results The CNN model achieves good results, with 99.96% accuracy in identifying the valid particle track in a sample (A-det metric). In approximately 53% of the samples where the valid track is identified, there are also some false positives (A-det|conf metric). In 90.2% of the samples, the valid track is given the highest probability (A-det|high metric). Finally, the valid track is not detected in 0.04% of the samples (A-notdet metric). These results are compiled in Table 12. Table 13 shows the confusion matrix generated by the results from evaluating the model with the testing dataset.

Recurrent Neural Network

Architecture We use two stacked GRU layers, the first containing 40 units while the second containing 240 units (Figure 81). The input to the model is a vector of features, with each feature

TABLE 10

Results of the MLP model evaluation.

Metric	Result
A-det	99.95%
A-det|conf	38.32%
A-det|high	92.92%
A-notdet	0.05%
Time to Train	4 hours
Time to Predict/sample	120 μs

TABLE 11

The confusion matrix generated by evaluating the MLP model with the testing dataset.

	Predicted: Invalid	Predicted: Valid
Actual: Invalid	561,994	29,469
Actual: Valid	7	14,753

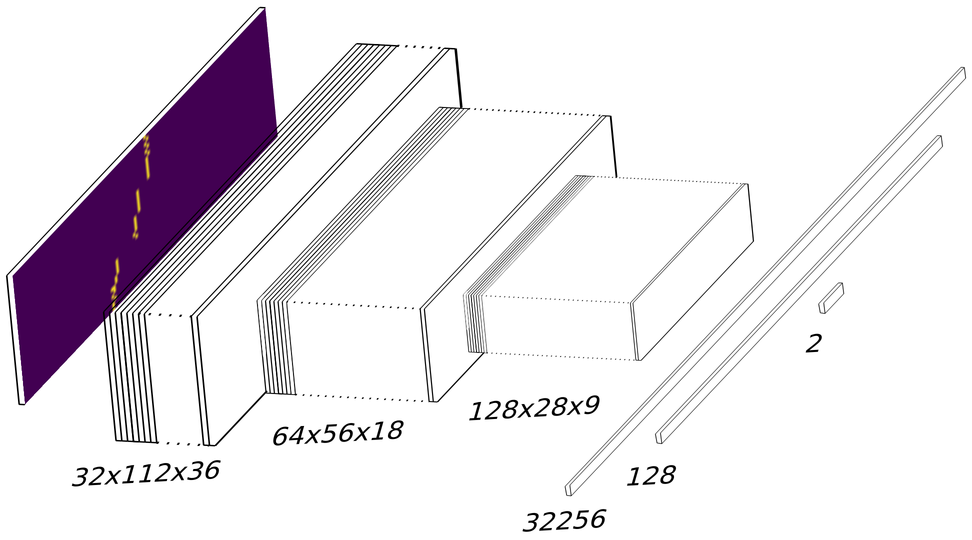


Fig. 80. Architecture of the CNN network.

TABLE 12

Results of the CNN model evaluation.

Metric	Result
A-det	99.96%
A-det$\|$conf	52.99%
A-det$\|$high	90.22%
A-notdet	0.04%
Time to Train	8 hours
Time to Predict/sample	1.2 ms

TABLE 13

The confusion matrix generated by evaluating the CNN model with the testing dataset.

	Predicted: Invalid	Predicted: Valid
Actual: Invalid	552,808	38,655
Actual: Valid	6	14,754

occupying one time step. As for the optimizer, we used *RMSprop* [114].

Results The RNN achieves an average mean absolute error (MAE) of about 1.57. In the context of this problem, the model's predicted particle trajectories were, on average, about 1.57 sensors away from the actual particle trajectories. Figures 82 and 83 show examples of actual particle tracks overlapped with the predicted portions of the particle tracks from the RNN. For the classification process, we used Mean Absolute Error (MAE) as the metric for spatial distance and explored different values as the maximum distance.

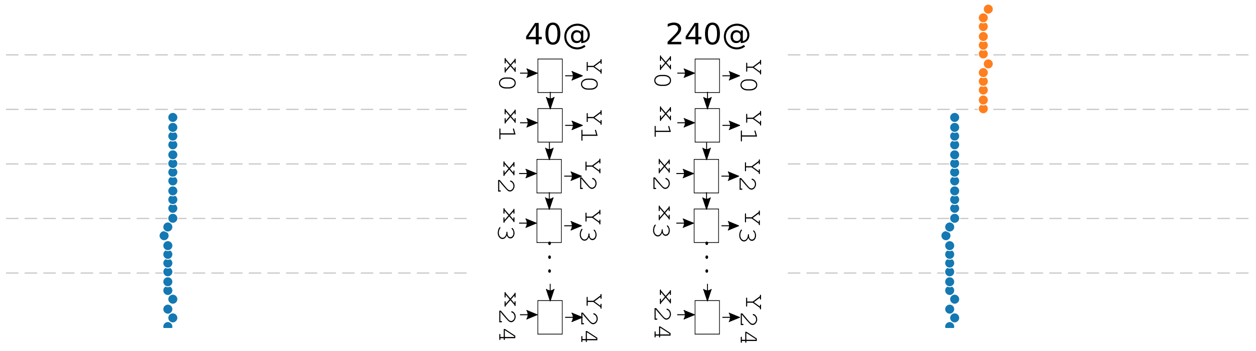


Fig. 81. Architecture of the recurrent neural network.

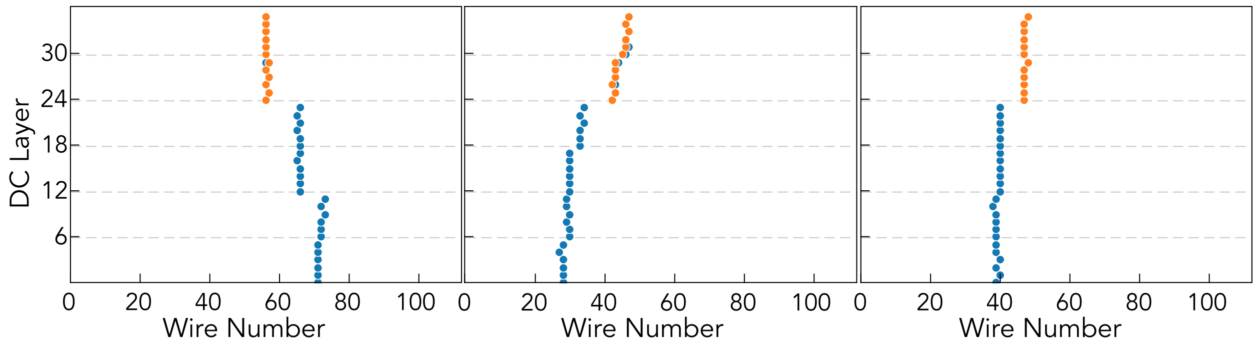


Fig. 82. Valid particle tracks (blue) and the predictions of the RNN for part of them (orange). If the RNN correctly predicted the track, then the blue and orange overlap. The small spatial distance between the predicted and actual portions of the tracks means that the actual tracks are likely valid.

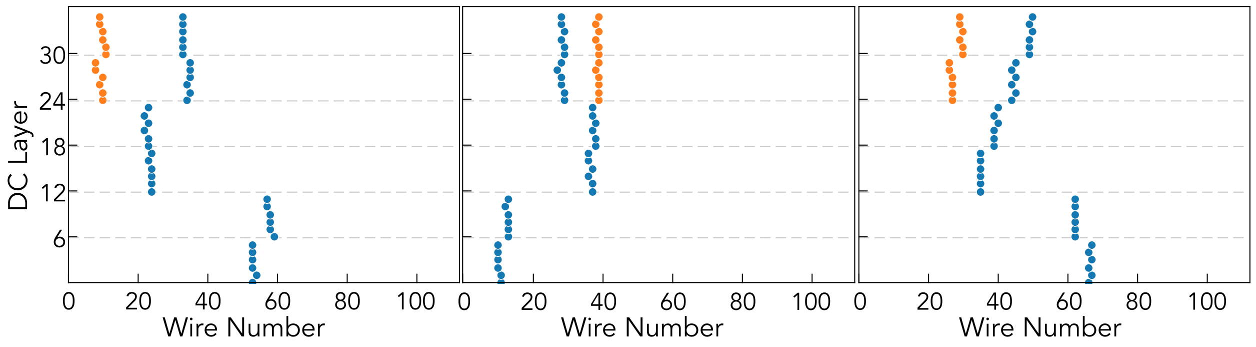


Fig. 83. Invalid particle tracks (blue) and the prediction of the RNN for part of them (orange). The larger spatial distance between the predicted and actual portions of the tracks means that the actual tracks are likely invalid.

TABLE 14

Results of the RNN model evaluation for different maximum distances.

Metric	Max Dist: 2	Max Dist: 3	Max Dist: 4
A-det (%)	91.98	97.58	99.0
A-det-conf(%)	69.65	83.32	90.65
A-det|high (%)	71.74	73.91	74.17
A-notdet (%)	8.02	2.41	1.0

Tables 14 & 15 show the performance of the RNN for the different maximum distances. As expected, a larger maximum distance helps to identify more valid tracks but increases the number of invalid tracks misclassified as valid. On the other hand, a smaller distance decreases the number of tracks falsely identified as valid and increases the number of valid tracks missed by the algorithm. Since no probabilities are inferred from the network to classify as valid or invalid, the value we used to compute the A-det|conf metric is the distance between the inferred and the actual track candidate (the smaller, the better). The performance of the RNN shows that this approach can be promising; however, the three other machine-learning networks outperform it.

6.3.3 Performance Summary

We have presented four machine learning models applied to perform track classification for

TABLE 15
The confusion matrix generated by evaluating the RNN model with the testing dataset. The true positive and false negative percentages are presented as a fraction of the valid tracks in the dataset. Accordingly, the percentages of true and false negatives are presented as a fraction of the invalid tracks.

	Max Dist: 2	Max Dist: 3	Max Dist: 4
True Positive (%)	91.98	97.58	99.0
False Negative (%)	8.02	2.41	1.0
True Negative (%)	92.45	88.35	84.0
False Positive (%)	7.55	11.65	16.0

TABLE 16
Summary of the results for the four models for particle trajectory classification.

Model Type	A-det (%)	A-det|conf (%)	A-det|high (%)	A-notdet (%)	Training Time	Inference Time/sample
ERT	99.96	40.72	97.09	0.04	506 sec	306 μs
MLP	99.95	38.92	92.92	0.04	4 hours	120 μs
CNN	99.96	52.99	90.22	0.04	8 hours	1.2 *ms*
RNN	99.0	90.65	74.17	1.0	3 hours	343 μs

CLAS12 drift chambers, along with a brief introduction to each method. In this section, we summarize the results of all four methods in tables 16, 17.

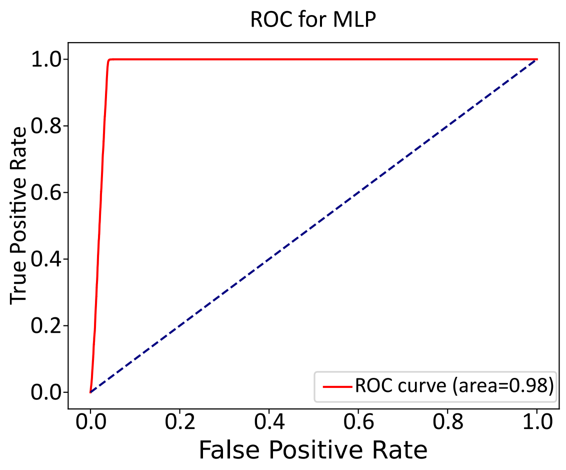
We can see that all methods accurately identify the valid track with over 99% success. ERT performs the best in accuracy, while MLP comes a close second; however, it performs 2.5 times better in inference speed. CNN gets the third place in both accuracy and inference time; an essential factor for that is that it utilizes a much larger data format (4032 features vs. six vs. 36). The RNN falls quite behind in accuracy, misclassifying about 1% of the valid tracks versus 0.04% of the rest of the models. Moreover, the percentage of events containing false classifications amount to 90.65%, more than double what the ERT and MLP models achieved. Out of this study, we decided to implement the MLP algorithm in the track reconstruction process of CLAS12 due to its high speed and high accuracy in detecting the valid track. Figure 84 presents the ROC curves of the

TABLE 17

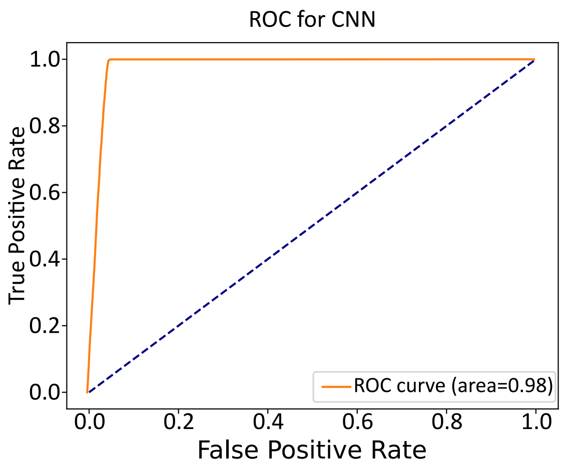
Confusion matrix for each machine learning model on the testing dataset. The true positive and false negative percentages are presented as a fraction of the valid tracks in the dataset. Accordingly, true and false negative percentages are presented as a fraction of the invalid tracks.

Model	True Positive	False Negative	True Negative	False Positive
Type	(%)	(%)	(%)	(%)
ERT	99.69	0.04	94.68	5.31
MLP	99.95	0.05	95.02	4.98
CNN	99.96	0.04	93.46	6.53
RNN	99.0	1.0	84.0	16.0

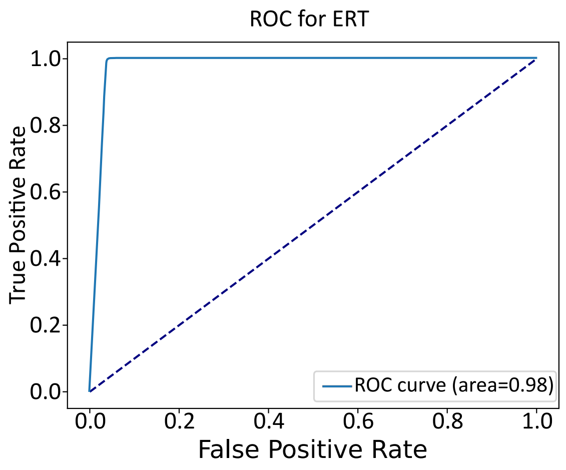
three models that output probabilities (MLP, CNN, and ERT), showing that their performance is almost identical.



(a)



(b)



(c)

Fig. 84. ROC curves produced from the (a) MLP, (b) CNN, and (c) ERT models.

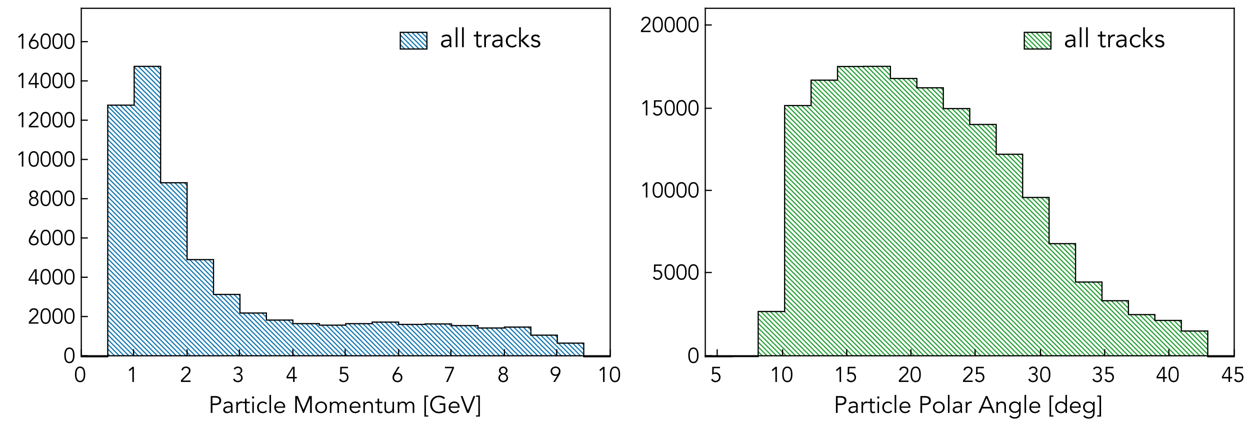
6.3.4 CLAS12 Tracking

In our studies, we used only one track type, namely negatively charged tracks, as training and validation datasets. However, the final implementation of the MLP had six input features, representing the mean values of clusters in each super-layer for both positively and negatively charged track candidates. Thus, classification in the output results in one of three possible cases: "false track", "negatively charged track", and "positively charged track". For training, we use events where two tracks were identified in one of the sectors of the drift chambers and use valid track features as positive samples for the neural network. As a negative sample ("false track"), two candidates are constructed by swapping clusters of two tracks. A random number of clusters (one or two) are swapped in each event (determined by a random number generator). This method assures that no valid reconstructed track can be represented as a "false track" in the training sample. It also provides better accuracy because it produces "false" track candidates similar to the real track by swapping only one or two clusters.

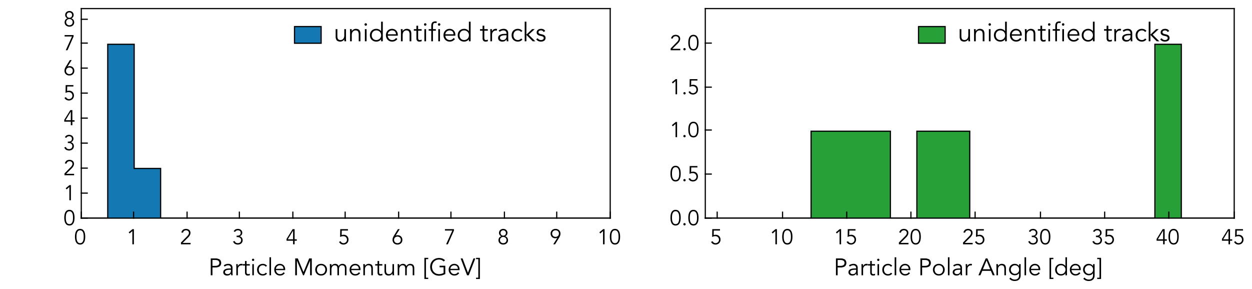
The trained network is then used to identify track candidates from real experimental data, and predictions are compared with the tracks reconstructed by the conventional algorithm. In this case, we do not use cuts/thresholds for classification. Classification is done by picking the track candidate with the highest probability. From our tests, most of the time ($> 98.5\%$), the valid track has a probability of > 0.85, but if so happens that the highest probability is 0.3, we pick the respective track and pass it to the tracking algorithm to consider.

The distribution of negative tracks is shown in Figure 85 (top row) as a function of the particle momentum and polar angle in a laboratory reference frame. In Figure 85 (bottom row), the distribution of tracks that are not identified by the network as good tracks are shown as a function of momentum and angle. The neural network did not identify only a tiny fraction (9 out of 65224 tracks) as a good track candidate. In Figure 86, the efficiency of track identification is shown as a function of the particle momentum and polar angle. The track identification accuracy for both positively and negatively charged particles is summarized in Table 18, showing efficiency $> 99.9\%$ for both particle charges.

The implementation of our MLP network was ported into the standard CLAS12 software. The ML module follows the clustering algorithm and composes all possible combinations of tracks found from each segment. The ML module then evaluates all track candidates. The suggestions of tracks are saved into intermediary data structures and are passed to the tracking algorithm, which uses these suggestions to construct and fit the final tracks. Our study with experimental data shows that our neural network can identify good tracks with efficiency above 99.9% and provide a 35% speedup compared to the existing tracking code. The time reduction comes from fewer



(a)



(b)

Fig. 85. Effectiveness of the tracking machine learning model. (a) Distribution of tracks as a function of the particle momentum and polar angle reconstructed by the conventional tracking algorithm . (b) Distribution of tracks as a function of momentum and angle for track candidates not identified by ML.

TABLE 18

Track identification efficiency on negatively and positively charged particles.

Particle Charge	Conventional Tracks	ML Predicted	ML missed	Efficiency
negative	65224	65215	9	0.999862
positive	177434	177411	23	0.999865

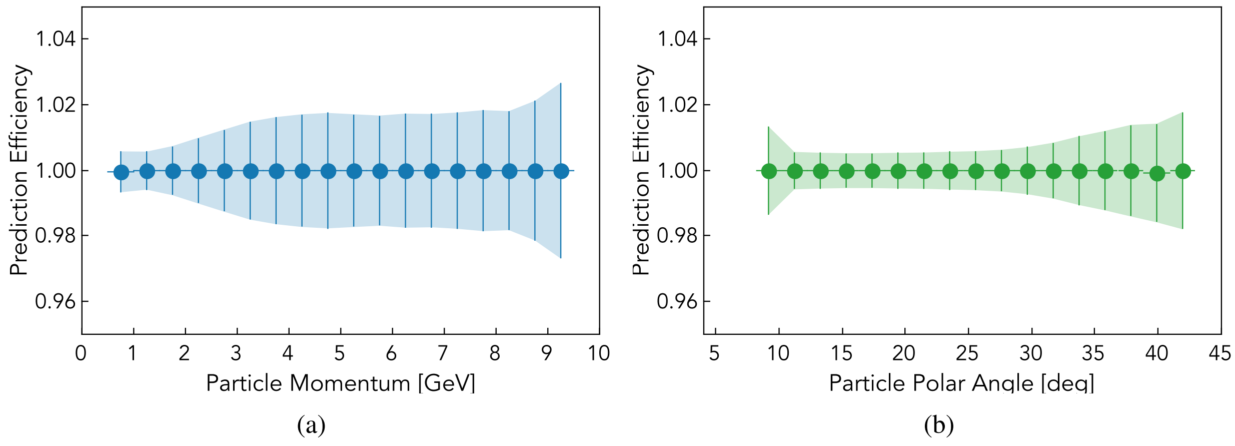


Fig. 86. Track identification efficiency using suggestions from the MLP network. Efficiency presented as a function of the particle (a) momentum and (b) polar angle.

invalid tracks needing to be fitted with the Kalman-Filter, which would be thrown away later due to non-convergence. Fitting an event's track candidates takes about 350-400 ms (depending on combinatorics). Thus, ML track candidate finding removes this overhead ($120\mu s\, vs. 350ms$) by minimizing the number of tracks that conventional algorithms must consider and intensively fit.

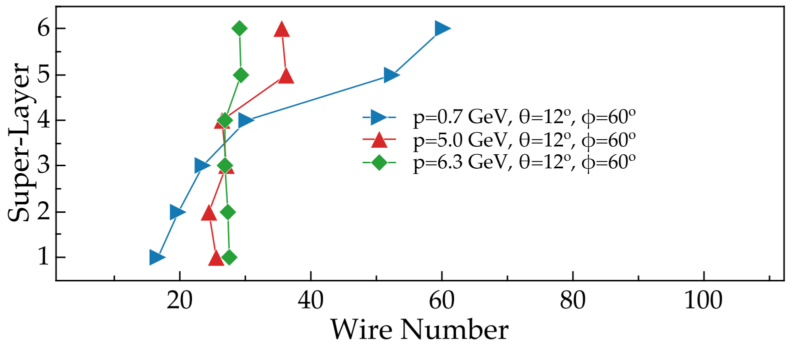
6.3.5 Summary

We have demonstrated the performance of four machine learning models, aiding the CLAS12 reconstruction code with track classification. We have shown that all four models, ERT, MLP, RNN, and CNN, perform well for this task. The MLP shows the best accuracy and inference speed results on real data. A small systematic study was implemented, showing that inference accuracy is increased substantially by training the models on datasets with false samples very similar to true ones. Finally, we evaluated the performance of the MLP model in comparison to the traditional track candidate selection algorithm method, which showed an equally good track identification efficiency and resulted in 35% track reconstruction speed up. In the future, we plan to extend the ML model by introducing a pre-processing step to further filter the candidate tracks. An RNN will be introduced to estimate the path the true track will follow. This estimation will narrow down the number of candidates the MLP needs to consider. This network will be first implemented in the CLAS12 reconstruction software to test improvements in track reconstruction efficiency. Then, it will be complemented with a denoising auto-encoder developed for CLAS12 tracking [115], which already shows significant improvements for segment identification in high background (high luminosity) conditions.

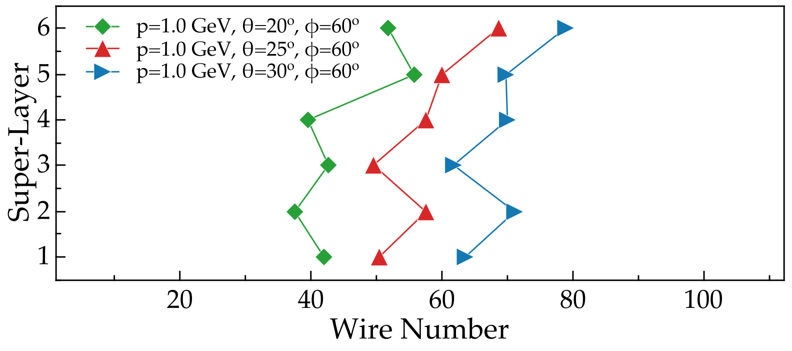
6.4 TRACK PARAMETER ESTIMATION

In this section, we use ML to reconstruct particle track parameters from valid hits identified by the previous network in the drift chamber. Combining the track-finding neural network mentioned above with a track parameter extraction network will allow experiments to reconstruct particles in real-time (at rates comparable with experimental data collection rates), achieving a significant speedup versus the traditional approach ($4ms$ vs. $350ms$). The track parameter extraction network uses clusters reconstructed in each super-layer of the drift chambers. Examples of tracks are shown in Figure 87, where cluster mean positions for each super-layer are plotted for each example track.

In Figure 87a, tracks with different momenta but with same angles ($\theta = 12^{\circ}$ and $\phi = 60^{\circ}$) are shown. The particles with different momenta passing through the solenoid magnet enter the first region of drift chambers in different locations. As seen in the figure, traveling through the entire length of drift chambers and passing through a toroidal magnetic field, particles with different momenta change their trajectory. In Figure 87b, particles with the same momenta and azimuthal angle ($p = 1\ GeV$ and $\phi = 60^{\circ}$) are shown for different polar angle values. As evident from the figures, each track has a unique set of clusters in drift chambers. We used a neural network to learn the output track parameters based on the provided input cluster positions and study different types of machine learning models and hyperparameters to determine which network provides the best parameter inference in the least amount of time.



(a)



(b)

Fig. 87. Examples of tracks in sector one of the drift chambers. The average wire position for a cluster is plotted vs. the super-layer number for the tracks. (a) tracks with different momenta but the same angles ($\theta = 12^\circ$ and $\phi = 60^\circ$). (b) tracks for different polar angles with the same momentum and azimuthal angle ($p = 1\ GeV$ and $\phi = 60^\circ$).

6.4.1 Data Description

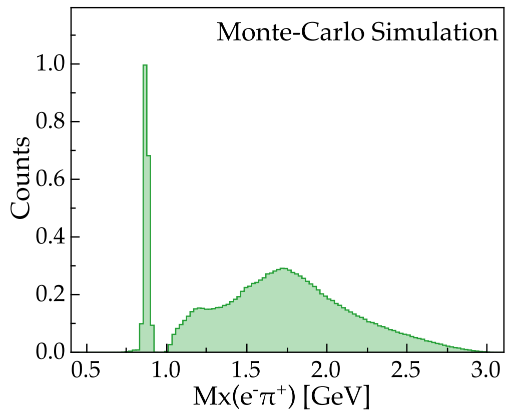
Data for these studies are produced using Geant [116], based on CLAS12 detector emulation software, EMC [117]. GEMC is a broadly used software for all analyses performed in CLAS12 [118, 119, 120] and is actively maintained to simulate high-energy physics signals accurately. The generated data are reconstructed using CLAS12 data reconstruction software [83] are then normalized and used to train and test the ML models against reconstructed track parameters. This process allows us to generate datasets large enough to train a machine-learning model optimally while assuring that the key features and patterns learned from training can accurately represent any new experiment in the existing settings.

The missing mass distribution for $H(e^-, e^-\pi^+)X$ can be seen in Figure 88, where the stages of reconstruction software are shown. Figure 88a shows the missing mass distribution for simulated events before processing them with GEMC. The missing nucleon peak is visible at mass value $m_n = 0.938$. In Figure 88b, the missing mass distribution is shown after CLAS12 software reconstruction of charged particles (e^- and π^+), using only hits in drift chambers (called Hit Based Tracking). The peak is wider because the particle passing through detector components experiences scattering, which results in energy loss and the reconstructed final momentum not being the same as the initial momentum of the particle scattered off the target. The second stage of track reconstruction is time-based tracking. Track parameters (i.e., momentum, polar and azimuthal angle) are further refined by fitting track candidates with Kalman-Filter, using timing information from hits that the track candidate contains. The final missing mass distribution after the time-based tracking with improved momentum resolution is shown in Figure 88c.

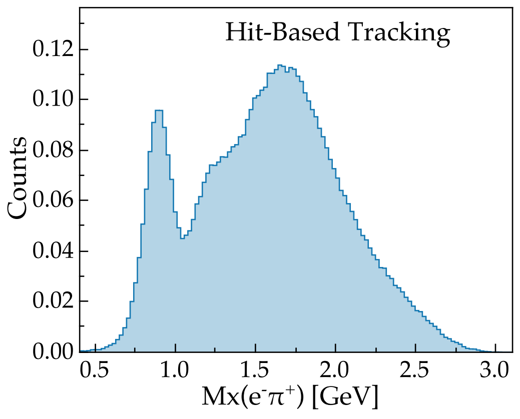
Training Dataset

The training dataset [1] consists of 6 input numbers (nodes), representing the mean wire position for a cluster in each super-layer of the drift chambers, and three output nodes, representing the particle parameters: Momentum (p), polar angle (θ), and azimuthal angle (ϕ). The reconstructed track parameters for training are taken from the results of time-based tracking. The results from the ML models are then compared to distributions from hit-based tracking. For our studies, we experimented with three ML models, namely Multi-Layer Perceptron (MLP), Extremely Randomized Trees (ERT), and Gradient Boosting Trees (GBT). For all models, we evaluated different hyperparameters and present the ones producing the best results. We constructed missing mass distributions using inferred track parameters to evaluate model performance and compared them

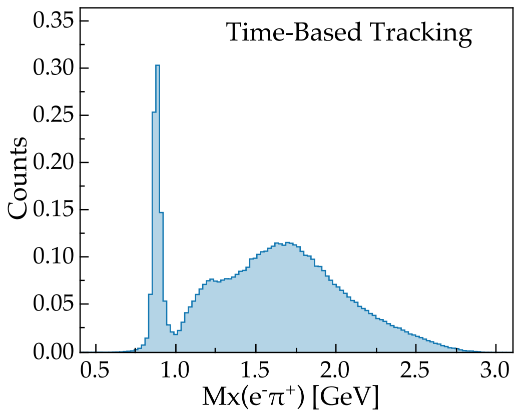
[1] Data available at `https://github.com/gavalian/artin/tree/main/projects/pe/py`



(a)



(b)



(c)

Fig. 88. Missing mass distribution of $e^-\pi^+$ for generated $H(e^-,e^-\pi^+)X$ events. Calculated using (a) particle parameters that were generated by the physics generator (Pythia), (b) particle parameters reconstructed by Hit Based Tracking (HB) algorithm and (c) using particle parameters from Time Based Tracking (TB) reconstruction algorithm.

to missing mass distributions from the hit-based tracking of CLAS12 reconstruction software.

We used randomly selected simulated data to develop the models since the geometry used in the simulation is the same for all six sectors, and the ML model can be trained for one sector and applied to all. We specifically trained each model on data from sector 2 (i.e., training set, $\approx$ 100k rows) and then evaluated them using the data from all remaining sectors (i.e., testing set, $\approx$ 100k rows). In an experimental setup, detector components are usually shifted slightly from their intended ideal positions, and track reconstruction software must consider these shifts to reconstruct track parameters accurately. Extending this method to experimental data will require training the model for each sector individually to account for slight uncertainties in detector positioning. Finally, we must note that normalization was applied to both the input and output features when building/testing each model. Their performance improved significantly compared to utilizing the raw input/output values (regardless of how the models' attributes were tuned).

6.4.2 Models Description and Performance Evaluation

In this section, we present the utilization and evaluation of three machine learning algorithms to deal with the challenge of track parameter reconstruction, namely Multi-layer Perceptron (MLP), Extremely Randomized Trees (ERT), and Gradient Boosting Trees (GBT). The three algorithms were chosen based on our previous work applying machine learning on CLAS12 data for different applications [107, 121], as well as extensive experimentation with several other algorithms, including other tree-based algorithms, convolutional and graph neural networks. We evaluated the algorithms mentioned above on the dataset presented above, tuned their hyper-parameters using grid-search cross-validation, and presented the ones that achieved the best results below.

Multi-Layer Perceptron

Due to the complexity of the problem at hand, the MLP network for this application was configured with four hidden layers. The input layer contained six neurons, the subsequent layers contained 12, 24, 24, and 12, respectively, and the output layer contained 3 (for each of the aforementioned output particle parameters). This architecture can be seen in figure 89. Several combinations of activation functions were tested, where one function would be utilized for the input layer and all hidden layers, and another function would be utilized specifically for the output layer. Each configuration's performance was evaluated by measuring its parameter estimation efficiency as a function of parameters. For example, if the true (original) value of the momentum parameter is p_i for track i and the predicted value is p, an inference resolution can be calculated as $(p - p_i)/p$.

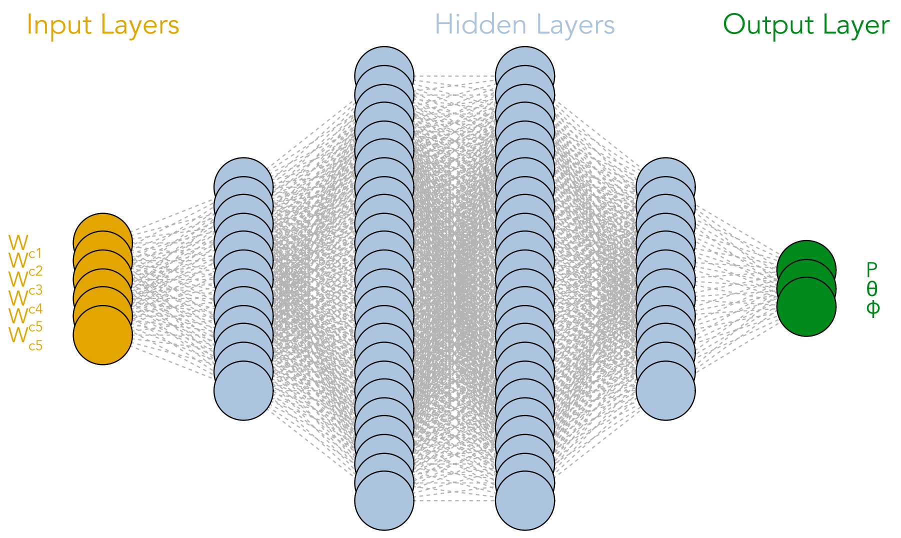


Fig. 89. The track parameter reconstruction MLP Network. The MLP is structured with an input layer of 6 neurons, corresponding to the mean wire position of a cluster in each super-layer of the drift chambers, four hidden layers (with 12, 24, 24, and 12 neurons, respectively), and an output layer of 3 neurons for the output particle parameters

TABLE 19

Comparison of activation function results for MLP.

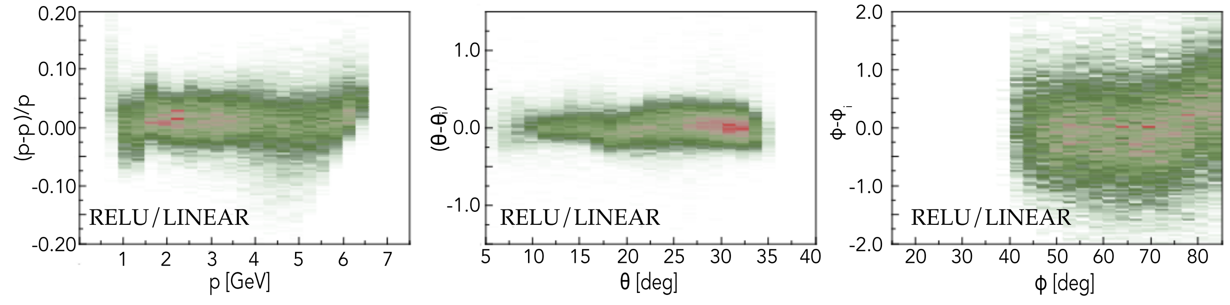
Activation Function	Mean Squared Error	Score
RELU/LINEAR	2.1e-04	0.94
RELU/TANH	2.8e-04	0.93
RELU/SIGMOID	2.5e-04	0.94
TANH/TANH	2.1e-04	0.94
TANH/LINEAR	1.9e-04	0.94
SIGMOID/LINEAR	5.8e-04	0.91
SIGMOID/SIGMOID	6.4e-03	0.68

Figure 90 shows the inference calculated for each output parameter using different combinations of activation functions. Figure 91 gives a Gaussian fitted over each of the plots in figure 90, showing mean inferences with the sigmas of the fits set as error bars for each point. Table 19 shows the mean squared error and score given by each combination of activation functions after testing the trained models.

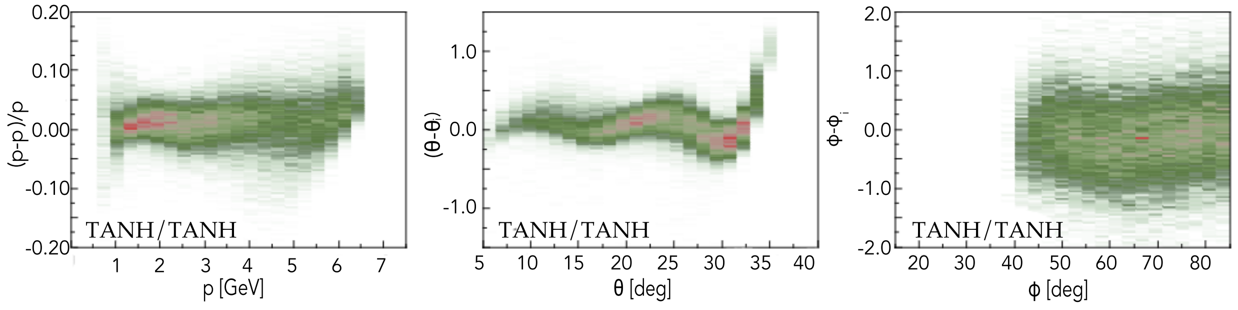
The number of epochs used for training was 1500, and the batch size was 64. The solver used for weight optimization was "Adam", a stochastic gradient descent method designed based on adaptive estimates of lower-order moments [111]. We utilized a learning rate of 0.0001 and a beta 1 value of 0.99. This beta 1 parameter of the optimizer affects the exponential decay rate for the first moment estimates (allowing larger changes to be made to the weights during the initial steps of training and later reduced to be fine-tuned). Finally, the AMSGrad variant of the algorithm was applied (which optimizes the scaling of gradient updates when calculating the square roots of exponential moving averages of squared past gradients) [122]. This optimization is effective when training networks with larger output spaces (more than one parameter). The default values of the remaining parameters of the "Adam" optimizer in TensorFlow's Keras API were used.

Extremely Randomized Trees

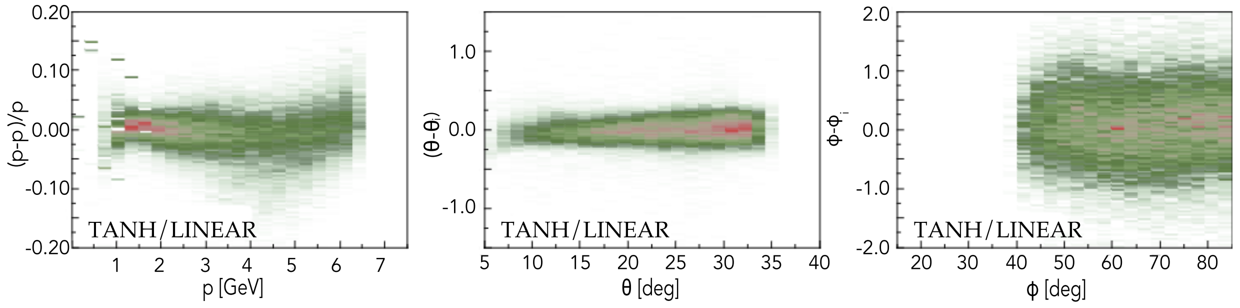
Our application uses a model with three hundred estimators (decision trees), with no limit on the number of features to be considered when splitting. In addition, we used the information gain (entropy) split quality criterion. The rest of the parameters were kept at their default values. Figure 92 shows the performance of this algorithm for different tracks. The inferences of the



(a)

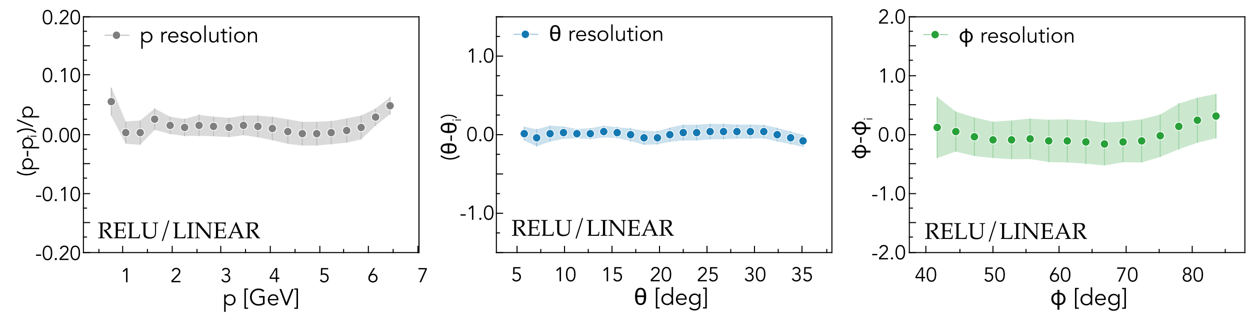


(b)

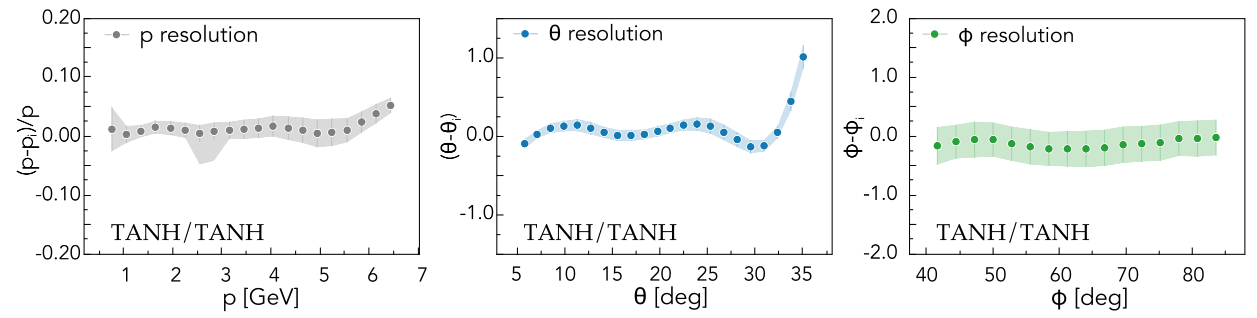


(c)

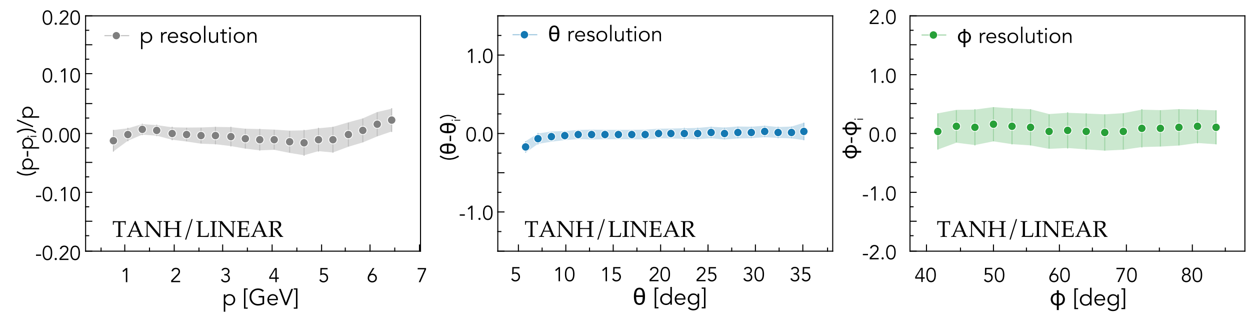
Fig. 90. Reconstructed resolution for track parameters. p(left), θ(middle), ϕ(right), using the MLP with different combinations of activation functions for the hidden and output layers. (a) Hidden layers: *ReLU*. Output layer: *linear*, (b) Hidden layers: *tanh*. Output layer: *tanh*, (c) Hidden layers: *tanh*. Output layer: *linear*.



(a)

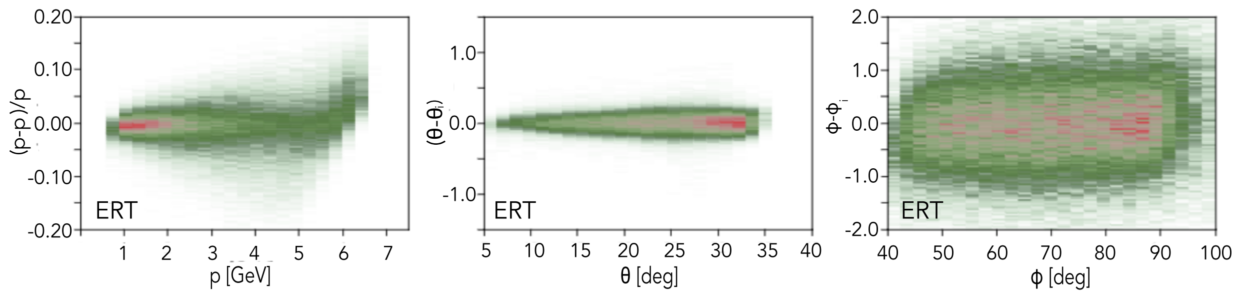


(b)

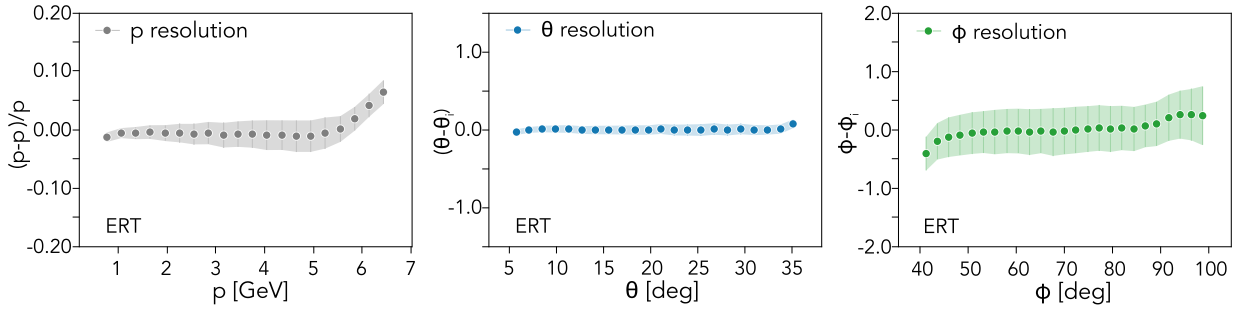


(c)

Fig. 91. Gaussian fit of the reconstructed resolution means for track parameters. p(left), θ(middle), ϕ(right), using the MLP with different combinations of activation functions for the hidden and output layers. (a) Hidden layers: *ReLU*. Output layer: *linear*, (b) Hidden layers: *tanh*. Output layer: *tanh*, (c) Hidden layers: *tanh*. Output layer: *linear*. The sigmas of the fit are set as error bars for each point.

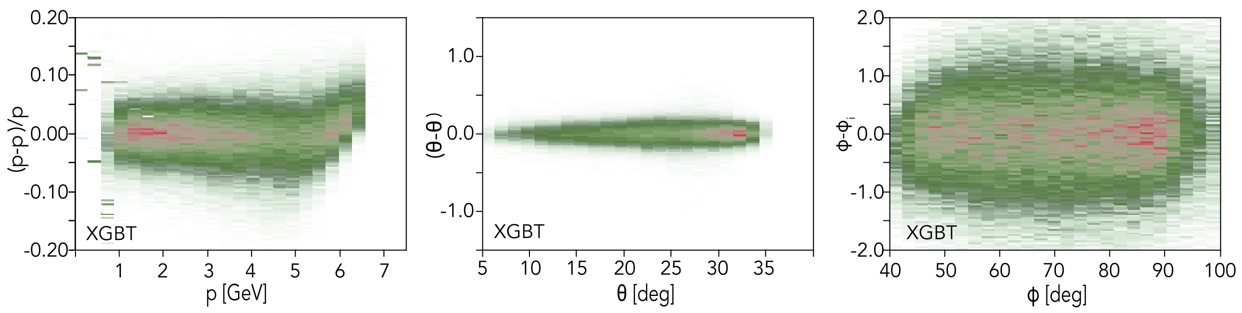


(a)

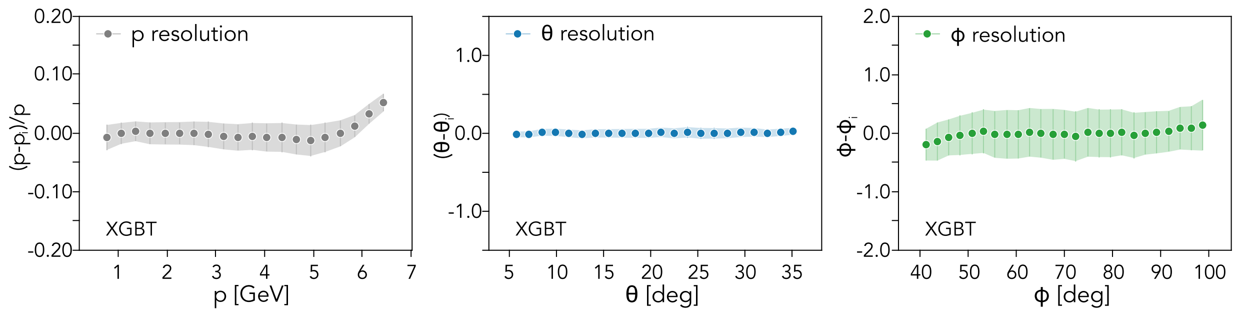


(b)

Fig. 92. Reconstruction performance for p(left), θ(middle), ϕ(right), using the ERT model. (a) The reconstructed resolution, (b) Gaussian fit of the reconstructed resolution mean. Sigmas are set as error bars.



(a)



(b)

Fig. 93. Reconstruction performance for p(left), θ(middle), ϕ(right), using the GBT. (a) The reconstructed resolution, (b) Gaussian fit of the reconstructed resolution mean. Sigmas are set as error bars.

algorithm remain within 5% of the actual track parameters, with performance declining to 10% when the momentum increases. For the polar and azimuthal angles, the error remains within 1 degree in all cases.

Gradient Boosting Trees

The Gradient Boosting Trees (GBT) network was trained on simulated sample data to extract the performance of reconstructing track parameters and compare them to other networks (shown in Figure 93). We used grid-search cross-validation to tune the parameters for the model used in these experiments. We found that the following parameters produced the best results: *learning rate:* 0.1, *max depth:* 6, *max delta step:* no restriction, *gamma:* no restriction, *min child weight:* 7, *number of parallel trees:* 4, *number of boosting iterations:* 1300. The performance of the GBT algorithm is presented in Figure 93 and shows trends similar to the ones observed in the ERT model.

TABLE 20

Training and inference time per sample for different network models.

Model	Training Time (ms)	Inference Time (ms)
Multi-Layer Perceptron	0.015	0.018
Extremely Randomized Trees	0.057	0.019
Gradient Boosting Trees	0.087	0.027

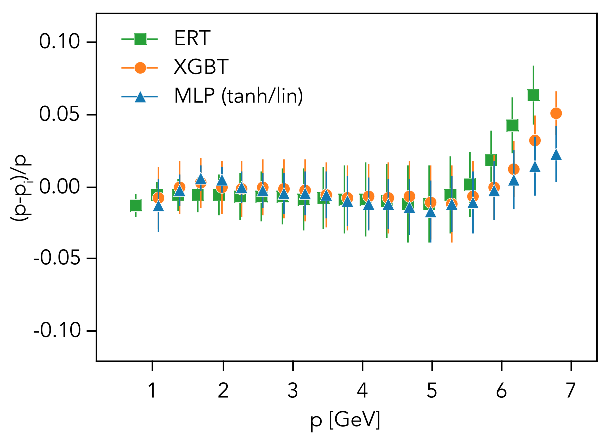
6.4.3 Results

This section presents studies with several neural network architectures to reconstruct charged particle momentum and direction using raw data from CLAS12 drift chambers. The precision of reconstructed parameters is summarized in Figure 94. The Multi-Layer perceptron performs better in momentum reconstruction precision along the generated range, while ERT and GBT underperform in the high momentum range. The direction of the particle (polar and azimuthal angle) is also reconstructed better with MLP, primarily flat across the entire range.

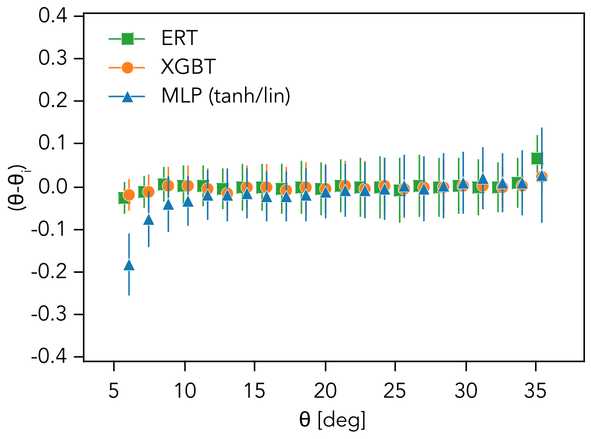
The inference times were measured for each of the networks (presented in Table 20). All models perform similarly, with MLP being faster on inference times and capable of processing events with 22 kHz rate (calculated using 2.5 tracks on average in each physics event). When used with a track classifier network, accounting for the time spent on clustering the data and running classification on all possible combinations of track candidates, we estimated that the full track reconstruction can be achieved in real-time with a rate of 32 kHz, faster than data acquisition speed. The real-time track reconstruction rate is calculated by considering the clustering time from drift chamber reconstruction and track candidate classification time using the current implementation of track reconstruction using artificial intelligence [123].

The developed models were used to analyze simulated data and calculate missing mass distribution, compared with the results of conventional hit-based tracking. The comparison of track parameters, such as momentum, polar angle, and azimuthal angle, are shown in Figure 95 for simulated reaction $H(e^-,e^-\pi^+)X$ for both methods. The particle parameter distributions presented for positively- (top row) and negatively- (bottom row) charged particles match those reconstructed by conventional hit-based tracking algorithms very well.

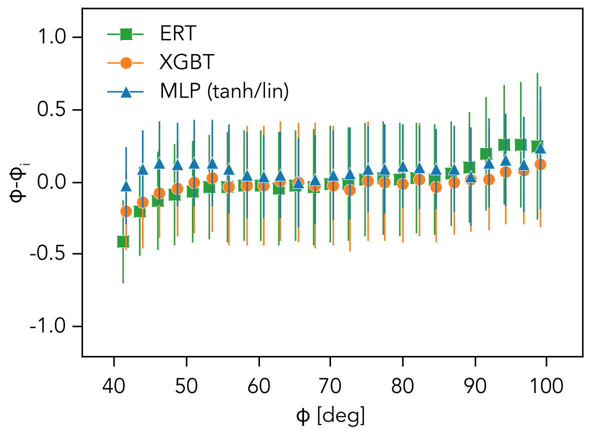
Further, the missing mass distributions for $H(e^-,e^-\pi^+)X$ are calculated for conventional and AI tracking, and results are presented in Figure 96. The filled histograms show distribution from the conventional algorithm, while the solid line histograms are from AI track reconstruction. It is



(a)

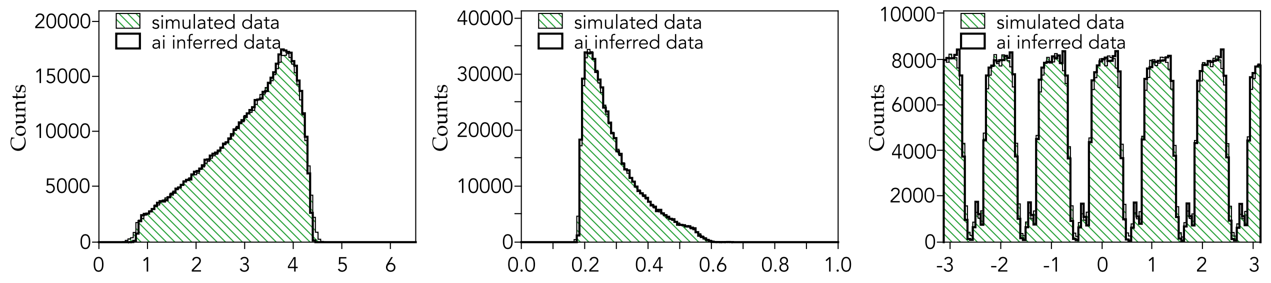


(b)

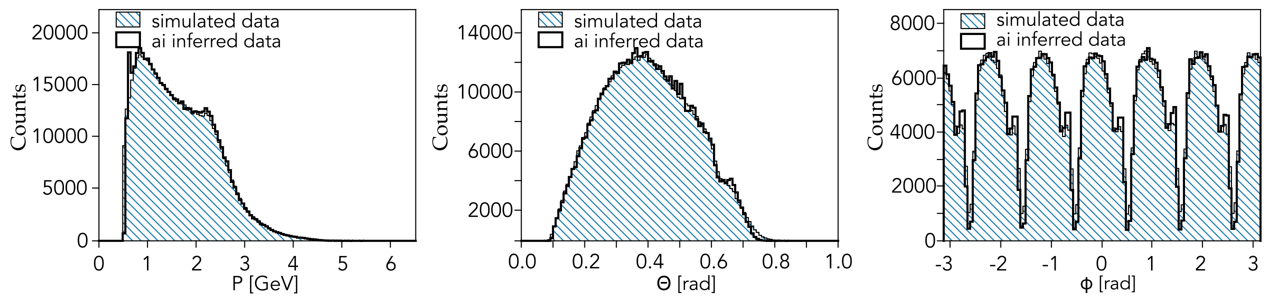


(c)

Fig. 94. Machine Learning models performance comparison. A comparison among the Multi-Layer Perceptron (MLP), Extremely Randomized Trees (ERT), and Gradient Boosting Trees (GBT) on the testing dataset, with respect to (a) momentum, (b) polar angle and (c) azimuthal angle.



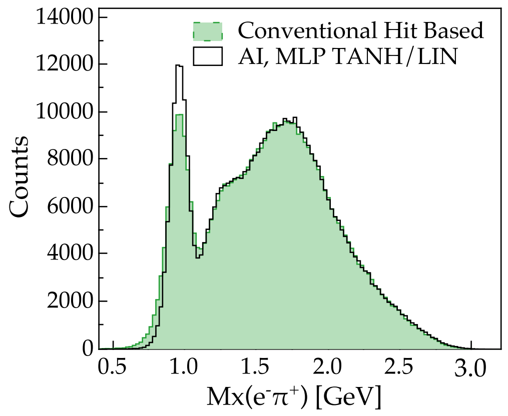
(a)



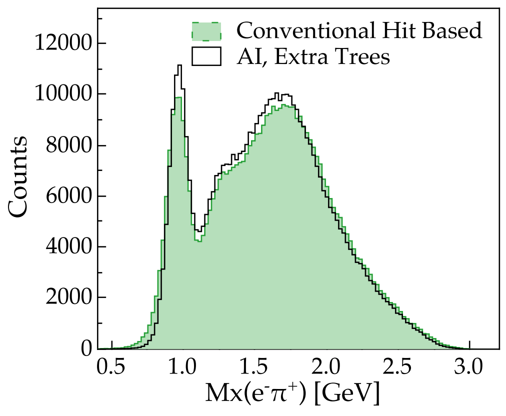
(b)

Fig. 95. Parameters for (a) negative and (b) positive tracks with Hit-Based and ML tracking.

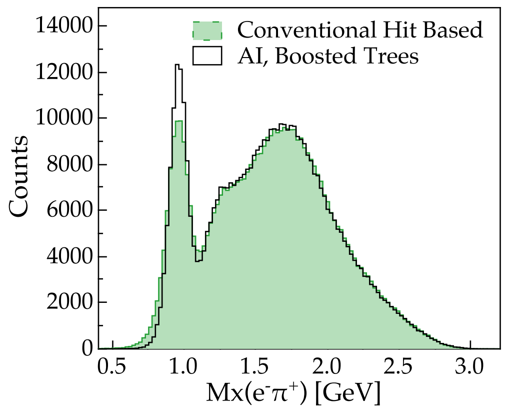
evident that in all three cases (for all networks), the AI-inferred track parameters result in a much better missing mass distribution resolution. The Gradient Boosting Trees network performs the best, while MLP comes second. As stated, our goal is to reconstruct particle parameters and attain inference in the shortest possible time. While the MLP model performs faster in inference time, it would take a significantly longer training time for its inferred track parameters to provide a better missing mass distribution resolution. For example, the MLP model was also trained with 10,000 epochs rather than 1,500, and while the inference slightly improved, it still did not outperform the GBT model. Given the performance of the GBT model, we concluded that the trade-off between training time/hardware resources and equalized parameter estimation performance provided by the other models did not garner sufficient merit. Furthermore, gaining a slight improvement in inference does not warrant any extensive man labor required to fine-tune each model to the highest possible optimizations (e.g., further testing different configurations of activation layers in the MLP model, number of neurons per layer, etc.). Highly accurate results are achieved in a short time using the model attributes selected. Overall, we conclude that the missing mass distribution calculated from the GBT model's track parameters yields the best results compared to the other models of our



(a)



(b)



(c)

Fig. 96. Missing mass distribution of $e^{-}\pi^{+}$, for generated $H(e^{-},e^{-}\pi^{+})X$ events. Results produced by using different networks, (a) MLP, (b) ERT, (c) GBT, are compared to the missing mass reconstructed by a conventional tracking algorithm at the hit-based level.

experiment.

6.4.4 Summary

In this work, we presented a machine-learning approach to reconstructing charged track parameters in CLAS12 using only hit positions in drift chambers. The models were trained using simulated track parameters (i.e., momentum, polar and azimuthal angles). Multi-Layer Perceptron (MLP), Extremely Randomized Trees (ERT), and Gradient Boosting Trees (GBT) were used as test networks, with GBT showing the best performance in track parameter inference. The tracks reconstructed by the ML models were used to calculate physics observables, namely missing mass of the $H(e^-,e^-\pi^+)X$ system, which was compared to missing mass calculated using tracks reconstructed by a conventional hit-based tracking algorithm.

It was shown that the reconstructed nucleon final state using ML-inferred tracks has better resolution than the conventional approach. Moreover, the inference takes a fraction of the time (4 *ms* per event) compared to the conventional hit-based tracking algorithm (350 *ms* per event). The achieved model is not enough to do physics analysis and isolate rare physics reactions; however, it offers many other benefits.

This approach to data reconstruction has significant implications for detector monitoring when conducting nuclear physics experiments. For example, a typical experiment in CLAS12 collects data at a rate of 11 *kHz* with a neural network capable of reconstructing real physics quantities in real-time (using a 48 core CPU) and allowing the monitoring of detector performance. Thus, this approach also opens the door to performing real-time calibrations.

CHAPTER 7

CONCLUSION AND FUTURE WORK

In this work, we have contributed to two research areas: (1) Parallel runtime systems for exascale-era platforms and (2) High-energy physics utilizing machine learning. In this section, we summarize the contributions in each area and discuss possible future directions for these works.

In parallel runtime systems, we have introduced a novel design and implementation of PREMA that significantly refactors and extends the initial version in multiple directions. In particular, Chapter 3 shows the advantages of enhancing PREMA with multiple threads and presents an evolutionary methodology for porting legacy MPI systems to multi-core platforms based on the principle of separation of concerns. The derived implementation can leverage both shared and distributed memory parallelism implicitly and provides an interface that one can utilize to develop custom 2-level scheduling and load-balancing policies. It also provides performance improvements on a parallel 3D advancing front mesh refinement application achieving up to 56% speedup compared to an MPI implementation. In Chapter 4, we have presented the integration of PREMA with lightweight threads utilizing Argobots. This effort overcomes limitations previously exhibited by PREMA while incorporating features for effortlessly handling control flow in shared and distributed memory, as well as a tasking framework that allows for fine-grained parallelism inside the execution of remote method invocations. We showed that the integration of tasking on top of remote method invocations could have tremendous effects on irregular applications, like 3D mesh refinement, achieving up to 50% improvement through exhibiting multi-grain over-decomposition both on the data and task level. Finally, Chapter 5 introduced an efficient tasking framework that handles performance portability for multi-device heterogeneous nodes of up to 300%. Integrating this framework into PREMA allows it to leverage exascale HPC systems consisting of multiple heterogeneous nodes. Evaluation results on a proxy distributed memory application show that the end product of this work incurs low latency and scalable performance (up to 40% versus the MPI+CUDA) while providing a simple and uniform interface independent of the target hardware.

While PREMA has been thoroughly refactored to accommodate modern, exascale-era platforms better, there are still features missing that could further improve its impact:

- **Fault Tolerance:** A significant issue for modern computing platforms is the declining Mean Time to Failure (MTTF) stemming from the ever-increasing number of computing nodes. PREMA could facilitate application-driven fault tolerance by utilizing the abstraction of mobile objects. For example, the runtime system could easily store copies of such data onto reliable destinations, keep track of the involved processing elements and, in case of faults, use the most recent versions to recover the application in a consistent state.

- **Power Management:** Power consumption is another constraint that is becoming relevant as the size of HPC systems increases. Applications not only need to run fast, but they also need to adhere to a power budget to be efficient. PREMA could assist in this regard by automatically monitoring the overall power used and dynamically adjusting the number of CPU cores it utilizes to limit the energy it consumes effectively.

- **Integration with Machine Learning:** Machine learning has become an integral part of today's technology with enormous capabilities that we are still exploring. PREMA could leverage these capabilities to make better overall decisions and optimize efficiency.

Even though our involvement in high-energy physics started just as a bridge to train and practice machine learning, we eventually accomplished significant contributions that are part of the Jefferson Lab's production code, providing valuable speedups in processing time. In Section 6.2, we used Convolutional Auto Encoder neural networks to denoise raw data from CLAS12 drift chambers. The Convolutional Auto Encoder showed excellent performance, reducing noise hits by more than 90% while retaining, on average, 94% of the valid hits. For high background conditions, the gains in track reconstruction are much higher, e.g., up to 1.8 times improvement for 110nA. In Section 6.3, we describe the development of four machine learning (ML) models that assist the tracking algorithm by identifying valid track candidates from the measurements in drift chambers. As a result of this work, an MLP network classifier was implemented as part of the CLAS12 reconstruction software to provide the tracking code with recommended track candidates. The resulting software achieved an accuracy of greater than 99% and resulted in an end-to-end speedup of 35% compared to existing algorithms. Finally, in Section 6.4, we present studies of track parameter reconstruction from raw information in CLAS12 detector's Drift Chambers. We study the resolution of tracks reconstructed with different types of ML models/algorithms using simulated data; the resulting model is capable of reconstructing track parameters (particle momentum, and polar and azimuthal angles) with accuracy similar to Hit Based (HB) tracking code, but 150 times faster, which allows physics reactions to be identified in real-time (with a rate of about 34 kHz) during experimental data collection.

Based on our experience with using machine learning in this area, some apparent applications that could potentially leverage this approach in the future include:

- **Detector Monitoring:** Reconstructing tracks from separate sectors in the drift chamber can help to monitor the detector's health and make it easy to identify newly developed inefficient regions of tracking detectors.

- **Real-Time Detector Calibration:** The detector components can be initially calibrated during the experimental data collection without extensive data processing after data taking for calibration purposes.

- **Improved Data Processing speed:** With initial track parameter reconstruction during data taking, the conventional algorithms can skip the process of hit-based tracking and rely on final, time-based tracking to improve the measured track parameters. This can significantly reduce data processing times since hit-based tracking currently comprises 35% of the total tracking time.

- **Physics Reaction Identification:** High-intensity experiments searching for rare reactions process a large amount of data to isolate required event topologies. Tagging the physics reactions in real-time during experimental data collection can significantly reduce data analysis and reaction search times.

REFERENCES

[1] R. E. Kalman, “A new approach to linear filtering and prediction problems,” *Journal of Basic Engineering*, vol. 82, no. 1, pp. 35–45, 1960.

[2] V. Burkert *et al.*, “The CLAS12 spectrometer at Jefferson laboratory,” *Nucl. Instrum. Meth. A*, vol. 959, p. 163419, 2020.

[3] D. Feng, A. N. Chernikov, and N. P. Chrisochoides, “A hybrid parallel Delaunay image-to-mesh conversion algorithm scalable on distributed-memory clusters,” *Procedia Engineering*, vol. 163, pp. 59–71, 2016.

[4] G. Dodge and T. Whitfield, “AI Collaboration Between ODU, Jefferson Lab Improves Data Analysis,” [Online]. Available: https://www.odu.edu/article/ai-collaboration-between-odu-jefferson-lab-improves-data-analysis, 2022, accessed: 2023-04-01.

[5] G. Gavalian, Personal Communication, Feb 2023.

[6] P. Messina, “The exascale computing project,” *Computing in Science & Engineering*, vol. 19, no. 3, pp. 63–67, 2017.

[7] N. P. Chrisochoides, “Telescopic approach for extreme-scale parallel mesh generation for CFD applications,” in *Proc. 46th AIAA Fluid Dynamics Conference*, 2016, p. 3181.

[8] N. Chrisochoides, “Parallel mesh generation,” *Numerical Solution of Partial Differential Equations on Parallel Computers*, vol. 51, pp. 237–259, 2005.

[9] C. H. Koelbel and M. E. Zosel, “The high performance FORTRAN handbook.” MIT Press, 1993.

[10] B. L. Chamberlain, S.-E. Choi, E. C. Lewis, C. Lin, L. Snyder, and W. D. Weathersby, “ZPL: A machine independent programming language for parallel computers,” *IEEE Trans. Softw. Eng.*, vol. 26, no. 3, pp. 197–211, 2000.

[11] M. C. Rinard and M. S. Lam, “The design, implementation, and evaluation of Jade,” *ACM Trans. Program. Lang. Syst.*, vol. 20, no. 3, pp. 483–545, 1998.

[12] R. W. Numrich and J. Reid, “Co-array fortran for parallel programming,” *SIGPLAN Fortran Forum*, vol. 17, no. 2, pp. 1–31, 1998.

[13] A. Krishnamurthy, D. E. Culler, A. Dusseau, S. C. Goldstein, S. Lumetta, T. von Eicken, and K. Yelick, "Parallel programming in Split-C," in *Proc. ACM/IEEE Conference on Supercomputing*, 1993, pp. 262–273.

[14] W. W. Carlson, J. M. Draper, D. E. Culler, K. A. Yelick, E. D. Brooks, and K. H. Warren, "Introduction to UPC and language specification," UC Berkeley, Tech. Rep., 1999.

[15] K. A. Yelick *et al.*, "Titanium: A high performance Java dialect," *Concurrency - Practice and Experience*, vol. 10, pp. 825–836, 1998.

[16] J. Nieplocha, B. Palmer, V. Tipparaju, M. Krishnan, H. Trease, and E. Aprà, "Advances, applications and performance of the global arrays shared memory programming toolkit," *International Journal of High Performance Computing Applications*, vol. 20, pp. 203–231, 2006.

[17] Y. Zheng, A. Kamil, M. B. Driscoll, H. Shan, and K. A. Yelick, "UPC++: A PGAS extension for C++," in *Proc. IEEE 28th International Parallel and Distributed Processing Symposium*, 2014, pp. 1105–1114.

[18] P. Charles, C. Grothoff, V. Saraswat, C. Donawa, A. Kielstra, K. Ebcioglu, C. von Praun, and V. Sarkar, "X10: An object-oriented approach to non-uniform cluster computing," *SIGPLAN Not.*, vol. 40, no. 10, pp. 519–538, 2005.

[19] B. Chamberlain, D. Callahan, and H. Zima, "Parallel programmability and the Chapel language," *Int. J. High Perform. Comput. Appl.*, vol. 21, no. 3, pp. 291–312, 2007.

[20] V. Cavé, J. Zhao, J. Shirako, and V. Sarkar, "Habanero-Java: The new adventures of old X10," in *Proc. 9th International Conference on Principles and Practice of Programming in Java*, 2011, pp. 51–61.

[21] D. Majeti and V. Sarkar, "Heterogeneous Habanero-C (h2c): A portable programming model for heterogeneous processors," in *Proc. IEEE International Parallel and Distributed Processing Symposium Workshop*, 2015, pp. 708–717.

[22] R. K. Raj, E. Tempero, H. M. Levy, A. P. Black, N. C. Hutchinson, and E. Jul, "Emerald: A general-purpose programming language," *Software: Practice and Experience*, vol. 21, no. 1, pp. 91–118, 1991.

[23] J. Chase, F. Amador, E. Lazowska, H. Levy, and R. Littlefield, “The Amber system: Parallel programming on a network of multiprocessors,” *SIGOPS Oper. Syst. Rev.*, vol. 23, no. 5, pp. 147–158, 1989.

[24] H. Bal, M. Kaashoek, and A. Tanenbaum, “Orca: A language for parallel programming of distributed systems,” *IEEE Transactions on Software Engineering*, vol. 18, no. 3, pp. 190–205, 1992.

[25] L. V. Kale and S. Krishnan, “Charm++: A portable concurrent object oriented system based on C++,” *SIGPLAN Not.*, vol. 28, no. 10, pp. 91–108, 1993.

[26] P. Cicotti and S. B. Baden, “Asynchronous programming with Tarragon,” in *Proc. 2006 ACM/IEEE Conference on Supercomputing*. Association for Computing Machinery, 2006, pp. 159–169.

[27] H. Kaiser, T. Heller, B. Adelstein-Lelbach, A. Serio, and D. Fey, “HPX: A task based programming model in a global address space,” in *Proc. 8th International Conference on Partitioned Global Address Space Programming Models*, 2014, pp. 1–11.

[28] A. Deshpande and M. Schultz, “Efficient parallel programming with Linda,” in *Proc. 1992 ACM/IEEE Conference on Supercomputing*, 1992, pp. 238–244.

[29] Z. Budimlić *et al.*, “Concurrent Collections,” *Sci. Program.*, vol. 18, no. 3–4, pp. 203–217, 2010.

[30] A. Buss *et al.*, “STAPL: Standard template adaptive parallel library,” in *Proc. 3rd Annual Haifa Experimental Systems Conference*, 2010, pp. 1–10.

[31] M. Kulkarni, K. Pingali, B. Walter, G. Ramanarayanan, K. Bala, and L. P. Chew, “Optimistic parallelism requires abstractions,” in *Proc. 28th ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2007, p. 211–222.

[32] G. Bosilca, A. Bouteiller, A. Danalis, T. Herault, P. Lemarinier, and J. Dongarra, “DAGuE: A generic distributed DAG engine for high performance computing,” *Parallel Computing*, vol. 38, no. 1, pp. 37–51, 2012.

[33] C. Augonnet, S. Thibault, R. Namyst, and P. A. Wacrenier, “StarPU: A unified platform for task scheduling on heterogeneous multicore architectures,” *Concurr. Comput.: Pract. Exper.*, vol. 23, no. 2, pp. 187–198, 2011.

[34] M. Bauer, S. Treichler, E. Slaughter, and A. Aiken, "Legion: Expressing locality and independence with logical regions," in *Proc. International Conference on High Performance Computing, Networking, Storage and Analysis*, 2012, pp. 1–11.

[35] L. Cambier, Y. Qian, and E. F. Darve, "TaskTorrent: a lightweight distributed task-based runtime system in C++," in *Proc. IEEE/ACM 3rd Annual Parallel Applications Workshop: Alternatives To MPI+X (PAW-ATM)*, 2020, pp. 16–26.

[36] T. G. Mattson *et al.*, "The open community runtime: A runtime system for extreme scale computing," in *Proc. IEEE High Performance Extreme Computing Conference (HPEC)*, 2016, pp. 1–7.

[37] L. Dagum and R. Menon, "OpenMP: an industry standard API for shared-memory programming," *Computational Science & Engineering*, vol. 5, no. 1, pp. 46–55, 1998.

[38] J. M. Perez, R. M. Badia, and J. Labarta, "Handling task dependencies under strided and aliased references," in *Proc. 24th ACM International Conference on Supercomputing*, 2010, pp. 263–274.

[39] D. A. Beckingsale, J. Burmark, R. Hornung, H. Jones, W. Killian, A. J. Kunen, O. Pearce, P. Robinson, B. S. Ryujin, and T. R. Scogland, "RAJA: Portable performance for large-scale scientific applications," in *Proc. IEEE/ACM International Workshop on Performance, Portability and Productivity in HPC (P3HPC)*, 2019, pp. 71–81.

[40] H. Carter Edwards, C. R. Trott, and D. Sunderland, "Kokkos: Enabling manycore performance portability through polymorphic memory access patterns," *Journal of Parallel and Distributed Computing*, vol. 74, 2014.

[41] K. Group, "SYCL," [Online]. Available: https://www.khronos.org/sycl/.

[42] R. D. Blumofe, C. F. Joerg, B. C. Kuszmaul, C. E. Leiserson, K. H. Randall, and Y. Zhou, "Cilk: An efficient multithreaded runtime system," *Journal of Parallel and Distributed Computing*, vol. 37, no. 1, pp. 55–69, 1996.

[43] I. Corporation, "Advanced HPC threading: Intel oneAPI threading building blocks," [Online]. Available: https://www.intel.com/content/www/us/en/developer/tools/oneapi/onetbb.html#gs.paw2fc.

[44] NVIDIA, P. Vingelmann, and F. H. Fitzek, "CUDA," [Online]. Available: https://developer.nvidia.com/cuda-toolkit.

[45] J. E. Stone, D. Gohara, and G. Shi, "OpenCL: A parallel programming standard for heterogeneous computing systems," *Computing in Science Engineering*, vol. 12, no. 3, pp. 66–73, 2010.

[46] K. Barker, A. Chernikov, N. Chrisochoides, and K. Pingali, "A load balancing framework for adaptive and asynchronous applications," *IEEE Transactions on Parallel and Distributed Systems*, vol. 15, pp. 183–192, 2004.

[47] P. Thomadakis, C. Tsolakis, and N. Chrisochoides, "Multithreaded runtime framework for parallel and adaptive applications," *Eng. with Comput.*, vol. 38, no. 5, pp. 4675–4695, 2022.

[48] N. Chrisochoides, "PREMA: Portable runtime environment for multicomputer architectures," [Online]. Available: https://web.archive.org/web/19970609213320/http://www.cs.cornell.edu/Info/People/nikosc/projects/prema/index.html.

[49] T. von Eicken, D. E. Culler, S. C. Goldstein, and K. E. Schauser, "Active messages: A mechanism for integrated communication and computation," *SIGARCH Comput. Archit. News*, vol. 20, no. 2, pp. 256–266, 1992.

[50] K. Barker, N. Chrisochoides, D. Nave, J. Dobellaere, and K. Pingali, "Data movement and control substrate for parallel adaptive applications," *Concurrency and Computation:Practice and Experience*, pp. 77–105, 2002.

[51] N. Chrisochoides, I. Kodukula, and K. Pingali, "Data movement and control substrate for parallel scientific computing," in *Proc. Communication and Architectural Support for Network-Based Parallel Computing*, 1997, pp. 256–268.

[52] N. Chrisochoides, K. Barker, D. Nave, and C. Hawblitzel, "Mobile object layer: A runtime substrate for parallel adaptive and irregular computations," *Adv. Eng. Softw.*, vol. 31, no. 8-9, pp. 621–637, 2000.

[53] N. Chrisochoides and C. Hawblitzel, "Data migration substrate for the load balancing of parallel adaptive unstructured mesh computations," in *Proc. 6th Int'l Conf. on Numerical Grid Generation in Computational Field Simulation*, 1998.

[54] A. Fedorov and N. Chrisochoides, "Location management in object-based distributed computing," in *Proc. IEEE International Conference on Cluster Computing*, 2004, pp. 299–308.

[55] D. Nave, N. Chrisochoides, and L. Chew, “Guaranteed-quality parallel delaunay refinement for restricted polyhedral domains,” *Computational Geometry*, vol. 28, no. 2, pp. 191–215, 2004.

[56] K. Garner, P. Thomadakis, T. Kennedy, C. Tsolakis, and N. Chrisochoides, “On the end-user productivity of a pseudo-constrained parallel data refinement method for the advancing front local reconnection mesh generation software,” in *Proc. AIAA Aviation Forum 2019*, 2019.

[57] M. Balasubramaniam, K. Barker, I. Banicescu, N. Chrisochoides, J. Pabico, and R. Carino, “A novel dynamic load balancing library for cluster computing,” in *Proc. 3rd International Symposium on Parallel and Distributed Computing*, 2004, pp. 346–353.

[58] R. D. Blumofe and C. E. Leiserson, “Scheduling multithreaded computations by work stealing,” *J. ACM*, vol. 46, no. 5, pp. 720–748, 1999.

[59] R. M. Metcalfe and D. R. Boggs, “Ethernet: Distributed packet switching for local computer networks,” *Commun. ACM*, vol. 19, no. 7, pp. 395–404, 1976.

[60] D. Dechev, P. Pirkelbauer, and B. Stroustrup, “Understanding and effectively preventing the ABA problem in descriptor-based lock-free designs,” in *Proc. 13th IEEE International Symposium on Object/Component/Service-Oriented Real-Time Distributed Computing*, 2010, pp. 185–192.

[61] A. Chernikov and N. Chrisochoides, “Parallel guaranteed quality delaunay uniform mesh refinement,” *SIAM Journal on Scientific Computing*, vol. 28, no. 5, pp. 1907–1926, 2006.

[62] F. Drakopoulos, C. Tsolakis, and N. P. Chrisochoides, “Fine-grained speculative topological transformation scheme for local reconnection methods,” *AIAA Journal*, vol. 57, no. 9, pp. 4007–4018, 2019.

[63] N. Petersson and B. Sjögreen, “Sw4 v1.1,” [Online]. Available: https://geodynamics.org/cig/software/sw4/, 2014.

[64] S. D. et al., “Tests of 3d elastodynamic codes: Final report for lifelines project 1a01,” *Technical Report. Pacific Eartquake Engineering Center*, 2001.

[65] S. Seo *et al.*, “Argobots: A lightweight low-level threading and tasking framework,” *IEEE Transactions on Parallel and Distributed Systems*, vol. 29, no. 3, pp. 512–526, 2018.

[66] K. B. Wheeler, R. C. Murphy, and D. Thain, “Qthreads: An api for programming with millions of lightweight threads,” in *Proc. IEEE International Symposium on Parallel and Distributed Processing*, 2008, pp. 1–8.

[67] J. Nakashima and K. Taura, “Massivethreads: A thread library for high productivity languages,” *Concurrent Objects and Beyond: Papers dedicated to Akinori Yonezawa on the Occasion of His 65th Birthday*, pp. 222–238, 2014.

[68] K. Taura and A. Yonezawa, “Fine-grain multithreading with minimal compiler support—a cost effective approach to implementing efficient multithreading languages,” in *Proc. ACM SIGPLAN 1997 Conference on Programming Language Design and Implementation*, 1997, pp. 320–333.

[69] C. Tsolakis, P. Thomadakis, and N. Chrisochoides, “Tasking framework for adaptive speculative parallel mesh generation,” *The Journal of Supercomputing*, vol. 78, pp. 1–32, 2022.

[70] N. Chrisochoides, “Parallel run-time system for adaptive mesh refinement,” in *Proc. Solving Irregularly Structured Problems in Parallel*, 1998, pp. 396–405.

[71] D. Chase and Y. Lev, “Dynamic circular work-stealing deque,” in *Proc. Seventeenth Annual ACM Symposium on Parallelism in Algorithms and Architectures*, 2005, pp. 21–28.

[72] D. K. Panda, H. Subramoni, C.-H. Chu, and M. Bayatpour, “The MVAPICH project: Transforming research into high-performance mpi library for HPC community,” *Journal of Computational Science*, vol. 52, p. 101208, 2021.

[73] U. S. D. of Energy Office of Science, “ECP proxy applications,” [Online]. Available: https://proxyapps.exascaleproject.org/.

[74] A. Duran, X. Teruel, R. Ferrer, X. Martorell, and E. Ayguade, “Barcelona openMP tasks suite: A set of benchmarks targeting the exploitation of task parallelism in openmp,” in *Proc. International Conference on Parallel Processing*, 2009, pp. 124–131.

[75] “Sw4lite,” [Online]. Available: https://github.com/geodynamics/sw4lite.

[76] P. A. Foteinos and N. P. Chrisochoides, “High quality real-time image-to-mesh conversion for finite element simulations,” *Journal of Parallel and Distributed Computing*, vol. 74, no. 2, pp. 2123–2140, 2014.

[77] A. N. Chernikov and N. P. Chrisochoides, “Three-dimensional delaunay refinement for multi-core processors,” in *Proc. International Conference on Supercomputing*, 2008, pp. 214–224.

[78] J. Ang, A. A. Chien, S. D. Hammond, A. Hoisie, I. Karlin, S. Pakin, J. Shalf, and J. Vetter, “Reimagining codesign for advanced scientific computing: Report for the ASCR workshop on reimagining codesign,” *Technical Report. USDOE Office Of Science*, 2021.

[79] N. Chrisochoides, “Multithreaded model for the dynamic load-balancing of parallel adaptive PDE computations,” *Applied Numerical Mathematics*, vol. 20, no. 4, pp. 349–365, 1996.

[80] P. Thomadakis and N. Chrisochoides, “Toward runtime support for unstructured and dynamic exascale-era applications,” *The Journal of Supercomputing*, no. 38, pp. 4675–4695, 2023. [Online]. Available: https://doi.org/10.1007/s11227-022-05023-z

[81] P. P. Laboratory-UIUC, “Charm++ jacobi3D proxy,” [Online]. Available: https://github.com/UIUC-PPL/charm/tree/jchoi/hips22/examples/charm++/cuda/gpudirect/jacobi3d/mpi.

[82] M. Mestayer *et al.*, “The CLAS12 drift chamber system,” *Nucl. Instrum. Meth. A*, vol. 959, p. 163518, 2020.

[83] V. Ziegler *et al.*, “The CLAS12 software framework and event reconstruction,” *Nucl. Instrum. Meth. A*, vol. 959, p. 163472, 2020.

[84] P. Geurts, D. Ernst, and L. Wehenkel, “Extremely randomized trees,” *Mach. Learn.*, vol. 63, no. 1, p. 3–42, Apr. 2006.

[85] J. R. Quinlan, “Induction of decision trees,” *Mach. Learn.*, vol. 1, no. 1, pp. 81–106, 1986.

[86] T. K. Ho, “Random decision forests,” in *Proc. 3rd international conference on document analysis and recognition*, vol. 1, 1995, pp. 278–282.

[87] T. Chen and C. Guestrin, “XGBoost: A scalable tree boosting system,” in *Proc. 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, 2016, pp. 785–794.

[88] S. Haykin, “Neural networks: A comprehensive foundation.” Prentice Hall PTR, 1994.

[89] V. Nair and G. E. Hinton, “Rectified linear units improve restricted boltzmann machines,” in *Proc. Int. Conf. on Mach. Learn.*, 2010, p. 807–814.

[90] L. C. Jain and L. R. Medsker, "Recurrent neural networks: Design and applications." CRC Press, Inc, 1999.

[91] K. Cho *et al.*, "Learning phrase representations using rnn encoder-decoder for statistical machine translation," *arXiv:1406.1078*, 2014.

[92] D. E. Rumelhart, G. E. Hinton, and R. J. Williams, "Learning internal representations by error propagation," in *Proc. Parallel Distributed Processing: Explorations in the Microstructure of Cognition, Vol. 1: Foundations*, Cambridge, MA, USA, 1986, pp. 318–362.

[93] A. Ng, "Sparse autoencoder," *Stanford CS294A Lecture notes*, vol. 72, no. 2011, pp. 1–19, 2011.

[94] S. Rifai, P. Vincent, X. Muller, X. Glorot, and Y. Bengio, "Contractive auto-encoders: Explicit invariance during feature extraction," in *Proc. 28th International Conference on International Conference on Machine Learning*, 2011, pp. 833–840.

[95] J. Xie, L. Xu, and E. Chen, "Image denoising and inpainting with deep neural networks," *Advances in Neural Information Processing Systems*, vol. 25, pp. 341–349, 2012.

[96] D. P. Kingma and M. Welling, "An introduction to variational autoencoders," *Foundations and Trends in Machine Learning*, vol. 12, no. 4, pp. 307–392, 2019.

[97] I. T. Jolliffe, "Principal component analysis: A beginner's guide — introduction and application," *Weather*, vol. 45, no. 10, pp. 375–382, 1990.

[98] W. Wang, Y. Huang, Y. Wang, and L. Wang, "Generalized autoencoder: A neural network framework for dimensionality reduction," in *Proc. IEEE conference on computer vision and pattern recognition workshops*, 2014, pp. 490–497.

[99] A. Krizhevsky and G. E. Hinton, "Using very deep autoencoders for content-based image retrieval." in *Proc. European Symposium on Artificial Neural Networks*, 2011.

[100] M. Sakurada and T. Yairi, "Anomaly detection using autoencoders with nonlinear dimensionality reduction," in *Proc. 2nd Workshop on Machine Learning for Sensory Data Analysis*, 2014, p. 4–11.

[101] F. Murtagh, "Multilayer perceptrons for classification and regression," *Neurocomputing*, vol. 2, no. 5, pp. 183–197, 1991.

[102] Y. LeCunn, Y. Bengio, and G. Hinton, “Deep learning,” *Nature*, vol. 521, no. 7553, pp. 436–444, 2015.

[103] J. Han and C. Moraga, “The influence of the sigmoid function parameters on the speed of backpropagation learning,” in *Proc. International Workshop on Artificial Neural Networks: From Natural to Artificial Neural Computation*, 1995, pp. 195–201.

[104] T. Dozat, “Incorporating Nesterov momentum into adam,” in *Proc. International Conference on Learning Representations, Workshop Track*, 2016.

[105] S. Stepanyan *et al.*, “CLAS12 FD charge particle reconstruction efficiency and the beam background merging,” *CLAS12-NOTE, 2020-005*, 2020.

[106] G. Gavalian, “Auto-encoders for Track Reconstruction in Drift Chambers for CLAS12,” arXiv:2009.05144, 2020.

[107] P. Thomadakis, A. Angelopoulos, G. Gavalian, and N. Chrisochoides, “Using machine learning for particle track identification in the CLAS12 detector,” *Computer Physics Communications*, vol. 276, p. 108360, 2022.

[108] A. P. Bradley, “The use of the area under the ROC curve in the evaluation of machine learning algorithms,” *Pattern Recogn.*, vol. 30, no. 7, pp. 1145–1159, 1997.

[109] F. Pedregosa *et al.*, “Scikit-learn: Machine learning in python,” *Journal of machine learning research*, vol. 12, pp. 2825–2830, 2011.

[110] M. Abadi *et al.*, “Tensorflow: A system for large-scale machine learning,” in *Proc. 12th USENIX Symposium on Operating Systems Design and Implementation*, 2016, pp. 265–283.

[111] D. Kingma and J. Ba, “Adam: A method for stochastic optimization,” in *Proc. International Conference on Learning Representations*, 2015.

[112] N. Srivastava, G. Hinton, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov, “Dropout: A simple way to prevent neural networks from overfitting,” *J. Mach. Learn. Res.*, vol. 15, no. 1, pp. 1929–1958, 2014.

[113] B. Xu, N. Wang, T. Chen, and M. Li, “Empirical evaluation of rectified activations in convolutional network,” arXiv:1505.00853, 2015.

[114] S. K. Hinton Geoffrey, Srivastava Nitish, "Neural networks for machine learning," Online Course Lecture. [Online]. Available: https://www.coursera.org/learn/neural-networks/home/welcome.

[115] P. Thomadakis, A. Angelopoulos, G. Gavalian, and N. Chrisochoides, "De-noising drift chambers in CLAS12 using convolutional auto encoders," *Comput. Phys. Commun.*, vol. 271, p. 108201, 2022.

[116] S. Agostinelli *et al.*, "Geant4—a simulation toolkit," *Nuclear Instruments and Methods in Physics Research Section A: Accelerators, Spectrometers, Detectors and Associated Equipment*, vol. 506, no. 3, pp. 250–303, 2003.

[117] M. Ungaro, "Geant based monte-carlo," [Online]. Available: https://gemc.jlab.org/gemc/. [Online]. Available: https://gemc.jlab.org/gemc/

[118] M. Khachatryan *et al.*, "Electron-beam energy reconstruction for neutrino oscillation measurements," *Nature*, vol. 599, no. 7886, pp. 565–570, 2021.

[119] P. Chatagnon *et al.*, "First measurement of timelike compton scattering," *Phys. Rev. Lett.*, vol. 127, p. 262501, 2021.

[120] A. E. Alaoui, N. Baltzell, and K. Hafidi, "A RICH detector for CLAS12 spectrometer," in *Proc. 2nd International Conference on Technology and Instrumentation in Particle Physics*, vol. 37, 2012, pp. 773–780.

[121] P. Thomadakis, A. Angelopoulos, G. Gavalian, and N. Chrisochoides, "De-noising drift chambers in clas12 using convolutional auto encoders," *Computer Physics Communications*, vol. 271, p. 108201, 2022.

[122] S. J. Reddi, S. Kale, and S. Kumar, "On the convergence of adam and beyond," arXiv:1904.09237, 2019.

[123] G. Gavalian, P. Thomadakis, A. Angelopoulos, N. Chrisochoides, R. De Vita, and V. Ziegler, "CLAS12 track reconstruction with artificial intelligence," arXiv:2202.06869, 2022.

VITA

Polykarpos Thomadakis
Department of Computer Science
Old Dominion University
Norfolk, VA 23529

EDUCATION

- Ph.D. Candidate, Department of Computer Science, Old Dominion University, Norfolk, VA, 23529 (2023)
- Bachelor of Science, Department of Computer Engineering, Volos, Greece (2016)

JOURNAL PUBLICATIONS

- P. Thomadakis, K. Garner, G. Gavalian and N. Chrisochoides. "Charged Particle Reconstruction in CLAS12 using Machine Learning". *Computer Physics Communications*, 2023. (Impact Factor: 4.7)
- P. Thomadakis and N. Chrisochoides. "Runtime Support for Unstructured Exascale-era Applications". *J. of Supercomputing*, 2023. (IF: 2.5)
- P. Thomadakis, C. Tsolakis and N. Chrisochoides. "Multithreaded Runtime Framework for Parallel and Adaptive Applications". *Eng. with Computers*, 2022. (IF: 8.0)
- P. Thomadakis, A. Angelopoulos, G. Gavalian and N. Chrisochoides. "Using Machine Learning for Particle Track Identification in the CLAS12 Detector". *Computer Physics Communications*, 2022. (IF: 4.7)
- P. Thomadakis, A. Angelopoulos, G. Gavalian, V. Ziegler and N. Chrisochoides. "Denoising Drift Chambers in CLAS12 Using Convolutional Autoencoders". *Computer Physics Communications*, 2022. (IF: 4.7)
- P. Thomadakis and N. Chrisochoides. "Runtime Support for Efficient Handling of Multi-device Heterogeneous Platforms". *J. of Supercomputing*, Submitted. (IF: 2.5)