Performance Analysis and Improvement for Scalable and Distributed Applications Based on Asynchronous Many-Task Systems

Nanmiao Wu
Louisiana State University and Agricultural and Mechanical College

Follow this and additional works at: https://digitalcommons.lsu.edu/gradschool_dissertations
Part of the Other Computer Sciences Commons, and the Software Engineering Commons

Recommended Citation
https://digitalcommons.lsu.edu/gradschool_dissertations/5781

This Dissertation is brought to you for free and open access by the Graduate School at LSU Digital Commons. It has been accepted for inclusion in LSU Doctoral Dissertations by an authorized graduate school editor of LSU Digital Commons. For more information, please contact gradetd@lsu.edu.
PERFORMANCE ANALYSIS AND IMPROVEMENT FOR SCALABLE AND DISTRIBUTED APPLICATIONS BASED ON ASYNCHRONOUS MANY-TASK SYSTEMS

A Dissertation

Submitted to the Graduate Faculty of the Louisiana State University and Agricultural and Mechanical College in partial fulfillment of the requirements for the degree of Doctor of Philosophy

in

Division of Electrical and Computer Engineering

by

Nanmiao Wu
B.S., Nankai University, 2013
M.S., University of Macau, 2016
May 2022
Acknowledgements

I would like to express my most sincere gratitude and appreciation to my advisor, Dr. Hartmut Kaiser, for his continuous support, invaluable guidance, and encouragement throughout the development of the work. Without his support this work would not have been produced in this shape. I would like to thank Dr. J. (Ram) Ramanujam for being in my committee chair and for his support through my time at Center for Computation and Technology at LSU. I would like to thank my committee members Dr. Jianhua Chen, Dr. Golden G Richard III, Dr. Jerry L. Trahan, and Dr. Jerome W. Hoffman, whose expertise, assistance, and support are invaluable in the development of the work.

I would like to thank my collaborators Dr. Vito Giovanni Castellana and Dr. Andrew Lumsdaine at Pacific Northwest National Laboratory for their thoughtful suggestions and help for my research.

I would like to thank my dear colleagues and friends at STE||AR group, Adrian, Ali, Avah, Bibek, Bita, Chuanqiu, Dominic, Ioannis, Karame, Katie, Max, Nikunj, Parsa, Patrick, Rod, Shahrzad, Steve, Tianyi, Weile, for all their constructive discussions and valuable suggestions and also other researchers I had the opportunity to collaborate with.

My deep gratitude to my mother and father for their continuous love and support. They encourage me to look up at the stars and navigate my own adventure. I would like to thank my husband Wentao, for his unending support on this journey. I would like to thank my best friend Linyin, for being my rock and my inspiration.
# Table of Contents

ACKNOWLEDGEMENTS ................................................................. ii

LIST OF FIGURES ................................................................. v

ABSTRACT ................................................................................... viii

CHAPTER
1 INTRODUCTION ........................................................................ 1
   1.1 Research Contributions ................................................... 4
   1.2 Dissertation Outline ....................................................... 5

2 BACKGROUND .......................................................................... 6

3 TASK BENCH ............................................................................ 10
   3.1 Introduction ....................................................................... 10
   3.2 Generic HPX Implementation ........................................... 12
   3.3 Improved HPX Implementation ......................................... 15
   3.4 Experimental Results ..................................................... 18
   3.5 Summary .......................................................................... 27

4 SHAD ...................................................................................... 29
   4.1 Introduction ....................................................................... 29
   4.2 Abstract Runtime Interface ............................................. 29
   4.3 Performance Results ....................................................... 36
   4.4 Summary .......................................................................... 44

5 PHYLANX ................................................................................. 46
   5.1 Introduction ....................................................................... 46
   5.2 Distributed Computing in Phylanx ..................................... 46
   5.3 Alternating Least Square (ALS) Algorithm ....................... 49
   5.4 Distributed ALS in Phylanx .............................................. 51
   5.5 Performance Results ....................................................... 54
   5.6 Summary .......................................................................... 56

6 SECOND-ORDER OPTIMIZATION ............................................. 58
   6.1 Introduction ....................................................................... 58
   6.2 Hessian-free Newton Method .......................................... 59
   6.3 Performance Results ....................................................... 59
   6.4 Summary .......................................................................... 62

7 CONCLUSION ............................................................................ 63

APPENDIX. COPYRIGHT INFORMATION .................................... 66

REFERENCES .............................................................................. 67
List of Figures

2.1 HPX runtime system [1] ................................................................. 8

3.1 The stencil pattern, vertices denote tasks and edges between vertices represents task dependencies.................................................. 11

3.2 The stencil pattern with two HPX localities, dependencies within the same node use on-node sharing while dependencies across the node require inter-node communication...................................... 12

3.3 The stencil pattern with two HPX localities, a parallel execution policy is specified with a hpx::for_loop() to explore the parallelism. ............ 13

3.4 A global barrier, one core does communication sequentially and after that each core does computational task in parallel............................ 14

3.5 The stencil pattern with two HPX localities, hpx::fork_join_executor is utilized to reduce HPX threads re-creation overheads...................... 16

3.6 Overlapping communication and computation. ................................. 18

3.7 Enabling work-stealing policy for overdecomposition scenario. ............ 19

3.8 The Stencil pattern, 1 node (48 cores), 48 tasks. .............................. 21

3.9 The stencil pattern, 4 nodes (48 cores per node), 192 cores, 192 tasks. ................................................................................................. 22

3.10 The stencil pattern, 8 nodes (48 cores per node), 384 cores, 384 tasks. ......................................................................................... 23

3.11 The stencil pattern, weak scaling performance, 48 cores per node, 48 tasks per node, each colorful line \(2^N\) represents different grain sizes, where \(N = 4, 6, ... 26\). ......................................................... 25

3.12 METG of each system with varying number of nodes for different overdecomposition, lower is better, flat is ideal............................ 26

4.1 SHAD three layers [1] ................................................................. 30

4.2 SHAD+HPX........................................................................... 34

4.3 Speedup of parallel policy vs. sequential policy for SHAD algorithms with the proposed design (SHAD with HPX support), varying the size of shad::array (integers), 1 node (32 cores). ................. 37
4.4 Strong scaling, comparison between SHAD with HPX support, SHAD with GMT support, and SHAD with TBB support, using `shad::transform` on `shad::array`, varying the number of cores with 100 Millions elements (integers), 1 node (32 cores). .......................... 38

4.5 Performance of `shad::reduce` algorithm on `shad::array`, varying the number of elements (integers), comparing the performance of the proposed design (SHAD with HPX support) with that of when SHAD are supported by TBB and GMT, respectively, 1 node (32 cores). .............................................................. 38

4.6 Performance of `shad::count_if` algorithm on `shad::array`, varying the number of elements (integers), comparing the performance of the proposed design (SHAD with HPX support) with that of when SHAD are supported by TBB and GMT, respectively, 1 node (32 cores). .............................................................. 39

4.7 Strong scaling of the proposed design (SHAD with HPX support), using `shad::array` for SHAD algorithms, varying the number of cores, for 20 Millions elements (integers). ......................... 40

4.8 Strong scaling of the proposed design (SHAD with HPX support), using `shad::array` for SHAD algorithms, varying the number of cores, for 100 Millions elements (integers). ............................. 41

4.9 Weak scaling of the proposed design (SHAD with HPX support), applying SHAD algorithms on `shad::array` using parallel policy when each node (32 cores) processes 1 Billion elements (integers). ................................................................. 42

4.10 Weak scaling, comparison of proposed design (SHAD with HPX support) and SHAD with GMT support, applying `shad::reduce` on `shad::array` using parallel policy, varying node count with 1 Billion elements (integers) per node. .......................................................... 43

4.11 Strong scaling of the proposed design (SHAD with HPX support), applying `shad::array` for SHAD algorithms using parallel policy with 1 Billion elements (integers). ................................. 43

4.12 Strong scaling, comparison between SHAD with HPX support and SHAD with GMT support, applying `shad::count_if` on `shad::array` using parallel policy, varying node count with 1 Billion elements (integers). .................................................. 44

5.1 Phylanx software stack [2]. ................................................................. 47
5.2 Different tilings in Phylanx .................................................. 48
5.3 Distributed diagonal primitive with 2 localities, each locality contains a 2-D array with original distributed vector on the specified k-th diagonal after applying the distributed diagonal primitive. ...... 49
5.4 Distributed all gather primitive with 2 localities, each locality is able to have the whole array after applying the distributed all gather primitive, where row tiling is selected on the distributed array. ........ 50
5.5 Matrix factorization, factoring the rating matrix into two much smaller matrices. .......................................................... 50
5.6 Distributed ALS in Phylanx where 2 localities are available, two copies of rating matrix with different tilings. ........................................ 52
5.7 Comparison of distributed ALS (Phylanx implementation) and non-distributed ALS NumPy implementation on part of MovieLens 20M dataset, which contains 4800 movies and 4800 users, varying the number of threads, 1 node (48 cores per node). ............... 55
5.8 Comparison of distributed ALS (Phylanx implementation) and non-distributed ALS NumPy implementation on part of MovieLens 20M dataset, which contains 4800 movies and 4800 users, varying the number of nodes (48 cores per node). ............................. 56
5.9 Performance of distributed ALS on different sizes of dataset, where the first number denotes the number of movies and second number denotes the number of users, varying the number of nodes (48 cores per node). .................................................. 57
6.1 Comparison of convergences of Krylov-Newton (KN) method and SGD optimizer on EMNIST dataset for an image classification task. .......................................................... 60
6.2 Comparison of throughput of Krylov-Newton (KN) method and SGD optimizer on EMNIST dataset for an image classification task, varying the number of threads. ............................... 61
6.3 Performance of Krylov-Newton (KN) method and SGD optimizer on EMNIST dataset for an image classification task, varying the number of process. .................................................. 61
Abstract

As the complexity of recent and future large-scale data and exascale systems architectures grows, so do productivity, portability, software scalability, and efficient utilization of system resources challenges presented to both industry and the research community. Software solutions and applications are expected to scale in performance on such complex systems. Asynchronous many-task (AMT) systems, taking advantage of multi-core architectures with light-weight threads, asynchronous executions, and smart scheduling, are showing promise in addressing these challenges.

In this research, we implement several scalable and distributed applications based on HPX, an exemplar AMT runtime system. First, a distributed HPX implementation for a parameterized benchmark Task Bench is introduced. The performance bottleneck is analyzed where the repeated HPX threads creation costs and a global barrier for all threads limit the performance. The methodologies to retain the spawning threads alive and overlap communication and computation are presented. The evaluation results prove the effectiveness of the improved approach, where HPX is comparable with the prevalent programming models and takes advantages of multi-task scenarios. Second, an algorithms and data-structures SHAD library with HPX support is introduced. The methodologies to support local and remote operations in synchronous and asynchronous manners are developed. The HPX implementation in support of the SHAD library is further provided. Performance results demonstrate that the proposed system presents the similar performance as SHAD with Intel TBB (Threading Building Blocks) support for shared-memory parallelism and is better to explore the distributed-memory parallelism than SHAD with GMT (Global Memory and Threading) support. Third, an asynchronous array processing framework Phylanx is introduced. The methodologies that support a distributed alternating least square algorithm are developed. The implementation of this algorithm along with a number of distributed primitives are provided. The performance results show that Phylanx implementation presents a good scalability. Finally, a scalable second-order method for
optimization is introduced. The implementation of a Krylov-Newton second-order method via PyTorch framework is provided. Evaluation results illustrate the effectiveness of scalability, convergence, and robust to hyper-parameters of the proposed method.
Chapter 1
Introduction

The up-rise of multi and many-core architectures has resulted in the challenges of developing efficient parallel programming techniques to effectively utilize the computing systems. Many scientific HPC (High Performance Computing) applications utilize a hybrid programming model, where MPI [3], an abbreviation for Message Passing Interface, is extensively used for inter-node communication and OpenMP [4], an abbreviation for Open Multi-processing, is a threading library to explore intra-node parallelism. In such traditional programming environments, the burdens of expressing parallelism from the algorithms, managing computational resources and communications, and mapping the computational tasks to the underlying hardware resources are on the software developers. These burdens become heavier as the number of cores and nodes, the application size, and heterogeneity of computational resources increase [5].

Asynchronous many-task (AMT) systems, taking advantage of multi-core architectures with light-weight threads, asynchronous executions, and smart scheduling, are gaining increasing popularity in the HPC community. These runtime systems relieve the above mentioned burdens on software developers. Instead, software developers are expected to express parallelism of their applications by forming dataflow execution trees that generate tasks which would be executed in the proper order. Further, AMT systems have shown promise towards effective utilization of available concurrency [6].

HPX is a general purpose C++ AMT runtime system for parallel and distributed applications of any scale [7]. It represents an innovative mixture of a global system-wide address space, overlapping communication with computation, moving work to data instead of data to work, oversubscribing execution resources, efficient fine-grained parallelization, minimal-overhead synchronization and context switching. In this work, I will analyze and improve the performance of several HPC applications that are based on the HPX runtime system to demonstrate an alternative to the prevalent programming models in the HPC
community.

Task Bench [8] is a parameterized benchmark to evaluate the performance of distributed programming systems. There are 15 parallel and distributed programming systems implemented in Task Bench, e.g. MPI, OpenMP, and MPI+OpenMP. With a core API, Task Bench enables the evaluation of n systems for m benchmarks with \(O(m + n)\) implementation effort, rather than the \(O(m \cdot n)\) effort required by other benchmark libraries, which significantly reduces the programming efforts to evaluate new systems and benchmarks. Task Bench benchmarks are defined by task graphs, which are the combination of an iteration space with a dependence relation, expressing communication and task dependency patterns common in real-world applications. The task graph is described by command-line parameters passed to invocations of Task Bench, and is generated at runtime. Such task graph is then executed within the parallel system, with key performance metrics being tracked by the Task Bench system. Task Bench introduces a novel metric, minimum effective task granularity to effectively compare the overheads of the systems, since weak scaling is possible to hide the system overheads for a larger size problem while strong scaling may fail to separate system overheads from application costs. In this work, a distributed HPX implementation is first implemented. The performance bottleneck and system overheads are then measured and analyzed. The methodology to further exploit the parallelism with reduced overhead is developed. Finally an improved implementation is presented and the performance is compared with other programming systems.

SHAD [9] is a high-performance algorithms and data-structures library, providing general purpose building blocks and supporting high-level custom utilities. SHAD is designed with scalability, flexibility, productivity, and portability in mind, and serves as a playground for research in parallel programming models, runtime systems, and their applications [1]. SHAD’s portability is achieved through the abstract runtime interface, which decouples the upper layers of the library and hides the low level details of the underlying architecture. This layer enables SHAD to be supported by different platforms, via different
runtime systems, e.g. Intel TBB (Threading Building Blocks) [10] and global memory and threading (GMT) [11]. However, current backends have limitations. TBB is only suitable for shared-memory parallelism only while GMT targeting distributed systems, relies on a centralized controller which limits scalability up to hundreds of nodes and creates a network hot spot due to all to one communication for synchronization, resulting in degraded performance at high process counts [1]. In this work, we provide HPX as an additional backend in support of the SHAD library and explore the scalability and performance of SHAD when being supported by HPX. The methodologies that support local and remote execution in synchronously or asynchronous manners are developed. The implementations of HPX support are provided. The performance of the proposed design is evaluated and further compared with existing backends of SHAD.

Phylanx, is an asynchronous array processing framework which automatically transforms user-provided Python code into an intermediate representation that is efficiently executed and distributed across all available compute resources as specified by the user [12]. Phylanx’s software framework is based on the HPX runtime system, which implicitly benefits from fine-grained parallelism, constrained-based synchronization, adaptive locality control, latency hiding, message driven computation, and overlapping computation with communication. In this work, the distributed computing in Phylanx is first introduced. The methodologies that support a distributed alternating least square (ALS) algorithm for recommendation system via Phylanx are developed. The implementations of this distributed algorithm along with a number of distributed primitives are provided. The performance of the distributed Phylanx implementation is evaluated and further compared with a non-distributed NumPy implementation.

Second-order optimization algorithms can be used for scalable deep learning training, enabling larger and more complex models to be trained in significantly less time [13]. There is significant difficulty with a naïve second-order method, i.e. Newton’s method. This is because forming the complete Hessian matrix is not feasible due to the storage require-
ments. In some cases, it is not even possible to compute or represent the Hessian matrix. This inspired us to use improved second-order methods, e.g., the Krylov-Newton method which combines Newton’s method with a Krylov solver for the linear solution [13–15]. In this work, the advantages and disadvantages of first-order and second-order methods are analyzed. An Hessian-free Krylov-Newton optimizer is implemented on both PyTorch [16] and NumPy [17]. The convergence behavior of the Hessian-free Krylov-Newton optimizer is evaluated and compared with a popular first-order optimizer. In addition, the scalable performance of the Hessian-free Krylov-Newton optimizer with different number of threads and processes is further characterized.

1.1 Research Contributions

This dissertation makes the following contributions:

- Extending the Task Bench library by adding a distributed HPX implementation with overlapping of communication and computation, and reduced overheads.

- Comparing the HPX implementation with main stream MPI and MPI + OpenMP libraries, where HPX implementation is comparable with them and takes advantages of multi-task scenarios.

- Providing a runtime alternative for the SHAD library by developing the methodologies and implementations to support local and remote execution in synchronous and asynchronous manners on the top of the HPX runtime system.

- Comparing the SHAD with HPX support against existing TBB and GMT backends, where SHAD with HPX presents similar performance as TBB and is better able to exploit shared and distributed-memory parallelism than GMT.

- Extending the Phylanx framework by adding a distributed recommendation algorithm along with a number of distributed primitives necessary for its implementation.
• Offering an alternative second-order optimizer for scalable deep learning training with reduced training time and tuning efforts.

1.2 Dissertation Outline

The remainder of this dissertation is structured as follows. Chapter 2 provides additional background information along with an overview of the HPX runtime system. Chapter 3 presents the Task Bench details, performance analysis for a basic HPX implementation along with an improved version with reduced overheads. Chapter 4 provides the introduction of the SHAD library along with the methodologies for supporting the local and remote execution in synchronously or asynchronously manners with respect to the HPX runtime. Chapter 5 presents an overview of the Phylanx framework, a distributed recommendation algorithm in Phylanx along with a set of distributed primitives. Chapter 6 discusses the advantages and disadvantages of first-order and second-order methods for scalable deep learning training along with a performance comparison between them. Chapter 7 concludes the dissertation.
Chapter 2
Background

HPC clusters are capable of tackling problems with a large number of processors, where these processors work simultaneously in order to reduce the total computational time. Scalability is used to measure the parallelization efficiency of the application, which is a proportion between the actual speedup and the ideal speedup when a specific number of processors are used. Generally, there are two forms of scalability, i.e. strong scaling and weak scaling.

Strong scaling is defined as how the execution time varies with the number of processors used for a fixed size problem. For example, when doubling the number of processors while keeping the problem size fixed, an ideal speedup is expected to be a factor of two. However, Amdahl’s law (strong scaling) [18, 19] states that the theoretic upper bound of a speedup for a fixed size problem is determined by the serial portion of the code, defined as

$$speedup = 1/(s + p/N),$$  \hspace{1cm} (2.1)

where $s$ is the portion of the execution time on the serial code, $p$ is the portion of the execution time on the parallel code, and $N$ is the number of processors.

Weak scaling is defined as how the execution time varies with the number of processors where each processor has the same size problem. For example, when doubling the number of processors, an ideal speedup is expected to handle a double size problem using the same execution time. However, Gustafson’s law (weak scaling) [20] states that the parallel code scales linearly with the number of processors while the serial code does not, and there is no limit for the scaled speedup, defined as

$$speedup = s + p \times N,$$  \hspace{1cm} (2.2)

where $s$, $p$, and $N$ have the same meaning as in Amdahl’s law.
While strong and weak scaling provide theoretical limits of speedups, there are more factors that limit the scalability and make our system SLOW: a) Starvation, \textit{i.e.} insufficient available concurrent work, b) Latencies, \textit{i.e.} time-distance delay due to access remote resources, c) Overhead, \textit{i.e.} extra work for management of parallel actions and resources on the critical execution path which is not necessary in the sequential variant, and d) Waiting for contention, \textit{i.e.} delays due to lack of availability of oversubscribed resources [21].

The HPX AMT runtime system is promising to alleviate the limitations by making the following efforts [7]:

- **Focus on latency hiding instead of latency avoidance, \textit{i.e.}** when a thread has to wait for some resources to finish its work, it would do other unrelated job in the mean time, allowing the system to hide the latencies from the user by filling the idle time by useful work.

- **Embrace fine-grained parallelism instead of heavy weight threads, \textit{i.e.}** HPX utilizes light-weight threads with extremely short context switching times. A task whose preconditions are not met would be suspended, and HPX threads would seamless switch to other tasks that can be executing in the meantime. HPX threads would re-continue on the initial task when the preconditions are met.

- **Rediscover constraint-based synchronization to replace global barriers, \textit{i.e.}** it is unnecessary to wait for iterations of a loop to finish before a thread can do other work, and it is sufficient to synchronize just a small subset of the threads whose preconditions are not met while proceeding other operations whenever their preconditions are satisfied.

- **Adaptive locality control instead of static data distribution, \textit{i.e.}** HPX provides a global and uniform address space, which supports fully dynamic data migration and dynamic load balance via migrating part of the data to different localities.
- Prefer moving work to the data over moving data to the work, *i.e.* HPX tries to minimize data transfer between different localities of a cluster and prefers to execute the code close to the locality which contains the data.

- **Favor message driven computation over message passing, *i.e.*** HPX allows for sending messages without requiring the receiver to actively wait for them via triggering actions and possibly continuations. Together with work-queue based scheduling, HPX is able to overlap communication with computation and thereby minimizes latencies.

HPX is widely portable on various platforms and systems, providing an unified and C++ standard-conforming API. HPX has been introduced in detail in work [22–25]. The main components of HPX are listed as follows and the sketch of HPX’s architecture is shown as in Figure 2.1.

Figure 2.1. HPX runtime system [1]

- **Threading Subsystem**, manages the lightweight threads which are scheduled on top of OS threads and have reduced latencies due to extremely short context switching times. HPX provides a set of scheduling policies to execute HPX threads, *e.g.* work-stealing and work-sharing policies, ensuring automatic load balancing of tasks.
This is important to achieve high resource utilization and good scalability of an user application.

- **Local Control Objects**, is a class of synchronization functions that support C++20 standard-conforming primitives to synchronize the execution of different threads, *e.g.* `hpx::barrier`, and asynchronous methods such as `hpx::async` and `hpx::future`.

- **Active Global Address Space (AGAS)**, spans a global address space over all localities and enables a uniform API for local and remote execution. This layer assigns a unique Global Identifier (GID) to each HPX object for tracking. When an HPX object is migrating from one locality to another, its GID does not change. AGAS would automatically and transparently resolve the address, which guarantees the load balancing.

- **Parcel Transport Layer**, implements active messages that are used to encapsulate remote calls. The migration of an HPX object from one locality to its destination locality is done through the network. In addition, this layer implicitly overlaps communication with computation.

- **Performance Counter Framework**, provides a way to monitor system metrics via performance counters which can be easily queried by users at runtime. Users can also create own counters to monitor metrics for their applications.

HPX represents an innovative mixture of a global system-wide address space, overlapping communication with computation, moving work to data instead of data to work, oversubscribing execution resources, efficient fine-grained parallelization, minimal-overhead synchronization and context switching [7]. In later chapters, we explore the scalability, analyze and improve the performance of scalable and distributed HPC applications that are based on the HPX runtime system.
Chapter 3
Task Bench

3.1 Introduction

Task Bench [8] is a parameterized benchmark for evaluating the performance of distributed programming systems. There are 15 parallel and distributed programming systems are implemented in Task Bench, i.e. Chapel [26], Charm++ [27], MPI [3], OpenMP [4], MPI + OpenMP, Dask [28], OmpSs [29], PaRSEC [30], Realm [31], Regent [32], Spark [33], StarPU [34], Swift/T [35], TensorFlow [36], and X10 [37]. The key property of Task Bench is that it successfully isolates the system-specific implementation from the implementation of the benchmarks themselves [8]. For example, if there are $n$ benchmarks and $m$ programming systems, the total effort including developing, debugging, testing, and tuning parameters is $O(m \cdot n)$. In practice, few work actually evaluate a large number of systems. Task Bench reduces this effort to $O(m + n)$ via introducing a core Task Bench API. In this way, new benchmarks are then implemented in terms of this core Task Bench API and each new system only needs to utilize this core Task Bench API, which dramatically reduces the programming effort required to evaluate new benchmarks and systems.

Task Bench can explicitly model various application behaviors by formulating each application as a task graph. A task graph is the combination of an iteration space with a dependence relation. The iteration space is assumed to be 2-dimensional where time is along the vertical axis and tasks are along horizontal axis. An example of task graph is shown in Figure 3.1. For example, in a stencil pattern, each vertex is connected with its two immediate neighbors, defined by the dependence relation $D(t, i) = \{i, i - 1, i + 1\}$, where $t$ denotes time step and $i$ represents vertex. Vertices denote tasks and edges between vertices represents task dependencies. Task Bench is able to model different applications by parameterizing the task graph, e.g. the height and width of the task graph, the dependencies between tasks, the type of kernel (compute-bound or memory-bound), etc. The task graph is described by command-line parameters passed to invocations of Task Bench,
and is generated at runtime. This task graph is then executed within the parallel system, with key performance metrics being tracked by the Task Bench system.

![Diagram ofStencil Pattern](image)

Figure 3.1. The stencil pattern, vertices denote tasks and edges between vertices represents task dependencies.

Strong and weak scaling, the most used measures for evaluating system performance, are possible to lead to inaccurate results. Strong scaling may fail to set apart the overheads from application costs while weak scaling can hide the runtime overheads for a large enough grain size. Task Bench introduces a new metric, METG (*Minimum Effective Task Granularity*), which essentially indicates the scaling capabilities of the system on target. METG exposes how high the computing performance, usually measured in FLOP/s (Floating Point Operations Per Second), can be maintained as the amount of work decreases. The reasoning behind METG is that for larger task granularity, almost all systems are expected to behave optimally while these systems likely diverge when task granularity approaches or drops below METG. In this work, we use the same choice of 50% as the Task Bench paper [8] to compare the smallest average task granularity such that each system reaches at least 50% peak efficiency.

In this work, we first introduce a generic HPX implementation for Task Bench and analyze the performance limitation. Further we present an improved version that reduces overheads. In addition, we evaluate the performance of HPX implementation and compare it with existing Task Bench implementations based on other (main stream) libraries.
### 3.2 Generic HPX Implementation

An example of stencil pattern is shown as in Figure 3.1. Each vertex is connected with its two immediate neighbors, defined by the dependence relation \( D(t, i) = \{i, i - 1, i + 1\} \), where \( t \) denotes time step and \( i \) represents vertex. Note that time is viewed as flowing from top to bottom and along horizontal dimension we have a number of vertices, \( i.e. \) tasks. The edge between the vertices denotes the task dependencies.

To explore the parallelism, a number of HPX localities can be specified. A locality in HPX describes a synchronous domain of execution, which can be a single compute node in a cluster or a NUMA domain. In this work, a compute node is used as a HPX locality. For example, two nodes are specified for above stencil pattern, see Figure 3.2, where each node is responsible for two tasks. The task dependencies within same node can use on-node sharing while the task dependencies that across node require inter-node communication.

![Cross-node communication](image)

Figure 3.2. The stencil pattern with two HPX localities, dependencies within the same node use on-node sharing while dependencies across the node require inter-node communication.

Generally, there are two events occurring at each time step. First is to exchange data either via on-node sharing or cross-node communication. Second is to execute each task using the provided core Task Bench API. In this work, We utilize HPX for the shared-memory parallelism and use MPI to handle the inter-node communication. The syntax and semantics are the same as using MPI directly, just with `hpx_main` to initialize the environment as shown in Listing 3.1.
Listing 3.1. Example code of HPX with MPI

```c
int hpx_main(int argc, char *argv[])
{
    int n_ranks, rank;
    MPI_Comm_size(MPI_COMM_WORLD, &n_ranks);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    // ... some codes here ...
    return hpx::finalize();
}
```

Figure 3.3. The stencil pattern with two HPX localities, a parallel execution policy is specified with a `hpx::for_loop()` to explore the parallelism.

A plain for loop is utilized to sequentially collect the MPI requests for sending and receiving data between HPX localities. A global barrier is placed at the end of this plain for loop to wait for all HPX threads, as shown in Fig 3.4. For instance, there are four cores in a compute node. In this case, only one core does the communication sequentially. After the sequential communication, each core does computational task in parallel. According to Amdahl’s law, the ratio of sequential time ($t_{overhead}$) and parallel work time ($t_{work}$) defines the upper bound of speedup. If $t_{overhead} / t_{work}$ is equal to 0.5, then upper bound of speedup is a factor of 2. One way to increase the speedup is to increase the parallel work time ($t_{work}$), such that the $t_{overhead} / t_{work}$ decreases, e.g. to 0.1, then
upper bound of speedup increases to a factor of 10. This is the reason that we could see a relatively good performance of this implementation as the grain size is large enough.

After all MPI requests are completed, i.e. each boundary vertex has received/sent the data from/to its neighbor in other nodes, a parallel `hpx::for_loop()` is invoked, which implements loop functionality over a range specified by integral or iterator bounds. Execution policies can be applied to describe the behavior of a `hpx::for_loop()` with allowed parallelism and optional parameters to control properties, e.g. sequential execution policy (`hpx::execution::seq`) and parallel execution policy (`hpx::execution::par`). In this scenario, a parallel execution policy `hpx::execution::par` for the scheduling of the iterations is specified, so that `hpx::for_loop()` would be run in parallel, as shown in Fig 3.3. Note that the default parallel executor that is spawning a thread for each task is used if you don’t specify one explicitly. Executors are lightweight handles controlling where (the location that the work should be executed, on this core, on that node, or on this NUMA domain), when (the work relies on the availability of a result or no restrictions apply), and how (can be run concurrently or has to be run sequentially) to execute tasks [23]. Optional parameters can be specified to control properties, e.g. static_chunk_size. With a given static_chunk_size, loop iterations are divided into chunks with size of static_chunk_size. Listing 3.2 shows how `hpx::execution::par` with user-defined parameters, e.g. a work with chunk size one is assigned to each thread, using .with() is specified in `hpx::for_loop`. 

Figure 3.4. A global barrier, one core does communication sequentially and after that each core does computational task in parallel.
Listing 3.2. Example code of `hpx::execution::par` with user-defined parameters

```cpp
1 // Use HPX parallel execution policy with user-defined chunk size where a
   work with chunk size one is assigned to each thread
2 hpx::execution::static_chunk_size fixed(1);
3 hpx::for_loop(hpx::execution::par.with(fixed), begin, end, lambda_function);
```

Invoking `hpx::for_loop()` at every time step would need to re-create HPX threads. Creating the HPX thread takes about 1 $\mu$s while each vertex executing a compute-bound kernel with a grain size of one takes about 2.5 ns. If the task graph is 1000 time steps long, HPX threads would need to be repeatedly created for 1000 times, see Fig 3.3, and this leads to large runtime overheads. Such overheads could be ignored when the grain size is large, e.g. $2^{24}$, which takes 41.9 $\mu$s for a vertex to execute a compute-bound kernel, since the overheads of creating and managing the tasks are just a small fraction of the overall computation performed by the problem. However, the overheads could dominate when the grain size is smaller, e.g. $2^6$, which takes 0.2 $\mu$s for a vertex to execute a compute-bound kernel.

The performance of this generic implementation is comparable with other systems for larger grain sizes under a single-task per core scenario. However, for smaller grain sizes, this implementation incurs quite amount of HPX thread creation overheads. Further, when overdecomposition (multi-task is assigned to each core) is considered, the performance would be further limited by the sequential plain for loop that is used to collect MPI requests, because many threads are idle at most of time. To reduce such overheads and overlap communication with computation, we implement an improved version, which is introduced in next section.

### 3.3 Improved HPX Implementation

To reduce the HPX thread re-creation overheads at each time step, `hpx::fork_join_executor` is utilized. `hpx::fork_join_executor` follows `fork-join` semantic and creates HPX threads in a way of one thread per core. It is able to retain the spawning threads alive by allocat-
ing existing work to these threads, shown as in Figure 3.5. Listing 3.4 shows that hpx::execution::par is rebound to a hpx::fork_join_executor using .on(), also with user-defined parameters. Using hpx::fork_join_executor could significantly improve the performance in comparison to that using hpx::execution::par only. Because invoking hpx::for_loop using fork_join_executor would only create HPX threads once (one thread per core) and retains them for all the time steps.

Figure 3.5. The stencil pattern with two HPX localities, hpx::fork_join_executor is utilized to reduce HPX threads re-creation overheads.

Listing 3.3. Example code of hpx::fork_join_executor with disabled work-stealing policy.

```cpp
// Rebind parallel execution policy to fork_join_executor with user-defined parameters
using executor = hpx::execution::experimental::fork_join_executor;
executor exec(hpx::threads::thread_priority::normal,
             hpx::threads::thread_stacksize::small_,
             executor::loop_schedule::static_,
             std::chrono::microseconds(100));
// A single task is assigned to each core
hpx::execution::static_chunk_size fixed(1);
hpx::for_loop(hpx::execution::par.on(exec).with(fixed),
              first, end, lambda_function);
```
Further, `hpx::fork_join_executor` offers more ability and flexibility, where users can determine the following parameters:

- The priority of worker threads. Optional arguments are `high`, `normal` (default), and `low`. Note that high priority threads are executed before normal/low priority tasks are taken. When a queue is empty work will be taken from high priority queues first [38].

- The stack size of the work threads. Optional arguments are `small_`, `medium`, `large`, `huge`, and `nostack`.

- Enabling or disabling work-stealing policy. Optional arguments are `static_` and `dynamic`. The work-stealing policy is enabled when `dynamic` is specified, which is advantageous for multi-task per core scenario. The worker thread that finishes its local work can steal the work from currently active worker threads, ensuring automatic load balancing of tasks.

- A customized delay after which the executor will yield to other work. Other work will not be executed while the executor is busy or waiting for work. Only after the specified time, the executor yields to other work.

Listing 3.4. Example code of `hpx::fork_join_executor` with work-stealing policy.

```cpp
1 // Rebind parallel execution policy to fork_join_executor with user-defined parameters.
2 using executor = hpx::execution::experimental::fork_join_executor;
3 executor exec(hpx::threads::thread_priority::normal,
4     hpx::threads::thread_stacksize::small_,
5     executor::loop_schedule::dynamic,
6     std::chrono::microseconds(100));
7 // 8 tasks are assigned to each core.
8 (listing cont’d.)
```
In order to improve utilization we consider to increase the overlap of communication with computation, as shown in Fig 3.6. This is possible as only the cores that work on the tasks that require inter-node communication will be busy with communication tasks (see green square), while all remaining cores are available to start their computational tasks immediately (see purple square).

![Overlapping communication and computation](image)

Figure 3.6. Overlapping communication and computation.

In addition, a dynamic work-stealing policy can be enabled to further improve the performance, as shown in Fig 3.7. The cores that finish its own work can steal the work from its neighbours. This also indicates that HPX is more suitable for irregular applications where the work cannot be evenly distributed across the cores due to the support of work-stealing and load balancing.

### 3.4 Experimental Results

All experiments were conducted on Buran nodes of the Rostam cluster. The hardware and software details are shown in Table 3.1 and Table 3.2, respectively. In Section 3.4.1 the overheads of HPX and other systems are measured when considering the scenario where
the runtime overhead is dominant, where one computational task is assigned to each core. In Section 3.4.2, overdecomposition is adopted where more than one computational tasks are assigned to each core.

### 3.4.1 Performance of a single task on each core

A Task Bench stencil pattern is utilized to compare the performance of HPX with the mainstream MPI, OpenMP, and MPI + OpenMP systems. To characterize the performance limited by runtime overhead, the total number of tasks is set to the number of total cores. Each run is 48 tasks wide since a Buran node has 48 cores, and 1000 time steps long. We
run each system for five times and a confidence interval with 99% confidence level is shown for the variance.

Figure 3.8a presents the Tera FLOP/s reached with a compute-bound kernel, varying the grain size. Note that the time for each vertex to execute such a kernel with a grain size of one is measured as 2.5 ns on the target architecture and might be different on a different machine. Almost all systems achieve peak FLOP/s, i.e. $2.44 \times 10^{12}$, when the grain size is large enough. This is also specific to the architecture used to run the benchmarks. Figure 3.8b shows the efficiency of each system regarding to the peak FLOP/s vs. task granularity. Task granularity is measured by: wall time $\times$ number of cores / number of tasks. Task Bench uses METG (Minimum Effective Task Granularity) as a metric, which essentially indicates the scaling capabilities of the system on target. METG exposes how high the computing performance (FLOP/s) can be maintained as the amount of work per task gets smaller. The reasoning behind METG is that for larger task granularity, almost all systems are expected to behave optimally while these systems likely diverge when task granularity approaches or drops below METG. In this work, we use the same choice of 50% as the Task Bench paper to compare the smallest average task granularity such that each system reaches at least 50% peak efficiency. Figure 3.8 shows METG of each system, which is the intersection of its efficiency curve and the 50% efficiency red dashed line in Figure 3.8. To calculate METG, we first measure the peak FLOP/s and get the efficiency percentage of each system responding to the peak FLOP/s. For more details about METG, we refer to [8]. We find that MPI has the smallest METG, 3.9 $\mu$s, while HPX achieves a METG of 19.3 $\mu$s, and MPI + OpenMP reaches a METG of 50.9 $\mu$s. METGs under different scenarios are listed in Table 3.3.

Figure 3.9 and Figure 3.10 show the performance of each system using 4 and 8 nodes, respectively. We only report the results using 4 and 8 nodes because the trends of each system using 2 nodes are very similar as them. We observe MPI achieves the smallest METG. This is because for MPI implementation, each process is using one core and thus
Figure 3.8. The Stencil pattern, 1 node (48 cores), 48 tasks.
Figure 3.9. The stencil pattern, 4 nodes (48 cores per node), 192 cores, 192 tasks.
Figure 3.10. The stencil pattern, 8 nodes (48 cores per node), 384 cores, 384 tasks.
Table 3.3. METG (µs) of each system for the stencil pattern with a single task per core or with different overdecomposition, using 1 node.

<table>
<thead>
<tr>
<th>System</th>
<th>single task per core</th>
<th>overdecomposition 8</th>
<th>overdecomposition 16</th>
</tr>
</thead>
<tbody>
<tr>
<td>HPX</td>
<td>19.3</td>
<td>39.2</td>
<td>54.1</td>
</tr>
<tr>
<td>MPI</td>
<td>3.9</td>
<td>6.1</td>
<td>7.6</td>
</tr>
<tr>
<td>MPI + OpenMP</td>
<td>50.9</td>
<td>152.5</td>
<td>258.6</td>
</tr>
</tbody>
</table>

there is no parallelization and neither parallelization overheads. MPI is more suitable for cases when the work can be evenly distributed where each process has similar amount of work. For cases the work is not evenly distributed, for instance, there is a huge amount of work for one process, other processes that finish their work will have to wait for this process. Because MPI does not support work-stealing and handle load imbalance. Together with Figure 3.8, we notice HPX loses advantage over MPI + OpenMP once inter-node communication is involved.

Figure 3.11 shows the smallest grain size of each system that can be weak scaled to 8 nodes with 50% efficiency. Figure 3.11a shows that the smallest grain size of HPX that can be weak scaled to 8 nodes with 50% efficiency is $2^{16}$. Figure 3.11b demonstrates that such smallest grain size of MPI for is $2^{10}$ while Figure 3.11c shows that such smallest grain size of MPI + OpenMP for is $2^{14}$.

3.4.2 Performance of overdecomposition

To quantify the performance of overdecomposition, the total size of tasks is set to $N$ times the number of total cores, such that each core processes $N$ tasks. In this subsection, $N$ is set to 8 and 16, respectively.

Table 3.3 lists METGs of each system for the stencil pattern without overdecomposition, with overdecomposition 8 (8 tasks per core), and overdecomposition 16 (16 tasks per core), using one node, respectively. For all systems, MPI achieves the smallest METG for these three scenarios, HPX performs the second, and MPI + OpenMP reaches a largest METG.

As reported by Task Bench paper [8], METG summarizes system overheads and laten-
Figure 3.11. The stencil pattern, weak scaling performance, 48 cores per node, 48 tasks per node, each colorful line ($2^N$) represents different grain sizes, where $N = 4, 6, ..., 26$. 
Figure 3.12. METG of each system with varying number of nodes for different overdecomposition, lower is better, flat is ideal.
cies into one number. Figure 3.12 presents METGs of each system with varying number of nodes. Lower is better because a lower METG indicates a smaller task granularity required to achieve at least 50% overall efficiency. Flat is ideal because a flat line implies that the communication topology does not affect METG by increasing the number of nodes. We observe that MPI has lower and flat trends, while HPX and MPI + OpenMP have higher and rising tendencies. Again, this is because for MPI implementation, each process is using one core and thus there is no parallelization and neither parallelization overheads. MPI is more suitable for this scenario where the work can be evenly distributed and each process has similar amount of work. For HPX, it binds one thread to each core, so that operating system will not move these HPX threads. Each thread has its own work queue. When a thread finishes its own work, it will check its neighbors and also the whole system to steal the work. HPX is more suitable for cases that the work is irregular, or not evenly. In addition, HPX has parallelization, this will generate parallelization overhead, for instance, there are multi-thread protection codes that are not existed in sequential code.

3.5 Summary

First, we investigated the runtime overheads of each system when a single task was assigned to each core. We observed that all systems were able to (almost) achieve the peak FLOP/s as the grain size was large enough. However, as the grain size shrunk, MPI had less overhead than HPX and MPI + OpenMP. For weak scaling, MPI had the smaller grain size than HPX and MPI + OpenMP that can be weak scaled to 8 nodes with 50% overall efficiency.

We then quantified the overheads when overdecomposition was adopted, where multi-task was assigned to each core. Specifically, overdecomposition 8 (8 tasks per core) and overdecomposition 16 (16 tasks per core) were specified, respectively. We observed that when there were more tasks per node, MPI was the best and HPX achieved better performance than MPI + OpenMP. This was because for MPI implementation, each core was assigned to each process, and thus there was no parallelization and neither parallel
overheads. MPI was advantageous for the case where the work can be evenly distributed such that each process has similar amount of work. For cases where the work cannot be evenly distributed, MPI would have degraded performance due to the lack of support for work-stealing and load balance. While for HPX and MPI + OpenMP, parallelism incurred costs. The reason HPX performed better than MPI + OpenMP for multi-task scenarios was that HPX overlapped communication with computation and enabled work-stealing. While for MPI + OpenMP, it did not support work-stealing and collected communication requests in sequential. In addition, there was a global barrier for each node which further degraded the performance.

Utilizing Task Bench enabled us to investigate the overheads introduced by HPX and compared that with the main stream MPI and MPI + OpenMP libraries. The light-weight threads as well as work-stealing scheduling in HPX incurred some costs. Such overheads were negligible when the grain size is large enough. However, for small grain sizes, the overheads limited the performance. To conclude, the overheads of fine-grain parallelism was not inherent to the programming models, and benchmark studies like this one was expected to lead to further optimizations to reduce or eliminate the gap with respect to MPI.

This study has shown that there is potential for improvement for HPX for smaller grain sizes. For future work, there are several ways which may further improve the performance. First, we could utilize and optimize HPX channels as another way for the inter-node communication. Second, we could investigate the usage of asynchronous messaging. Here, some functionality was added, wherein the MPI_request would return a hpx::future for asynchronous function calls. Third, we could try different libraries for communication, e.g. libfabric. For larger node counts, using asynchronous communication with libfabric instead of synchronous communication with MPI showed a speed-up of a factor of 3 [39].
Chapter 4
SHAD

4.1 Introduction

Recent Big Data and exascale system architectures have presented challenges to both industry and the research community, expecting software solutions to be scalable on such complex systems. SHAD is the Scalable High-performance Algorithms and Data-structures C++ library, providing general purpose building blocks and supporting high-level custom utilities. SHAD is designed with scalability, flexibility, productivity, and portability in mind, and serves as a playground for research in parallel programming models, runtime systems, and their applications [1].

SHAD consists of three layers as shown in Figure 4.1. The upper layer, called SHAD extensions, contains high-level libraries obtained by composing data structures and/or other extensions, e.g. a graph library. The middle layer provides general purpose data structures, e.g. Array, Vector, Set, and Map. The bottom layer is an abstract runtime interface, which abstracts underlying hardware systems and manages remote procedures execution and data movements. This layer enables SHAD to be supported by different platforms, via different runtimes, e.g. Intel TBB [10] and global memory and threading (GMT) [11]. However, there are limitations of scalability using aforementioned backends. For Intel TBB, it is locally only for one node. For GMT, it relies on a centralized controller which limits the scalability and creates a network hot spot due to all to one communication for synchronization, resulting in degraded performance at high process counts. In this chapter, we explore the scalability and performance of SHAD when interfacing with HPX, the C++ standard library for parallelism and concurrency, as the underlying backend.

4.2 Abstract Runtime Interface

SHAD’s portability is achieved through the abstract runtime interface, which decouples the upper layers of the library and hides the low level details of the underlying architecture [1]. The abstract runtime interface relies on three main concepts: locality, task, and
SHAD Extensions
- High-level libraries obtained by composing data structures and/or other extensions.
- Examples: graph library and linear algebra library

General Purpose Data Structures
- Array, Vector, Set, and Map

Abstract Runtime Interface
- Abstracts the underlying hardware/runtime systems
- Manages remote procedures execution and data movements.

Figure 4.1. SHAD three layers [1]

handles. A locality in SHAD is denoted as an entity wherein the memory is directly accessible. A locality can be a node in a cluster, a core, or a NUMA domain. A task is the basic unit of computation, which can be executed synchronously or asynchronously on any locality. Handles are identifiers for spawning tasks and can be used to check the completion of a task. Note that multiple tasks can be associated to the same handle, forming a task group. The abstract runtime interface offers several methods for asynchronous and synchronous task execution, including methods to check for the completion of asynchronous tasks as shown in following [9]:

- **[async]ExecuteAt** [asynchronously] execute a function on a selected locality.
- **[async]ExecuteAtWithRet** [asynchronously] execute a function on a selected locality and return a result.
- **[async]ExecuteOnAll** [asynchronously] execute a function on all localities.
- **[async]forEachAt** [asynchronously] execute a parallel loop on a specific locality.
- **[async]forEachOnAll** [asynchronously] execute a parallel loop on all localities.
- **waitForCompletion** wait for completion of a set of tasks.
template<typename FunT, typename InArgsT>
static void executeAt(const Locality &loc, FunT &&function,
const InArgsT &args) {
    using FunctionTy = void (*)(const InArgsT &);
    // check if the given locality is valid
    checkLocality(loc);
    FunctionTy fn = std::forward<decltype(function)>(function);
    using action_type = invoke_executeAt_action<decltype(fn)>;
    std::uint32_t loc_id = getLocalityId(loc);
    hpx::naming::id_type id = hpx::naming::get_id_from_locality_id(loc_id);
    hpx::sync<action_type>(id, fn, args);
}

template<typename F>
struct invoke_executeAt_action;
template<typename T>
struct invoke_executeAt_action<void (*)(T)> : ::hpx::actions::action<
    void (*)(void (*)(T), T),
    &invoke_executeAt<void (*)(T)>::call,
    invoke_executeAt_action<void (*)(T)>::call, {}
>

template<typename F>
struct invoke_executeAt;
template<typename T>
struct invoke_executeAt<void (*)(T)> {
    static void call(void (*)(f)(T), T args) {
        return f(std::move(args));
    }
};
Listing 4.1 shows an example code of `ExecuteAt` method in SHAD with HPX backend. In order to call a function remotely, i.e. on another locality, we adapt this function along with its arguments to an action. `action_type` is an user defined action. Using `hpx::sync(action_type)`, HPX synchronously executes the action, where the first argument to the action is the destination locality wherein the action should be run, the second argument to the action is the function, and the following are the arguments to the function. The arguments of the action need to get serialized at some point by putting into a binary archive that can be sent over the network, which is done by the HPX parcel port.

Listing 4.2. Example code of `asyncExecuteAtWithRet` method in SHAD with HPX backend

```cpp
template <typename FunT, typename InArgsT, typename ResT>
static void asyncExecuteAtWithRet(Handle &handle, const Locality &loc, FunT &&function, const InArgsT &args, ResT *result) {
    using FunctionTy = void (*)(Handle &, const InArgsT &, ResT *);
    FunctionTy fn = std::forward<decltype(function)>(function);
    // check if the given locality is valid
    checkLocality(loc);
    handle.id_ = handle.IsNull() ? HandleTrait<hpx_tag>::CreateNewHandle() : handle.id_;
    using action_type = invoke_asyncExecuteAtWithRet_action<decltype(fn)>;
    std::uint32_t loc_id = getLocalityId(loc);
    hpx::naming::id_type id = hpx::naming::get_id_from_locality_id(loc_id);
    handle.id_->run([=]() {
        *result = action_type()(id, fn, args, *result);
    });
}
```

```cpp
template <typename F>
struct invoke_asyncExecuteAtWithRet_action;
```

(listing cont’d.)
Listing 4.2 shows an example code of asyncExecuteAtWithRet method in SHAD with HPX backend. In SHAD, handles are utilized to spawn tasks and wait for termination of asynchronous operations. Multiple tasks can be associated to the same handle. When SHAD is supported by HPX, hpx::execution::experimental::task_group is used for spawning tasks.

We investigate the scalability and performance of SHAD when mapping its runtime interface to HPX, as shown in Figure 4.2. If an action or a remote procedure is called on a global object of SHAD, the active global address space (AGAS) would ensure that the global object could be transparently migrated from one locality to another locality. Specially, AGAS would resolve the address and deliver the active message to the responding locality and enqueue it in the locality’s thread scheduler. When interfacing with HPX, SHAD benefits from fine-grained parallelism, message driven computation, adaptive locality.
control, constraint-based synchronization, overlapping communication with computation, and minimal overheads from the lightweight threading system.

SHAD provides general data structures, e.g. `shad::array`, `shad::vector`, `shad::unordered_set`, and `shad::unordered_map`, which are designed to accommodate significant amount of data that can be accessed in massively parallel environments, and used as building blocks for SHAD extensions, i.e. higher level software libraries [9]. SHAD also offers user-friendly standard template library (STL)-inspired interfaces, SHAD algorithms, improving user productivity and guaranteeing easy porting of existing applications. These interfaces not only can be distributed on a set of localities which provides high capacity, but also can be modified and accessed in parallel, guaranteeing thread safe and offering high performance. Further, SHAD algorithms automatically manages synchronization and data movements. These compliant has analogous semantics, concepts, and syntax to STL’s APIs, in terms of iterators, ranges, and algorithms. Two execution policies are provided for performance, `distributed_sequential` and `distributed_parallel`. For `distributed_sequential`, it provides the same semantics as a shared-memory sequential execution. For instance, given a distributed container with `distributed_sequential` policy, the range of the container is first splitted into multiple portions, the STL algorithm is then called on each portion se-

---

1For a more complete description of SHAD general purpose data-structures, we refer readers to [9]
Table 4.1. SHAD algorithms, a black circle means the algorithm supports execution policies while a blank circle means not.

<table>
<thead>
<tr>
<th>Function</th>
<th>Support execution policies</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Comparison operations</strong></td>
<td></td>
</tr>
<tr>
<td>equal</td>
<td>●</td>
</tr>
<tr>
<td>lexicographical_compare</td>
<td>●</td>
</tr>
<tr>
<td><strong>Minimum/maximum operations</strong></td>
<td></td>
</tr>
<tr>
<td>max_element, min_element</td>
<td>●</td>
</tr>
<tr>
<td>minmax_element</td>
<td>●</td>
</tr>
<tr>
<td><strong>Modifying sequence operations</strong></td>
<td></td>
</tr>
<tr>
<td>copy, copy_if</td>
<td>●</td>
</tr>
<tr>
<td>fill</td>
<td>●</td>
</tr>
<tr>
<td>transform</td>
<td>●</td>
</tr>
<tr>
<td>generate</td>
<td>●</td>
</tr>
<tr>
<td>replace, replace_if</td>
<td>●</td>
</tr>
<tr>
<td><strong>Non-modifying sequence operations</strong></td>
<td></td>
</tr>
<tr>
<td>all_of, any_of, none_of</td>
<td>●</td>
</tr>
<tr>
<td>find, find_if, find_if_not</td>
<td>●</td>
</tr>
<tr>
<td>for_each</td>
<td>●</td>
</tr>
<tr>
<td>count, count_if</td>
<td>●</td>
</tr>
<tr>
<td><strong>Numeric operations</strong></td>
<td></td>
</tr>
<tr>
<td>iota</td>
<td>○</td>
</tr>
<tr>
<td>accumulate</td>
<td>○</td>
</tr>
<tr>
<td>partial_sum</td>
<td>○</td>
</tr>
<tr>
<td>reduce</td>
<td>●</td>
</tr>
<tr>
<td>inclusive_scan</td>
<td>●</td>
</tr>
<tr>
<td>transform_reduce</td>
<td>●</td>
</tr>
<tr>
<td>transform_inclusive_scan</td>
<td>●</td>
</tr>
</tbody>
</table>

Sequentially and the result of a portion is carried to the computation of next portion. For `distributed_parallel`, it provides the same semantics as a shared-memory parallel execution policy. For example, given a distributed container with `distributed_parallel` policy, the range of the container is first splitted into multiple portions, the STL algorithm is then applied independently over each portion, and after a final synchronization, the partial results are reduced into a final result, known as MapReduce execution pattern [40]. Existing SHAD algorithms are listed in Table 4.1.
Table 4.2. NERSC’s Cori Haswell Node

<table>
<thead>
<tr>
<th>CPU</th>
<th>Intel Xeon CPU E5-2698 v3</th>
</tr>
</thead>
<tbody>
<tr>
<td>Socket(s)</td>
<td>2</td>
</tr>
<tr>
<td>Core(s) per socket</td>
<td>16</td>
</tr>
<tr>
<td>RAM</td>
<td>128 GB</td>
</tr>
<tr>
<td>Interconnect</td>
<td>Cray Aries</td>
</tr>
<tr>
<td>NUMA node(s)</td>
<td>2</td>
</tr>
<tr>
<td>L1d cache</td>
<td>32K</td>
</tr>
<tr>
<td>L1i cache</td>
<td>32K</td>
</tr>
<tr>
<td>L2 cache</td>
<td>256K</td>
</tr>
<tr>
<td>L3 cache</td>
<td>40960K</td>
</tr>
</tbody>
</table>

Table 4.3. Summary of Software used.

<table>
<thead>
<tr>
<th>gcc</th>
<th>8.3.0</th>
</tr>
</thead>
<tbody>
<tr>
<td>boost</td>
<td>1.72.0</td>
</tr>
<tr>
<td>gperftools</td>
<td>2.7.0</td>
</tr>
<tr>
<td>tbb</td>
<td>2020.3</td>
</tr>
<tr>
<td>hwloc</td>
<td>2.6.0</td>
</tr>
<tr>
<td>OpenMPI</td>
<td>3.1.4</td>
</tr>
<tr>
<td>cmake</td>
<td>3.21.3</td>
</tr>
</tbody>
</table>

4.3 Performance Results

In this section, we evaluate the performance of the proposed design on Haswell nodes of the Cori supercomputer [41]. Haswell nodes information and summary of software used are listed in Table 4.2 and Table 4.3, respectively. We analyze the SHAD algorithms on SHAD core data-structure `shad::array`.

**Performance on one locality:** SHAD STL-inspired algorithms and SHAD data-structures are designed to be thread-safe, modified and accessed in parallel. We first quantify the performance of SHAD data-structures with different sizes. Further we compare the performance of SHAD algorithms with the proposed design, i.e. support SHAD via HPX, with that when SHAD are supported by TBB and GMT, respectively.

Figure 4.3 depicts the speedup obtained with the parallel execution policy compared to the sequential execution policy of the proposed design (SHAD with HPX support) for different SHAD algorithms, varying the size of `shad::array`, from 1 Million to 10 Billions elements (integers). We observe that the speedup of `shad::minmax_element` is increasing as the number of elements grows. For `shad::count_if` and `shad::reduce`, the speedups
are increasing when the size of \texttt{shad::array} increases to 1 Billion and then decreasing a bit as the size increases from 1 Billion to 10 Billions. For \texttt{shad::transform}, the speedup is increasing to 11x when the size of \texttt{shad::array} increases to 10 Millions and then decreasing when the size of \texttt{shad::array} continues to grow. This is because \texttt{shad::transform} has more memory operations than other three algorithms. A Haswell node has two NUMA node and each NUMA node has 40 MB L3 cache. The 10 Millions (integers) correspond to the L3 cache sizes. For a more detailed analysis, a finer number of element around 10 Millions is required.

Figure 4.4 shows the comparison between SHAD with HPX support, SHAD with GMT support, and SHAD with TBB support, applying \texttt{shad::transform} on \texttt{shad::array}, varying the number of cores with 100 Millions elements (integers). We observe that TBB obtains a speedup of 2.8x when using 32 cores over 1 core. HPX reaches a speedup of 6.0x when using 32 cores over 1 core. GMT achieves a speedup of 4.1 when using 32 cores over 1 core. When compared with GMT, TBB reaches a speedup of 3.4x over GMT when using 1 core, and reaches a speedup of 2.3x over GMT when using 32 cores. It can be seen that
Figure 4.4. Strong scaling, comparison between SHAD with HPX support, SHAD with GMT support, and SHAD with TBB support, using shad::transform on shad::array, varying the number of cores with 100 Millions elements (integers), 1 node (32 cores).

HPX achieves a 1.5x speedup over GMT when using 1 core, and obtains a 2.1x speedup over GMT when using 32 cores. When SHAD is supported by HPX or TBB, it is better able to exploit the intra-node parallelism than it is supported by GMT.

Figure 4.5. Performance of shad::reduce algorithm on shad::array, varying the number of elements (integers), compare the performance of the proposed design (SHAD with HPX support) with that of when SHAD are supported by TBB and GMT, respectively, 1 node (32 cores).
Figure 4.5 presents performance of shad::reduce algorithm on shad::array, varying the size of shad::array from 10 Millions to 10 Billions elements (integers). For each size we perform 6 experiments: the sequential policy and the parallel policy of TBB support, the sequential policy and the parallel policy of GMT support, the sequential policy and the parallel policy of the proposed design (HPX support). Generally, HPX and TBB present very similar performance of sequential and parallel policies, for these sizes. For GMT, when the size is 10 Millions, parallel policy performs worse than sequential policy. In addition, GMT has a lower throughput than TBB and HPX using parallel policy when the size of shad::array is small. Such throughput is rising as the size of shad::array grows and GMT can reach similar performance as TBB and HPX when using 10 Billions elements, and they achieve a speedup of 9x, obtained with the parallel execution policy compared to the sequential execution policy.

Figure 4.6 presents performance of shad::count_if algorithm on shad::array, varying the size of shad::array from 10 Millions to 10 Billions elements (integers). For each size we perform 6 experiments: the sequential policy and the parallel policy of TBB support, the sequential policy and the parallel policy of GMT support, the sequential policy and the parallel policy of the proposed design (HPX support). Generally, HPX and TBB present very similar performance of sequential and parallel policies, for these sizes. For GMT, when the size is 10 Millions, parallel policy performs worse than sequential policy. In addition, GMT has a lower throughput than TBB and HPX using parallel policy when the size of shad::array is small. Such throughput is rising as the size of shad::array grows and GMT can reach similar performance as TBB and HPX when using 10 Billions elements, and they achieve a speedup of 9x, obtained with the parallel execution policy compared to the sequential execution policy.
port, the sequential policy and the parallel policy of GMT support, the sequential policy and the parallel policy of the proposed design (HPX support). Generally, HPX and TBB have very similar performance of sequential and parallel policies, for these sizes. For GMT, when the size is 10 Millions, parallel policy performs worse than sequential policy. In addition, GMT has a lower throughput than TBB and HPX using parallel policy when the size is small. Such throughput is rising as the size of \texttt{shad::array} grows and GMT can reach similar performance as TBB and HPX when using 10 Billions elements, where the parallel policy vs. sequential policy has a speedup of 12x.

Figure 4.7 shows the speedup of SHAD algorithms of the proposed design (SHAD with HPX support) on \texttt{shad::array} using parallel policy with different number of cores over 1 core, using a \texttt{shad::array} of 20 Millions (integers). We observe that almost a linear increasing speedup for each algorithm as the number of cores increases. \texttt{shad::minmax_element} achieves a speedup of 12.1x, \texttt{shad::transform} obtain a speedup of 8.0x, \texttt{shad::reduce} reaches a speedup of 7.7x, and \texttt{shad::count_if} achieves a speedup of 5.7x.

Figure 4.8 shows the speedup of SHAD algorithms of the proposed design (SHAD with HPX support) on \texttt{shad::array} for SHAD algorithms, varying the number of cores, for 20 Millions elements (integers).
Figure 4.8. Strong scaling of the proposed design (SHAD with HPX support), using \texttt{shad::array} for SHAD algorithms, varying the number of cores, for 100 Millions elements (integers).

HPX support) on \texttt{shad::array} using parallel policy with different number of cores over 1 core, using a \texttt{shad::array} of 100 Millions (integers). We observe that almost a linear increasing speedup for each algorithm as the number of cores increases. \texttt{shad::minmax_element} achieves a speedup of 13.6x, \texttt{shad::transform} obtain a speedup of 6.0x, \texttt{shad::reduce} reaches a speedup of 7.8x, and \texttt{shad::count_if} achieves a speedup of 9.2x.

SHAD itself does not provide shared-memory parallelism because it may conflict with the underlying runtime systems. The performance of SHAD using one node with different backends diverges due to the differences of the underlying provided shared-memory parallelism.

**Performance on multiple localities:** SHAD STL-inspired algorithms can be distributed on a number of nodes of a cluster and process terabyte (TB) scale data. They are designed to be thread-safe, can be modified and accessed in parallel. We run the experiments on multiple nodes and increase the node count from 1 to 8.

Figure 4.9 illustrates the weak scalability of the proposed design (SHAD with HPX support), applying SHAD algorithms on \texttt{shad::array} with 1 Billion elements (integers)
Figure 4.9. Weak scaling of the proposed design (SHAD with HPX support), applying SHAD algorithms on `shad::array` using parallel policy when each node (32 cores) processes 1 Billion elements (integers).

The experimental results show that the SHAD algorithms have almost linearly weak scaling performance. We observe that `shad::reduce` achieves speedup of $0.71x$ when weak scaled to 8 nodes. For `shad::count_if`, the speedup drops to $0.51x$ when the node count increases from 1 to 2 due to the communication costs, and then increases to $0.58x$ when using 8 nodes. For `shad::transform`, it reaches speedup of $0.58x$ when using 8 nodes. `shad::minmax_element` achieves speedup of $0.56x$ when weak scaled to 8 nodes.

Figure 4.10 shows the comparison of weak scalability between the proposed design (SHAD with HPX support) and SHAD with GMT support, applying `shad::reduce` on `shad::array` using parallel policy, varying node count with 1 Billion elements (integers) per node. We observe that HPX obtains a speedup of $0.71x$ when weak scaled to 8 nodes. When compared with GMT, HPX achieves a speedup of $1.35x$ over GMT when using 1 node. When increasing the node count from 1 to 2, where inter-node communication occurs, HPX reaches a speedup of $1.75x$ over GMT. When further increasing the node count to 8, HPX obtains a speedup of $1.37x$ over GMT.

Figure 4.11 demonstrates the strong scaling of the proposed design (SHAD with HPX...
Figure 4.10. Weak scaling, comparison of proposed design (SHAD with HPX support) and SHAD with GMT support, applying `shad::reduce` on `shad::array` using parallel policy, varying node count with 1 Billion elements (integers) per node.

Figure 4.11. Strong scaling of the proposed design (SHAD with HPX support), applying `shad::array` for SHAD algorithms using parallel policy with 1 Billion elements (integers). It can be seen that SHAD algorithms achieve linear strong scaling efficiency. We observe that `shad::minmax_element` achieves a speedup of 4.36x when
weak scaled to 8 nodes. For \texttt{shad::transform}, it reaches a speedup of 3.15x when using 8 nodes. For \texttt{shad::count_if}, it obtains a speedup of 4.37x when the node count increases to 8 nodes. \texttt{shad::reduce} achieves a speedup of 4.56x when strong scaled to 8 nodes.

![Graph showing speedup vs number of nodes for SHAD (HPX) and SHAD (GMT)](image)

Figure 4.12. Strong scaling, comparison between SHAD with HPX support and SHAD with GMT support, applying \texttt{shad::count_if} on \texttt{shad::array} using parallel policy, varying node count with 1 Billion elements (integers).

Figure 4.12 compares the strong scalability between the proposed design (SHAD with HPX support) and SHAD with GMT support, applying \texttt{shad::count_if} on \texttt{shad::array} using parallel policy, varying node count with 1 Billion elements (integers). We observe that HPX archives a speedup of 4.4x when strong scaled to 8 nodes over 1 node. When compared with GMT, HPX obtains a speedup of 1.44x over GMT when using 1 node and the speedup increases to 4.0 when the node count increases to 8.

4.4 Summary

This chapter introduced a scalable high-performance algorithms and data-structures C++ library, SHAD. SHAD can be supported by different runtime systems via an abstract runtime interface. In this chapter, we provided HPX as an additional backend in support of the SHAD library and explore the scalability and performance of SHAD when interfacing with HPX. The methodologies that support local and remote execution in synchronously
or asynchronous manners were developed. The implementations of HPX support were provided. The performance of the proposed design was evaluated and further compared with existing backends of SHAD. SHAD with HPX support achieved good strong and weak scalability. On single node, SHAD with HPX support presented very similar performance as SHAD with TBB support, and both of them performed better than SHAD with GMT support when the number of elements was small, e.g. 10 Millions. Once the number of elements got larger, e.g. 10 Billions, all of them presented almost the same performance. For multi-node, SHAD with HPX support was better able to explore the inter-node parallelism than SHAD with GMT support. Because GMT relied on a centralized controller which may limit its scalability and created a network hot spot due to all to one communication for synchronization.
Chapter 5
Phylanx

5.1 Introduction

Phylanx, is an asynchronous array processing framework which automatically transforms user-provided Python code into an intermediate representation that is efficiently executed and distributed across all available compute resources as specified by the user [12].

Phylanx is based on HPX runtime system, implicitly and naturally benefiting from the advantages of HPX - hiding latency, fine-grained parallelism, constraint-based synchronization, message driven computation, adaptive locality control, moving work to the data, and overlapping communication with computation.

Phylanx’s software framework is as shown in Figure 5.1. The HPX runtime system is able to manage the task execution locally, i.e. on a single compute node, as well as in a distributed setting, i.e. on a set of compute nodes in a cluster. NumPy [42], is a well-optimized, high level library for multidimensional array with the support for mathematical functions. NumPy library’s API is applied in Phylanx as the interface to the user. Pybind11 [43] is utilized to avoid data copies between Python and C++ when operating on NumPy data objects. Phylanx’s data structures rely on the high-performance open-source C++ library Blaze [44], which supports HPX as a parallelization library backend and perfectly maps its data to Python data structures, e.g. a Python list is mapped to a C++ vector, 1-D and 2-D NumPy arrays are mapped to a Blaze vector and a Blaze matrix, respectively [2]. For more detailed information of Phylanx, we refer to [2] and [45].

5.2 Distributed Computing in Phylanx

The recent availability of large-scale datasets makes the machine learning and deep leaning applications hard to fit into a single node’s memory. Distributed representing the multidimensional array and then dividing the work across a set of compute nodes is one of

Sections 5.4 and 5.5 of this chapter have been previously published as Steven R. Brandt, Bita Hasheminezhad, Nanmiao Wu, et al., “Distributed Asynchronous Array Computing with the JetLag Environment”, the International Conference for High Performance Computing, Networking, Storage, and Analysis, November 2020, pp. 49-57. ©2022 IEEE. Reprinted with permission.
To distribute the multidimensional data array, Phylanx tiles the data arrays. There are a number of tiling types, e.g., row tiling, column tiling, and sym tiling, as shown in Figure 5.2. For example, each locality is responsible for one or multiple rows when row-tiling is selected. The number of rows that each locality is assigned to, is determined by the total number of localities and the total number of rows.

Each of the tiles of the data arrays is represented like a fully local data array with an attached annotation. The annotations represent the information about all partitions of the distributed array, which are metadata about the array, containing a unique name that is recognized by AGAS, a generation number, the ID of the current locality, the number of localities the array is distributed across, and the tiling method of the array [46]. Listing 5.1 shows an example of the distributed diagonal primitive with an attached annotation, wherein \texttt{arr} denotes a distributed vector, \(k\) (default is 0), denotes the diagonal above the main diagonal when \(k > 0\) and the diagonal below the main diagonal when \(k < 0\), \texttt{tiling\_type} denotes how locality is responsible for the given tile of data, \texttt{tiling\_index} (non-
(a) Original data matrix, 4 rows and 4 columns, (b) Row-tiling, 4 localities are available, each locality is responsible for one row.

(c) Column-tiling, 4 localities are available, each locality is responsible for one column. (d) Sym-tiling, 4 localities are available, each locality is responsible for one block.

Figure 5.2. Different tilings in Phylanx

negative integer, default to current locality) denotes the tile index that needs to generate the diagonal array for, and numtiles (default to the number of localities specified in the application) denotes the number of tiles of the returned array. This distributed diagonal primitive returns a 2-D array with arr on the k-th diagonal. Figure 5.3 shows an example of applying the distributed diagonal primitive on a 1-D vector with 2 localities.

Listing 5.1. Distributed diagonal primitive with arguments.

```python
1 diag_d(
2     _1_arr,
3     __arg(_2_k, 0),
4     __arg(_3_tiling_type, "sym"),
5     __arg(_4_tile_index, find_here()),
6          (listing cont'd.)
```
Using attached annotations simplifies the implementation. Phylanx execution is strictly single program multiple data (SPMD) style, i.e. each locality executes the same program on a different part of the distributed array and they communicate as necessary. To execute the program on a locality, sometimes a part of the data that is on a remote locality is required. In that case, the primitive fetches that partition using the information provided by the annotation [46]. For example, we apply the distributed all gather primitive on 2 localities, where each locality contains different part of the distributed array, and this primitive returns a future holding a 2-D array with all values sending by all participating localities. Figure 5.4 shows that each locality is able to have the whole array after applying the distributed all gather primitive, where row tiling is selected on the distributed array.

5.3 Alternating Least Square (ALS) Algorithm

Recently, an important challenge is to help users sort through a large number of products and locate the ones they most prefer to. One of the applications that address this challenge is the recommender system, which offers customers with personalized recommendations for items that suit their unique tastes, e.g. Netflix movie recommendations.

Collaborative filtering approach [47] is one of the strategies, relying only on past user behaviors. For example, previous transactions records or earlier item ratings. The basic idea behind collaborative filtering is to use the known preferences of a group of users
and make recommendations for other users [48]. Matrix factorization [49] is one of the approaches of collaborative filtering that tries to explain the ratings by characterizing both items and users. That is, given the ratings that users have for certain items and predict the possible ratings for the rest of unrated items.

ALS is a matrix factorization approach that can be utilized for recommender systems. The basic idea is to take a large matrix and factor it into some smaller representations of the original matrix, as shown in Figure 5.5, where the original $m \times n$ rating matrix, where
\(m\) is the number of total users and \(n\) is the number of total items, is factored into two much smaller matrices, a “all users by k features” matrix \((m \times k)\), called user matrix, and a “all items by k features” matrix \((k \times n)\), called item matrix. The feature dimension, \(k\), is usually set to a relatively small number, \(e.g. 10\). These features are called hidden features or hidden factors, and are learned from the data. Therefore, the goal is to find the vector for each user and the vector for each item. With the user matrix and item matrix, we can estimate the missing ratings and do recommendations. We can formulate the problem with the aim to minimize the least square error of the observed ratings by finding the optimal user matrix and item matrix (with regularization) as \([50]\)

\[
\min_{X,Y} \sum_{r_{ui}\text{observed}} (r_{ui} - x^T u y_i)^2 + \lambda \left( \sum_u \|x_u\|^2 + \sum_i \|y_i\|^2 \right), \tag{5.1}
\]

where \(r_{ui}\) is the rating for item \(i\) by user \(u\), \(x_u\) and \(y_i\) are the feature vectors for users and items, respectively, \(X\) and \(Y\) are the \(m \times k\) user matrix and \(k \times n\) item matrix, respectively, \(u \in \mathbb{M}, i \in \mathbb{N}, \mathbb{M} = \{1, \ldots, u, \ldots m\}\), and \(\mathbb{N} = \{1, \ldots, i, \ldots n\}\). ALS approach is to first fix item matrix \(Y\) and optimize user matrix \(X\), then fix user matrix \(X\) and optimize item matrix \(Y\), and repeat these two steps until \(X\) and \(Y\) converge.

### 5.4 Distributed ALS in Phylanx

To implement a distributed ALS in Phylanx, we first tile the rating matrix by user, \(i.e.\) row tiling, called \(R_1\), and each locality is responsible for one tile. Similarly, we tile the ratings matrix by item, \(i.e.\) column tiling, called \(R_2\), and each locality is responsible for one tile, as shown in Figure 5.6 where 2 localities are available. Therefore, there are two copies of rating matrix with different tilings. Then the user matrix \(X\) and the item matrix \(Y\) are randomly generated as initial. Following that, we broadcast the user matrix \(X\) and the item matrix \(Y\). Next, we use row tiling rating matrix \(R_1\) and item matrix \(Y\) to update user matrix \(X\), and then use column tiling rating matrix \(R_2\) and the updated user matrix \(X\) to further update item matrix \(Y\), for a few iterations until user matrix \(X\) and item matrix \(Y\) converge. The algorithm of distributed ALS is shown in 1. The Python
source code of distributed ALS is shown in Listing 5.2.

![Image of distributed ALS](image)

Figure 5.6. Distributed ALS in Phylanx where 2 localities are available, two copies of rating matrix with different tilings.

**Algorithm 1** Distributed ALS algorithm [50]

1: Partition rating matrix by user to create $R_1$ and partition rating matrix by item to create $R_2$.
2: Random generate user matrix $X$ and item matrix $Y$ and broadcast them.
3: while user matrix and item matrix do not converge do
4: Use $R_1$ and item matrix $Y$ to update user matrix $X$.
5: Use $R_2$ and updated user matrix $X$ to further update item matrix $Y$.
6: end while

Listing 5.2. Python source code of distributed ALS implementation [46].

```python
from phylanx import Phylanx

@Phylanx
def als_d(ratings_row, ratings_column, regularization, num_factors, iterations, alpha, enable_output):
    total_num_users = shape_d(ratings_row, 0)
    total_num_items = shape_d(ratings_row, 1)
    num_users = shape(ratings_row, 0)
    (listing cont’d.)
```

52
num_items = shape(ratings_column, 1)
conf_row = alpha * ratings_row
conf_column = alpha * ratings_column
conf_u = constant(0.0, [total_num_items])
conf_i = constant(0.0, [total_num_users])
c_u = constant(0.0, [total_num_items, total_num_items])
c_i = constant(0.0, [total_num_users, total_num_users])
p_u = constant(0.0, [total_num_items])
p_i = constant(0.0, [total_num_users])
X_local = random_d([total_num_users, num_factors], nil, nil, nil, "row")
Y_local = random_d([total_num_items, num_factors], nil, nil, nil, "row")
X = all_gather_d(X_local)
Y = all_gather_d(Y_local)
I_f = identity(num_factors)
I_i = identity(total_num_items)
I_u = identity(total_num_users)
k = 0
i = 0
u = 0
XtX = dot(transpose(X), X) + regularization * I_f
YtY = dot(transpose(Y), Y) + regularization * I_f
A = constant(0.0, [num_factors, num_factors])
b = constant(0.0, [num_factors])
while k < iterations:
    if enable_output:
        print("iteration ", k)
        print("X: ", X_local)
        print("Y: ", Y_local)
    while u < num_users:
        conf_u = slice_row(conf_row, u)
        (listing cont’d.)
c_u = diag(conf_u)
p_u = __ne(conf_u, 0.0, true)
A = dot(dot(transpose(Y), c_u), Y) + YtY
b = dot(dot(transpose(Y), (c_u + I_i)), p_u)
X_local[u] = dot(inverse(A), b)
u = u + 1
u = 0
X = all_gather_d(X_local)
XtX = dot(transpose(X), X) + regularization * I_f
while i < num_items:
    conf_i = slice_column(conf_column, i)
c_i = diag(conf_i)
p_i = __ne(conf_i, 0.0, true)
A = dot(dot(transpose(X), c_i), X) + XtX
b = dot(dot(transpose(X), (c_i + I_u)), p_i)
Y_local[i] = dot(inverse(A), b)
i = i + 1
i = 0
Y = all_gather_d(Y_local)
YtY = dot(transpose(Y), Y) + regularization * I_f
k = k + 1
return (X, Y)

5.5 Performance Results

We test the performance of the distributed ALS algorithm on Queen Bee 3 system maintained by LONI [51], where the information about the compute nodes is shown in Table 5.1. A non-distributed ALS NumPy implementation is used as a benchmark. Phylanx is built on HPX 1.5.0, master of Blaze, BlazeTensor, and Pybind11 as of Sep 2020. Both Phylanx and the NumPy implementations use openblas 0.3.10. The dataset is MovieLens [52] 20M, which contains 20 million ratings, 27,000 movies, and 138,000 users.

Figure 5.7 shows the comparison of the distributed ALS Phylanx implementation and
Table 5.1. Specification of the compute nodes of the Queen Bee supercomputer.

<table>
<thead>
<tr>
<th>Processor type</th>
<th>Intel Cascade Lake Xeon 64bit</th>
</tr>
</thead>
<tbody>
<tr>
<td>Processor</td>
<td>2 24-Core</td>
</tr>
<tr>
<td>Processor speed</td>
<td>2.4GHz</td>
</tr>
<tr>
<td>Memory</td>
<td>192 GB</td>
</tr>
</tbody>
</table>

Figure 5.7. Comparison of distributed ALS (Phylanx implementation) and non-distributed ALS NumPy implementation on part of MovieLens 20M dataset, which contains 4800 movies and 4800 users, varying the number of threads, 1 node (48 cores per node).

the non-distributed ALS NumPy implementation on part of the MovieLens 20M dataset, which contains 4800 movies and 4800 users. We observe that Phylanx implementation is faster than non-distributed NumPy implementation when using 8 or more threads. The performance of Phylanx implementation is worse than NumPy implementation when using less than 8 threads is due to the overheads introduced by Phylanx execution. When parallelization is added, the performance is better. It can be seen that Phylanx implementation is better able to exploit parallelism than the NumPy implementation.

Figure 5.8 shows the strong scaling performance varying the number of nodes. We observe that the Phylanx implementation has a good scalability and improving speedups as the number of nodes increases. Specially, Phylanx implementation can achieve 26.3x speedup on 64 nodes over 1 node.
Figure 5.8. Comparison of distributed ALS (Phylanx implementation) and non-distributed ALS NumPy implementation on part of MovieLens 20M dataset, which contains 4800 movies and 4800 users, varying the number of nodes (48 cores per node).

Figure 5.9 demonstrates the speedup of distributed ALS Phylanx implementation on four different dataset sizes: which are $4800 \times 4800$, $9600 \times 9600$, $19200 \times 9600$, and $19200 \times 19200$ datasets, respectively, where the first number denotes the number of movies and second number denotes the number of users. It can be seen that distributed ALS Phylanx implementation has improving speedups as the size of datasets increases.

5.6 Summary

This chapter introduced an asynchronous array processing framework, Phylanx. The distributed computing in Phylanx along with a number of distributed primitives were presented. An alternating least square (ALS) recommendation algorithm was introduce. The methodologies that support a distributed ALS implementation in Phylanx were developed. The performance of the distributed Phylanx implementation was evaluated and further compared with a non-distributed NumPy implementation. The performance results shown that Phylanx implementation was better able to exploit the intra-node parallelism than the NumPy implementation. Further, Phylanx implementation presented a good scalability.
Figure 5.9. Performance of distributed ALS on different sizes of dataset, where the first number denotes the number of movies and second number denotes the number of users, varying the number of nodes (48 cores per node).
Chapter 6
Second-order optimization

6.1 Introduction

For machine learning applications, the first-order methods, *e.g.* stochastic gradient descent (SGD) and its variants, are the most popular optimizers due to their simplicity and versatility [53]. However, they have a number of notable shortcomings. First, it is relatively slow to converge since they only consider first-order gradient information. Second, they are sensitive to the choice of hyper-parameters, *e.g.* learning rate. In practice, training a machine learning application with an SGD optimizer is a trial and error tuning process that intertwines learning rate and minibatch size. Third, there is a limited scalability when do distributed training with SGD due to the overhead of parallelization itself, communication and synchronization costs.

Alternatively, second-order methods [53–56] can be utilized for scalable machine learning training, enabling larger and more complex models to be trained in significantly less time. Second-order methods have following appealing features. First, certain second-order methods are scalable, offering multiple opportunities for computational efficiency and parallelization, since they can use the entire training set as a mini-batch and thus exploit maximum data parallelism [13, 57–61]. Second, they involve less parameter tuning and thus are more robust to hyper-parameters [62]. However, there is significant difficulty with the naïve second-order method, *i.e.* Newton’s method. This is because for some machine learning applications, the number of parameters can be very large and thus expensive to compute and store a Hessian matrix. Further in some cases, Hessian may not even be computable or representable. Therefore, it is not realistic to apply naïve second-order method for large-scale machine learning applications. This inspires us to utilize improved second-order methods, *e.g.* Hessian-free Newton method.
6.2 Hessian-free Newton Method

Hessian-free Newton method has been investigated in the optimization community for decades [63]. Different from the naïve Newton method, which requires to compute Hessian matrix, Hessian-free Newton method computes the Hessian-vector product by using the finite difference method without forming the Hessian directly:

\[ H d = \lim_{\varepsilon \to 0} \frac{\nabla f(\theta + \varepsilon d) - \nabla f(\theta)}{\varepsilon}, \quad (6.1) \]

where \( f \) is the objective function, \( \theta \) denotes the parameters of the objective function, \( H \) is the Hessian matrix of \( f \), and \( d \) is a descent direction [13].

On the other hand, Hessian-free Newton method is appealing because it can perform an incomplete optimization via the linear conjugate gradient or conjugate residual algorithms, called Krylov-Newton (KN), and benefit from a super linear convergence rate when guaranteeing the linear solves are reasonably precise [64]. J. Martens [13] proposes a Hessian-free optimization method with a conjugate gradient algorithm.

Further, the Krylov-Newton method does not require the Hessian matrix as accurate as the gradient for each iteration, ensuring that the Hessian estimate is more robust to noise [65]. There is a number of existing work studies the choice of a smaller sampler for estimating the Hessian matrix, and proves that such subsampled Krylov-Newton method could be robust and effective when choosing a proper subsample size [66–68].

6.3 Performance Results

We show the performance of a Krylov-Newton method, implemented as a PyTorch optimizer and compare its performance with SGD on an image classification task, using multi-layer perceptron (MLP) network with one hidden layer. We measure the performance on Intel Cascade Lake Xeon 64-bit processor, 224-core 2.4 GHz processors, on Queen Bee 3 system maintained by LONI [51]. Here we use the EMNIST dataset [69], which contains 10 class, 240,000 training data, and 40,000 test data. The hidden layer size is set to 150.
For SGD, the learning rate is set to 0.01, the momentum is set to 0.9, and the mini-batch size is set to 64. For Krylov-Newton, learning rate is set to 0.33, full batch, and we load the model which runs with SGD for 2 epochs.

Figure 6.1 demonstrates the convergence of KN and SGD optimizers. For SGD, it takes 195 seconds to achieve 98.85% accuracy. While for KN, it takes 89 seconds to achieve the similar accuracy. KN achieves 2.3x speedup than SGD.

![Image Classification: EMNIST](image)

**Figure 6.1.** Comparison of convergences of Krylov-Newton (KN) method and SGD optimizer on EMNIST dataset for an image classification task.

Figure 6.2 shows the scalability of KN and SGD via different number of threads. KN scales well as the number of threads increasing while SGD does not scale well. In addition, KN obtains the higher throughput when using 4 or more threads.

To measure the scalability of KN using multiple nodes, we utilize Pytorch’s distributed data-parallel (DDP) module for single program multiple data training, wherein the Gloo [70] backend is selected. With DDP, the model is able to be replicated on every process and each model replica is fed with a different set of input data. DDP manages the gradient communications to keep model replicas synchronized and overlaps communication with the gradient computations to speedup training [71].

Figure 6.3 illustrates the scalability of KN varying the number of processes. KN scales
Figure 6.2. Comparison of throughput of Krylov-Newton (KN) method and SGD optimizer on EMNIST dataset for an image classification task, varying the number of threads. 

well and obtains an improved throughput as the number of processes increasing. Note that the performance of SGD varying the number of processes is not included in this experiment because SGD needs long time to tune the hyper-parameters for different number of processes, e.g. the train batch size and the number of epochs.

Figure 6.3. Performance of Krylov-Newton (KN) method and SGD optimizer on EMNIST dataset for an image classification task, varying the number of process.
6.4 Summary

This chapter introduced the Hessian-free Krylov-Newton second-order method as a means to accelerate the training process for machine learning. It was seen that using Hessian-free Krylov-Newton method can achieve the similar accuracy as the first-order optimizer SGD but with less time, i.e. a factor of 2.2x speedup. Further, Hessian-free Krylov-Newton optimizer was able to take advantage of parallel and distributed compute resources and enable scalable speedup, since second-order methods can exploit the maximum data parallelism by using a “maxibatch”—a mini-batch that consists of the entire training set. Utilizing improved second-order methods was promising for scalable machine learning training, enabling larger and more complex models to be trained in significantly less time.
Chapter 7
Conclusion

This work focused on the performance analysis and improvement for scalable and distributed applications based on an Asynchronous Many-task runtime system HPX. Four applications were studied, i.e. Task Bench, SHAD, Phylanx, and a Hessian-free second-order method.

With regards to Task Bench, a generic distributed HPX implementation was first implemented. The performance bottleneck and system overheads are then measured and analyzed. The methodology to further exploit the parallelism with reduced overheads was developed. Finally an improved implementation was presented and the performance was compared with other main stream programming systems. First, we investigated the runtime overheads of each system when a single task was assigned to each core. We observed that all systems were able to (almost) achieve the peak FLOP/s as the grain size was large enough. However, as the grain size shrunk, MPI had less overhead than HPX and MPI + OpenMP. For weak scaling, MPI had the smaller grain size than HPX and MPI + OpenMP that can be weak scaled to 8 nodes with 50% overall efficiency. We then quantified the overheads when overdecomposition was adopted, where multi-task was assigned to each core. We observed that when there were more tasks per node, HPX achieved better performance than MPI + OpenMP, while MPI was the best. This was because for MPI implementation, each core was assigned to each rank, and thus there was no parallel overheads. While for HPX and MPI + OpenMP, parallelism incurred costs. The reason HPX performed better than MPI + OpenMP for multi-task scenarios was that HPX overlapped communication with computation while MPI + OpenMP collected communication requests in sequential and there was a global barrier for each node which further degraded the performance.

With regards to SHAD, we explored the scalability and performance of SHAD when interfacing with HPX, the C++ standard library for parallelism and concurrency, as the underlying backend. The methodologies that support local and remote execution in syn-
chronously or asynchronous manners were developed. The performance of the proposed design was evaluated and further compared with existing backends of SHAD. SHAD with HPX support achieved good strong and weak scalability. On single node, SHAD with HPX support presented very similar performance as SHAD with TBB support, and both of them performed better than SHAD with GMT support when the number of elements was small, e.g. 10 Millions. Once the number of elements became larger, e.g. 10 Billions, all of them presented almost the same performance. For multi-node, SHAD with HPX support was better able to explore the intra-node and inter-node parallelism than SHAD with GMT support.

With regards to Phylanx, the distributed computing along with a number of distributed primitives in Phylanx were first introduced. An alternating least square (ALS) algorithm for recommendation system was presented. The methodologies that support a distributed ALS implementation in Phylanx were then developed. The performance of the distributed Phylanx implementation was evaluated and further compared with a non-distributed NumPy implementation. Phylanx implementation was better able to exploit the intra-node parallelism than the NumPy implementation. Further, Phylanx implementation presented a good scalability.

With regards to the Hessian-free second-order method, the advantages and disadvantages of first-order and second-order methods were analyzed. An Hessian-free Krylov-Newton optimizer was implemented on both PyTorch and NumPy. The convergence behavior of the Hessian-free Krylov-Newton optimizer was evaluated and compared with a popular first-order optimizer. In addition, the scalable performance of the Hessian-free Krylov-Newton optimizer with different number of threads and processes was further characterized. It was seen that using Hessian-free Krylov-Newton method can achieve the similar accuracy as the first-order optimizer SGD but with less time, i.e. a factor of 2.2x speedup. Further, Hessian-free Krylov-Newton optimizer was able to take advantage of parallel and distributed compute resources and enable scalable speedup, since second-
order methods can exploit the maximum data parallelism by using a “maxibatch”—a mini-
batch that consists of the entire training set. Utilizing improved second-order methods was
promising for scalable machine learning training, enabling larger and more complex models
to be trained in significantly less time.
Appendix. Copyright Information

Distributed Asynchronous Array Computing with the JetLag Environment


Author: Steven R. Brandt
Publisher: IEEE
Date: Nov. 2020

Copyright © 2020, IEEE

Thesis / Dissertation Reuse

The IEEE does not require individuals working on a thesis to obtain a formal reuse license, however, you may print out this statement to be used as a permission grant:

Requirements to be followed when using any portion (e.g., figure, graph, table, or textual material) of an IEEE copyrighted paper in a thesis:

1) In the case of textual material (e.g., using short quotes or referring to the work within these papers) users must give full credit to the original source (author, paper, publication) followed by the IEEE copyright line © [Year of original publication] IEEE.
2) In the case of illustrations or tabular material, we require that the copyright line © [Year of original publication] IEEE appear prominently with each reprinted figure and/or table.
3) If a substantial portion of the original paper is to be used, and if you are not the senior author, also obtain the senior author's approval.

Requirements to be followed when using an entire IEEE copyrighted paper in a thesis:

1) The following IEEE copyright/credit notice should be placed prominently in the references: © [Year of original publication] IEEE. Reprinted, with permission, from [Author names, paper title, IEEE publication title, and month/year of publication].
2) Only the accepted version of an IEEE copyrighted paper can be used when posting the paper or your thesis online.
3) In placing the thesis on the author's university website, please display the following message in a prominent place on the website: In reference to IEEE copyrighted material which is used with permission in this thesis, the IEEE does not endorse any of [university/educational entity's name goes here]'s products or services. Internal or personal use of this material is permitted. If interested in reprinting/republishing IEEE copyrighted material for advertising or promotional purposes or for creating new collective works for resale or redistribution, please go to http://www.ieee.org/publications_standards/publications/rights/rights_link.html to learn how to obtain a License from RightsLink.

If applicable, University Microfilms and/or ProQuest Library, or the Archives of Canada may supply single copies of the dissertation.
References


Vita

Nanmiao Wu received the B.S. degree from Nankai University, Tianjin, China, and the M.S. degree from the University of Macau, Macau, China, in 2013 and 2016, respectively. In 2016, she began her work toward the Ph.D. degree in the Department of Electrical and Computer Engineering at Louisiana State University. During her time at Louisiana State University, she worked with the STE||AR group and contributed to open source projects such as HPX, Phylanx, SHAD, and Task Bench.

Her research interests cover high-performance computing, parallel and distributed computing. Her Ph.D. thesis focuses on performance analysis and improvement for scalable and distributed applications. She previously interned at Pacific Northwest National Laboratory (2021).