8-3-2017

Compiler and Runtime Optimization Techniques for Implementation Scalable Parallel Applications

Zahra Khatami
Louisiana State University and Agricultural and Mechanical College

COMPILER AND RUNTIME OPTIMIZATION TECHNIQUES FOR IMPLEMENTING SCALABLE PARALLEL APPLICATIONS

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

The Department of Electrical and Computer Engineering

by
Zahra Khatami
B.S., University of Zanjan, 2010
M.S., Amirkabir University of Technology, 2013
M.S., Louisiana State University, 2017
December 2017

To my dear and lovely husband

Ali Moharrer.

Acknowledgements

Firstly, I would like to express my sincerest gratitude to my advisors, Prof. J. Ramanujam and Dr. Hartmut Kaiser. I am grateful to them for their consistent and invaluable support and guidance, which they provided me throughout my tenure as a PhD student at LSU. The beginning and completion of this long journey, full of challenges, would not have been possible without their generous support and efforts.

I want to express my special gratitude to all respected members of my advisory committee, Prof. Xin Li, Prof. Jianhua Chen, and Prof. Haitao Mo for their generous support and cooperation. Also, I would like to thank all my teachers of EECS for a great learning experience that I had during my graduate coursework at LSU.

I also wish to give my heartfelt thanks to my husband, Ali, who has been a constant source of support and encouragement during the challenges of graduate school and life. I am truly thankful for having you in my life.

Also, my deep and sincere gratitude to my family for their continuous and unparalleled love, help and support. Indebted to my mother and my father for giving me the opportunities and experiences that have made me who I am. I always knew that you believed in me and wanted the best for me. You selflessly encouraged me to explore new directions in life and seek my own destiny. Thank you for teaching me that my job in life was to learn, to be happy, and to know and understand myself; only then could I know and understand others.This journey would not have been possible if not for you.

Finally, I am obliged to my colleagues and friends who supported me during my PhD years. I thank Patricia Grubel, Adrian Serio, Steven Brandt, Sameer Abu-Asal, Lukas Troska, Bibek Wagle, Parsa Amini, Alireza Kheirkhah, Bryce Adelstein-Lelbach, Zach Byerly, Anton Bikineev, Sahar Navaz and Mohammad Rastegar Tohid for their warm friendship and support, who gave me the necessary distractions from my research and made my stay in Baton Rouge memorable.

Table of Contents

Abstract

The compiler is able to detect the data dependencies in an application and is able to analyze the specific sections of code for parallelization potential. However, all of these techniques provided by a compiler are usually applied at compile time, so they rely on static analysis, which is insufficient for achieving maximum parallelism and desired application scalability. These compiler techniques should consider both the static information gathered at compile time and dynamic analysis captured at runtime about the system to generate a safe parallel application. On the other hand, runtime information is often speculative. Solely relying on it doesn't guarantee maximal parallel performance. So collecting information at compile time could significantly improve the runtime techniques performance.

The goal is achieved in this research by introducing new techniques proposed for both compiler and runtime system that enable them to contribute with each other and utilize both static and dynamic analysis information to maximize application parallel performance. In the proposed framework, a compiler can implement dynamic runtime methods in its parallelization optimizations and a runtime system can apply static information in its parallelization methods implementation. The proposed techniques are able to use high-level programming abstractions and machine learning to relieve the programmer of difficult and tedious decisions that can significantly affect program behavior and performance.

Chapter 1
Introduction

1.1 Motivation

Using existing parallelization techniques, application developers are usually not able to utilize all available resources when parallelizing their applications; due to the hardware differences and applications properties. Since achieving expected scalability on each hardware requires efficient utilization of the available hardware resources, which needs a user to be aware of the details of the applied architectures. So user needs to spend a large portion of his time to study about each hardware and preparing his application to be compatible for that hardware; which is usually not feasible. The main motivation of this research is to provide a scalable framework for parallelizing an application productively for a targeted underlying hardware. One of the solutions for this challenge is improving information provided with the compiler. Since most of the compiler optimizations are based on the static information, so including dynamic analysis in those methods can improve those compiler techniques. Our first investigation in this research is improving OP2 compiler:

- Improving OP2 Compiler with HPX Runtime System. Unstructured grids are well studied and utilized in various application domains. OP2 provides a framework for the parallel execution of these unstructured grid applications on different multi-core/many-core hardware architectures [1, 2]. It uses source-source translation to generate an application's code for the targeted platform. The main goal of developing OP2 is to provide an abstraction level which allows users to parallelize their applications without having to worrying about architecture specific optimizations. This allows scientists to invest most of their time in understanding their domain problems, without learning details of new architectures, and still achieve efficient utilization of the available hardware. Its framework is designed to achieve

the near-optimal scaling on multi-core processors [3, 4]. However, as the compiler only has a static and defined access pattern [5–7], its analysis is not enough to obtain desired parallel scalability. In order to reach to this goal, OP2 needs to be able to extract parallelism automatically at runtime. By extending the compiler to take advantage of runtime information, OP2 will be able to consider dynamic analysis as well as static information while generating codes for different platforms. In Chapter 2, we improve the parallel application performance generated by the OP2 compiler by leveraging HPX, a C++ runtime system, to provide runtime optimizations. These optimizations include providing asynchronous tasking, loop interleaving, dynamic chunk sizing, and data prefetching.

In Chapter 2, we illustrate how including dynamic analysis and asynchronous techniques can improve the OP2 compiler performance. However, the static information are missed in those proposed dynamic techniques. Since those methods perform only based on dynamic analysis about the system, which are insufficient. There are several existing publications on automatically tuning optimizations techniques based on static information extracted at compile time. Since manually tuning parameters becomes ineffective and almost impossible when too many analysis are given to the program. Hence, many researches have extensively studied machine learning algorithms which optimize such parameters automatically; such as studied done in [8–12]. However, most of these optimization techniques require users to compile their application twice, first compilation for extracting static information and the second one for recompiling application based on those extracted data. Also, none of these considers both static and dynamic information. So our goal is to optimize an application's performance by predicting optimum parameters for its parallel algorithms by considering both static and dynamic information and to avoid unnecessary compilation. This leads us to our second investigation of this research, that is improving the HPX runtime techniques using static information provided with Clang compiler:

- Improving HPX Runtime System with Clang Compiler. The HPX runtime methods have been shown to be very effective in achieving better parallel scalability compared to the existing parallel techniques as it is illustrated in Chapter 2, especially for highly dynamic applications. However, as all of these HPX parallel algorithms perform based on the dynamic analysis provided at runtime, so they are unable to reach maximal parallel performance. The performance of an application depends on both the values measured at runtime and the related transformations performed at compile time. Collecting the outcome of static analysis performed by the compiler could significantly improve the HPX algorithms runtime decisions and therefore the application's performance. In order to reach this goal, HPX algorithms need to predict the optimum parameters by considering both static and dynamic information. In the current HPX algorithms, execution parameters should be selected by a user. Manually parallelizing all loops, however, may negatively effect an application's parallel performance, as some of those algorithms cannot scale desirably to a large number of threads. This leaves a user to execute each parallel algorithm of his/her application with different execution parameters to find the efficient parameter for that algorithm. This clearly is not an optimum approach. In Chapter 3, we illustrate how machine learning techniques can be applied to address these challenges. We improve the parallel performance of the applications written with HPX by automating such parameter selection which considers loops characteristics implemented with a learning model. Our results show that combining machine learning, compiler optimizations and runtime adaptation helps us to maximally utilize available resources and minimize execution time.

As it can be seen, the main motivation of this work is to develop a framework that utilizes both static and dynamic information about the system to make proper optimizations. As it is shown in Fig. 1.1, in this framework, both the compiler and the runtime exchange parameters that can be used to optimize their performance and achieve a safe parallelization. These optimizations are discussed briefly in the following subsections.

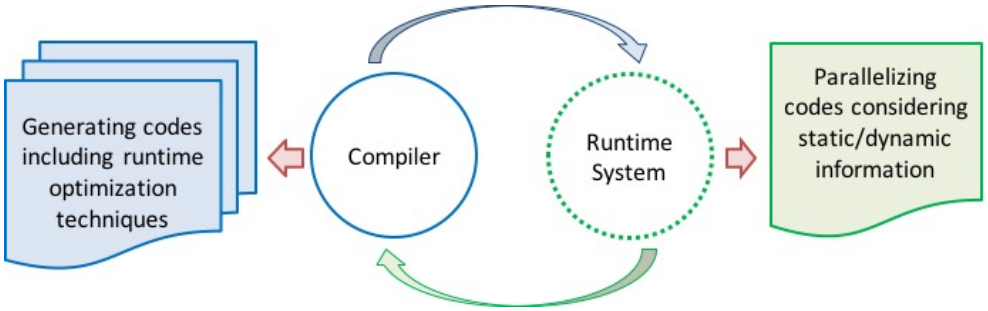


FIGURE 1.1. The main motivation of this work is to develop a framework that utilizes both static and dynamic information about the system to make proper optimizations. In this framework, both the compiler and the runtime exchange parameters that can be used to optimize their performance and achieve a safe parallelization.

1.2 Improving OP2 Compiler with HPX Runtime System

In Chapter 2, we propose different optimization methods that provide dynamic information for the code generated by the OP2 compiler. In that Chapter, we present the implementation of OP2 compiler that employs the proposed runtime techniques implemented using HPX to efficiently and automatically parallelize the dynamic applications. We show the algorithmic and implementation improvements using HPX in OP2 to hide the communication latencies and to achieve a desired scalability.

To our knowledge, we present a first attempt of redesigning OP2 to utilize the runtime techniques for improving performance of the parallel unstructured grid applications. The combination of these proposed techniques should yield a more portable and performant software stack for unstructured grid applications and enable the applications to properly scale to a higher level of parallelism compared to the existing OP2 implementation. The proposed optimization methods implemented in the developed framework can be summarized as follow:

1. Asynchronous task execution.
2. Dynamically setting chunk sizes.
3. Interleaving different loops together.
4. Prefetching data before its actual access

This hybrid framework provides the sufficient information for parallelizing an application to properly scale to a higher level of the parallelism compared to the existing OP2 implementation by combining task-based parallelism, grain size control, and prevalently asynchronous execution. The results in section 2.5 show that the parallelization performances are improved by around $40 - 50\%$ for an Airfoil application compared to using current version of an OP2. We briefly discuss these optimizations in the subsequent sub-sections.

1.2.1 Asynchronous Task Execution

One of the performance optimizations achieved in this research is enabling OP2 to provide the asynchronous task execution by using HPX. In the current OP2 design, OpenMP is used for parallel processing within a node and MPI is used for communication between different nodes. In these models, an implicit global barrier is present after each loop parallelization with OpenMP during thread synchronization [13]. Additionally, in MPI, there is an explicit barrier after nearly every step of the communication between different nodes. These two local and global synchronization barriers imposed by the programming models of OpenMP and MPI causes the parallel threads and processes to wait and results in impeding optimal parallelization. A closer analysis on the couple of unstructured applications reveals that only synchronization between small tasks are needed. So there is no need to wait for all computations and processes to be executed completely before going to the next steps.

In *blocking communication* applied using *"MPI_ Send()"* and *"MPI_ Recv()"*, each node must complete its current process before it can continue to the next step. On the other hand, in *non-blocking communication* applied using *"MPI_ Isend()"* and *"MPI_ Irecv()"*, the communications are not blocked even if the current process is not yet completed, which allows the other works to proceed. However, *"MPI_ Wait()"* or *"MPI_ Probe()"* should be used within this method to confirm whether the communications are completed, which avoids achieving optimum scaling. Another problem in using MPI is related to *granularity*, which is defined as the amount of the computations per each execution block. Fine-grained tasks have a small quantum of work and

help in achieving better load balancing [14, 15], while on the other hand, coarse-grained tasks have large amount of computations. In MPI, the coarse-grained tasks are often chosen to obtain an optimal efficiency [16], but leads to load imbalances. Integrating OpenMP into MPI is expected to increase the parallel performance, however the overheads added by loop scheduling and synchronization in OpenMP [17] in addition to creating unnecessary synchronization barriers and load imbalance issues mentioned above inhibit the promised speedup [16, 18].

In this research, HPX is used for both communication between different nodes and parallel processing within each node. It provides the dynamic parallel framework, which can easily optimize the parallelization of an unstructured grid applications. The $future$ construct used in HPX allows the system to start working on the next time step instead of waiting for all computations to be completed [19], which relaxes the global barriers, improves the parallel performance and enables an application to be executed asynchronously.

1.2.2 Dynamically Setting Chunk Sizes

HPX is able to dynamically control the chunk size during the execution by automatically setting amount of work spawned by each chunk. This optimization can be used for dynamically determining efficient chunk sizes of the loops generated by the OP2 compiler. Moreover, the new execution policy is proposed is this research that helps OP2 to make all chunk sizes of different dependent loops having same execution time, which decreases the waiting time between the chunks in those loops for obtaining better execution time. This method is discussed in more details in section 2.4.3.2.

1.2.3 Interleaving Loops Together

Using $future$-based techniques in HPX enables threads to continue their executions without waiting for the results of the previous steps to be completed, which eliminates unnecessary implicit global barrier. The new *future* based task execution technique is proposed in this research that enables OP2 to interleave different loops together at runtime by assigning output of each loop to a $future$ and passing loop inputs as $futures$ as well. As a result, if the loops are not de-

pendent on each other, they can be executed without waiting for the previous loops to complete their tasks, however, if they depend on the results of the previous loops, they will wait until the previous loops complete their processes, which removes the unnecessary barrier synchronizations between different loops. So HPX enables highly efficient parallelization compared to the existing equivalent applications. The main advantage of this method is minimizing total synchronization by rescheduling overheads and executing function asynchronously. This method is discussed in more details in section 2.4.3.

1.2.4 Prefetching Data Before Its Actual Access

Although HPX increases the on-node parallelism available to the user, however it results in increasing complexity of the memory hierarchies. One of the solutions to this problem is data prefetching, which effectively reduces the memory accesses latency [20, 21]. Thereby, the new cache prefetcher used in HPX is introduced to aid prefetching data in each time step before its actual access is executed. This method is implemented in such a way that data of the next iteration step is prefetched into the cache memory using prefetching iterator called in each iteration within a loop. The main difference between this method and the other existing methods is that HPX implementation combines a thread based prefetching method with the asynchronous task execution, which results in having asynchronous execution while prefetching data of all the containers within a loop. This method is discussed in more details in section 3.4.4.3.

1.3 Improving HPX Runtime System with Clang Compiler

- In [22, 23] different policies for executing HPX parallel algorithms are studied. These policies should be manually selected for each algorithm within an application, which may not be an optimum approach.
- Determining chunk size is another challenge in the current HPX algorithms. Chunk size is the amount of work performed by each task [19, 24] that is determined by either an *auto-partitioner* exposed by the HPX algorithms or is passed as an execution policy's parameter by using static/dynamic chunk size. However,

1. the experimental results in [25] and showed that the overheads of determining chunk size using *auto_partitioner* for some of the loops negatively effects the application's scalability;
2. on the other hand, the *static/dynamic_chunk_size* should be chosen by a user, which he/she may not know the optimum one and it can be achieved only by running application with different values of the chunk sizes and selecting the efficient one [22]..

- In addition to these challenges, in the prefetching method proposed in [26], a distance between each two prefetching operations should also be manually chosen by a user for each new program.

Automating these mentioned parameters selections by considering loops characteristics implemented in a learning model can optimize the HPX parallel applications performances. In Chapter 3 a new technique is proposed that enables HPX runtime system to select its parallel algorithms parameters at runtime by implementing machine learning algorithm on the extracted static and dynamic information. In that proposed technique, the static information about the loop body collected by the compiler and the dynamic information as provided by the runtime system is used to a learning model enabling a runtime decision to obtain highest possible performance of the loop under consideration. The presented method relies on a compiler-based source-to-source transformation, that transform a code and instructs the runtime system to apply a learning model to select either an appropriate code path (e.g. parallel or sequential loop execution) or certain parameters for the loop execution itself (e.g. chunk size or prefetching distance).

To the best of our knowledge, we present a first attempt in implementing learning models for predicting loop's parameters at runtime, in which designing these runtime techniques and capturing learning model's features are automatically performed at compile time. These proposed optimization methods discussed in Chapter 3 can be summarized as follow:

1. Determining efficient execution policy.

2. Determining efficient chunk size.

3. Determining efficient prefetching distance.

Our results show that combining machine learning, compiler optimizations and runtime adaptation helps us to maximally utilize available resources. The evaluated results discussed in Section 3.5 shows that these proposed techniques improve the parallel performance performance by around $12\% - 35\%$ for the Stream, Stencil, Matrix Multiplication and NBody application compared to setting their HPX loop's execution policy/parameters manually or using HPX auto-parallelization techniques. We briefly discuss these optimizations in the subsequent sub-sections.

1.3.1 Determining Efficient Execution Policy

In order to predict the optimum execution policy (sequential or parallel), we implement a binary logistic regression model [27] with HPX which analyzes extracted information of a loop. This model deals with the problems in which their outputs can only be two different types, $False$ or $True$. This method estimates the probability of the occurrence of each of these outputs by applying learned weights on a given characteristics set of a new loop. For implementing this learning model on a loop at runtime, we introduce a new HPX execution policy *par_if* that makes runtime system to decide whether executes loop in sequential or in parallel. This technique is implemented by considering loop's characteristics extracted with a Clang compiler and weights values learned with a binary logistic regression model. This method is discussed in more details in section 3.4.4.1.

1.3.2 Determining Efficient Chunk Size

In order to predict the optimum chunk size, we implemented a multinomial logistic regression model [27] with HPX which analyzes extracted information of a loop. This model can be considered as an extension of the binary logistic regression model that allows considering problems with more than two categories. It deals with multiclass problems to predict the output of a new application by computing probabilities of the occurrence of each different classes. For imple-

menting this learning model on a loop at runtime, we introduce a new HPX execution policy parameter *adaptive_chunk_size* that makes runtime system to choose an efficient chunk size for a loop. This technique is implemented by considering loop's characteristics extracted with a Clang compiler and weights values learned with a multinomial logistic regression model. This method is discussed in more details in section 3.4.4.2.

1.3.3 Determining Efficient Prefetching Distance

Same as determining efficient chunk size, a multinomial logistic regression model is used here for determining efficient prefetching distance. For implementing this learning model on a loop at runtime, we introduce a new HPX execution policy *make_prefetcher_policy* that makes runtime system to choose an efficient prefetching distance for a loop. This technique is implemented by considering loop's characteristics extracted with a Clang compiler and weights values learned with a multinomial logistic regression model. This method is discussed in more details in section 3.4.4.3.

1.4 Future Plan of Study

- In this research we intend to improve the performance of the code generated with OP2 compiler by delaying some optimizations into the runtime and controlling some parameters of the tasks dynamically. Although the experimental results of theses OP2 optimizations show an improvement in an application scalability, however all of these techniques are proposed for implementing parallelization in one node, not on the distributed one. Including dynamic information and runtime techniques in the distributed OP2 optimizations can result in achieving higher performance from distributed unstructured grids applications.
- Moreover, we developed new techniques that are able to implement a logistic regression on a loop to determine an optimum execution policy, chunk size and prefetching distance for an HPX loop. These techniques are able to consider both static and dynamic features of a loop and to implement a learning technique at runtime to make an optimum decision for its execution without requiring extra compilation. As powerful as these techniques may be,

more work is needed to be done in order to fully realize the potential of this work. Notably, the breadth of performance characteristics needs to be more carefully studied to understand the core features that relate to performance. Additionally more research is needed to ensure that the characteristics measured here also are relevant for other architectures such as the new Knights Landing chipset.

Chapter 6 outlines the tentative plans for our next research phase.

Chapter 2

OP2 Optimization with Runtime Parallelization Techniques

Increasing number of processors results in increasing amount of data exchanged between main memory and processors, which produces the overheads and avoids achieving desired parallelism level in an application. Optimizations techniques provided with OP2 compiler address this challenge and are able to improve parallelization performance. OP2 is able to examine the specific sections of an application's code for studying its parallelization potentials by considering its data dependencies. However, all of the information captured from these analysis are static and no dynamic information can be retrieved from them. So they are insufficient for obtaining desired scalability. To achieve better parallel performance with OP2 on modern multi-core/many-core hardware, some optimizations should be applied to intensify OP2's parallelization techniques, which are obtained in this research by delaying some compiler optimizations to be managed at runtime. The main goal in this chapter[1] is to show the OP2 compiler scalability improvement by using the proposed optimization methods implemented with the HPX, a C++ runtime system, to obtain a better parallelization performance. It is illustrated that the use of HPX's parallelization methods instead of OpenMP helps avoiding unnecessary synchronization barriers that results in extracting more parallelism for the parallel applications.

In this chapter, we propose a new technique that implementing it on an OP2 compiler makes it to provide asynchronous tasking, loop interleaving, dynamic chunk sizing, and data prefetching

[1]Parts of this chapter have been previously published as: Z. Khatami, H. Kaiser, and J. Ramanujam, "Using hpx and op2 for improving parallel scaling performance of unstructured grid applications, " in Parallel Processing Workshops (ICPPW), 2016 45th International Conference on. IEEE, 2016, pp. 190-199, Reprinted by permission. Z. Khatami, H. Kaiser, and J. Ramanujam, "Redesigning op2 compiler to use hpx runtime asynchronous techniques," in Parallel and Distributed Scientific and Engineering Computing (IPDPSW), 2017 18th IEEE International Workshop on. IEEE, 2017, Reprinted by permission. Both reprinted by permission IEEE.

to improve a parallel performance of the unstructured grids applications. It is illustrated that dynamic information obtained at runtime and static information obtained at compile time are both necessary to provide sufficient optimizations for optimal performance.

2.1 Introduction

Unstructured grids are well studied and utilized in various applications such as dense/sparse linear algebra, spectral methods, N-body methods [26], and Monte Carlo. The main goal of OP2 in to provide a scalable framework for these applications parallel execution targeted on different multi-core/many-core hardware architectures [1, 2]. OP2 is able to provide an abstraction level for a user to parallelize his applications without having to worrying about architecture specific optimizations. So he can invest most of his time in solving his scientific problems, without knowing details of the applied architectures and achieving efficient utilization of the available hardware resources. This framework is designed to achieve the near-optimal scaling on multi-core processors [3, 4]. However, as the compiler only has a static and defined access pattern [5–7], its analysis is not enough to obtain desired parallel scalability. In order to reach this goal, OP2 needs to be able to involve dynamic techniques in its parallelization techniques and to extract parallelism automatically at runtime.

In this research, we propose different optimization methods that provide dynamic information for code generated by the OP2 compiler, including providing asynchronous task execution, interleaving different loops together, dynamically setting chunk sizes of different dependent loops based on each other, and prefetching data. These proposed techniques are implemented using HPX runtime system via redesigning the OP2 framework in a way that employs both compiler's static analysis and dynamic runtime information. HPX is a parallel C++ runtime system that facilitates distributed operations and enables fine-grained task parallelism resulting in a better load balancing [28, 29]. It provides an efficient scalable parallelism by significantly reducing processor starvation and effective latencies while controlling overheads [30].

A closer analysis of unstructured applications reveals that synchronization is only required between small tasks. Prevalent parallelization paradigms, however, coerce users to join all tasks together before proceeding to the next step in the application. In HPX, we can utilize the $future$ construct to allow every task to proceed as long as the values it depends on are ready [31].This feature allows the HPX to relax the global barriers, enable flexibility, and improve the parallel performance of applications. In this research, HPX uses $futures$ based techniques to develop a new task execution strategy for codes generated by the OP2 compiler which is the basis for asynchronous tasking and interleaving loops.

In order to control the overheads introduced by the creation of each task, it is important to control the amount of work performed by each task. This amount of work is known as the *chunk size* [19, 31]. In addition, to properly interleave loops it is important for each loop to have very similar execution times which allows the waiting time between the execution of each loop to be minimal. We propose to address these two obstacles by creating a new execution policy which will dynamically control the chunk sizes during the application's execution. In addition, we also propose to create a new cache prefetcher that aids in prefetching data for each time step to reduce memory accesses latencies. This method is implemented in such a way that data of the next iteration step is prefetched into the cache memory using a prefetching iterator called in each iteration within a loop. The main difference between this method and the other existing methods is that HPX implementation combines a thread based prefetching method with the asynchronous task execution, which results in having asynchronous execution while prefetching data of all the containers within a loop.

To our knowledge, we present a first attempt of redesigning OP2 to utilize the runtime techniques for improving performance of the parallel unstructured grid applications. The combination of these proposed techniques should yield a more portable and performant software stack for unstructured grid applications and enable the applications to properly scale to a higher level of parallelism compared to the existing OP2 implementation. The results evaluated in Section 2.5

show that the parallelization performances are improved by around $40 - 50\%$ for an Airfoil application.

The remainder of this chapter is structured as follows: Section 2.2 briefly introduces OP2; Section 2.3 introduces the *future* and *dataflow* object in HPX; Section 2.4 the results from porting parallel simulation backend of OP2 to utilize HPX are studied as well as the details of implementing *hpx::for_each*, HPX asynchronous task execution methods, *dataflow* with a new execution policy, and the proposed prefetching method on an Airfoil application, and Section 2.5 evaluates the scaling speedup of the experimental tests. The conclusions and the future works can be found in Section 2.6.

2.2 Op2

OP2 is an active library that provides a parallel execution framework for unstructured grid applications on different multi-core/many-core hardware architectures [1]. It utilizes a source-to-source translator for generating code which targets different hardware configurations [2, 3, 32]. OP2 provides C/C++ and a Fortran API that translates an application into targeted back-end hardware implementation. The code can be transformed easily into different configurations such as serial, multi-threaded using OpenMP and CUDA, or heterogeneous which utilizes MPI, OpenMP, and CUDA [3]. Its goal is to decouple a scientific aspects and specifications of an application from its parallelization to help achieving near-optimal performance.

In general, OP2 splits an application into two parts:

1. Higher application level: Users can concentrate more on writing their code and solving their problems that will be the same and unchanged for different underlying hardwares.
2. Lower implementation level: This level optimizes an application execution process targeted different platform by analyzing data access patterns.

As a result, a user applies the same API statements for the functions calls while the generated code with OP2 is able to utilize the available resources of a target architecture efficiently. This methodology makes OP2 to be able to support for any future novel hardware architectures.

In this section, we first walk through a simple OP2 code to show its implementation details and then we introduce the Airfoil application which is used as a case study for this research.

2.2.1 Simple Code Implementation with OP2

This section generally shows how unstructured grids are defined with OP2. The OP2 API handles the data dependencies by providing mesh represented data layouts. The provided framework is defined based on sets, data on sets, mapping connectivity between the sets, and the computation on each set [2, 33]. Sets can be nodes, edges or faces. In these unstructured grids, the connectivity information is used to specify different mesh topologies. Figure 2.1 shows a mesh example that includes nodes and faces as sets. The value of data associated with each set is shown below each set and the mesh is represented by the connections between them.

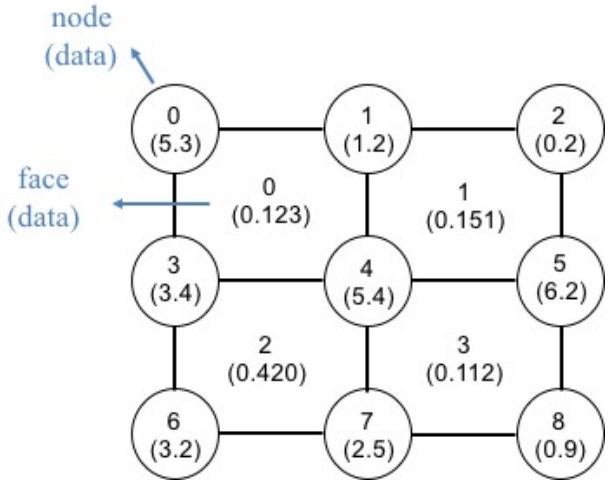


FIGURE 2.1. The mesh represented data layouts provided with OP2.

OP2 API for the mesh in figure 2.1 is shown as follows, which is the C/C++ API and defines 4 edges, 8 boundary edges and 9 nodes:

```
op_set nodes=op_decl_set(9,"nodes");
op_set edges=op_decl_set(4,"edges");
op_set bedges=op_decl_set(8,"bedges");
```

The mapping that declares the connection between 2 nodes is defined as follow:

```
int edge_map[28] = {0,1,1,2,2,5,5,4,4,3,3,6,6,7,7,
8,0,3,1,4,4,7,5,8}
op_map pedge=op_decl_map(edges,nodes,2,edge_map,pedge,"pedge");
```

Also the following mapping declares how 4 nodes are mapped on 1 face:

```
int face_map[16]={0,1,4,3,1,2,5,4,3,4,7,6,4,5,8,7};
op_map pface=op_decl_map(faces,nodes,4,face_map,"pface");
```

The values of each node and face are assigned as follow:

```
float vFaces[4] = {0.123, 0.151, 0.420, 0.112};
op_dat data_faces=op_decl_dat(faces,1,"float",vFaces,"data_faces");

float vNodes[9] = {5.3,1.2,0.2,3.4,5.4,6.2,3.2,2.5,0.9};
op_dat data_nodes=op_decl_dat(nodes,1,"float",vNodes,"data_nodes");
```

For this simple example assume that we want to update the value of all *face*s with summing values of its surrounding 4 nodes. The simple loop for this implementation is shown as follow:

```
for(int i = 0; i < number_of_faces; i++)
{
   faces[i] = faces[i] + node[faces[4*i]] + node[faces[4*i + 1]] +
      node[faces[4*i + 2]] + node[faces[4*i + 3]];
}
```

This loop can be implemented in parallel by calling *op_par_loop* as follow and passing *sum* as a user's kernel function as its first arguments:

```
op_par_loop_sum("sum",cells,
        op_arg_dat(faces,-1,OP_ID,1,"float",OP_WRITE),
        op_arg_dat(faces,-1,OP_ID,1,"float",OP_READ),
        op_arg_dat(nodes,0,pface,1,"float",OP_READ),
        op_arg_dat(nodes,1,pface,1,"float",OP_READ),
        op_arg_dat(nodes,2,pface,1,"float",OP_READ)
        op_arg_dat(nodes,3,pface,1,"float",OP_READ));
```

Each argument passed to each loop is generated based on data values used with *op_arg_dat*. These arguments are explicitly indicated that how each of the underlying data can be accessed inside a loop: *OP_READ* (read only), *OP_WRITE* (write) or *OP_INC* (increment to avoid race conditions due to indirect data access) [1]. For example, in *op_arg_dat(nodes,0,pface,1,"double",OP_READ)* , *OP_READ* marks the data as *read_only*. This argument is created from its inputs, where *nodes* is the data, 0 indicates that the data is accessed indirectly and it is the first index value on a face, *pface* is the mapping between the data, 1 is the data dimension, and *float* is the data type.

There are two different kinds of loop defined in OP2: $indirect$ and $direct$ loops. A loop is an $indirect$ loop if data is accessed through a mapping. Otherwise it is a $direct$ loop. Using map as a third argument in an *op_par_loop* indicates indirect accessed argument and using *OP_ID* indicates directly accessed argument. The more details about OP2 design and performance analysis can be found in [1] and [33], which shows that all unstructured grid applications can be easily described with sets and meshes as shown in the above example. These methods place no restriction on the algorithm and they allow the programmer to choose unique operations on each set.

```
op_par_loop_save_soln("save_soln", cells,
  op_arg_dat(p_q,-1,OP_ID,4,"double",OP_READ),
  op_arg_dat(p_qold,-1,OP_ID,4,"double",OP_WRITE));

op_par_loop_adt_calc("adt_calc",cells,
  op_arg_dat(p_x,0,pcell,2,"double",OP_READ),
  op_arg_dat(p_x,1,pcell,2,"double",OP_READ),
  op_arg_dat(p_x,2,pcell,2,"double",OP_READ),
  op_arg_dat(p_x,3,pcell,2,"double",OP_READ),
  op_arg_dat(p_q,-1,OP_ID,4,"double",OP_READ),
  op_arg_dat(p_adt,-1,OP_ID,1,"double",OP_WRITE));
```

FIGURE 2.2. The OP2 API functions *op_par_loop_save_soln* and *op_par_loop_adt_calc* represent two loops from an Airfoil application. *op_par_loop_save_soln* creates a $direct$ loop and *op_par_loop_adt_calc* creates an $indirect$ loop.

2.2.2 Airfoil Application

In this research, we study an Airfoil application, which is a standard unstructured mesh, finite volume, computational fluid dynamics (CFD) code, presented in [34], for the turbomachinery simulation and consists of over $720K$ nodes and about 1.5 million edges. As described in [34] and [25], it has both $direct$ and $indirect$ loops. We demonstrate $direct$ and $indirect$ loops in figure 2.7, which includes *op_par_loop_save_soln* and *op_par_loop_adt_calc* loops from Airfoil: *op_par_loop_save_soln* is a $direct$ loop that applies *save_soln* on the cells based on the *p_q* and *p_qold* arguments, and *op_par_loop_adt_calc* is an $indirect$ loop that applies *op_par_loop_adt_calc* on the cells based on *p_x*, *p_q*, and *p_adt* arguments passed to the loop. All of the computations on each set are implemented within these loops by performing operations of the user's kernels defined in a header file for each loop: *save_soln.h* for *op_par_loop_save_soln* and *adt_calc.h* for *op_par_loop_adt_calc*. More details can be found in [2] and [32].

OP2 framework is designed to obtain an optimum scaling on multi-core processors [3, 4]. However, as all of its analysis information are static [5–7], so they are not enough to achieve desired parallelism level. For this purpose, OP2 needs to be able to extract parallelism automatically at runtime. This challenge is addressed in this research by delaying some compiler optimizations to be managed at runtime. In this chapter, we propose different optimization methods that pro-

vide dynamic information for the parallelization techniques provided by the OP2 compiler. These techniques makes an OP2 to provide asynchronous task execution, interleaving different loops together, dynamically setting chunk sizes of different dependent loops based on each other, and prefetching data. These proposed techniques are implemented using HPX runtime system via re-designing the OP2 framework in a way that employs both compiler's static analysis and dynamic runtime information. The evaluation results in section 2.5 illustrate that HPX's parallelization methods helps eliminating unnecessary synchronization barriers and enables OP2 to extract more parallelism for the parallel unstructured grids applications. In the next section, we briefly discuss about how HPX provides the scalable framework that reduces the synchronizations and the latencies for implementing these proposed methods.

2.3 HPX

The ParalleX execution model [35, 36] addresses the challenges using conventional parallel techniques by implementing new forms of fine-grain task parallelism and enabling light weight synchronized message driven computation in a global address space. Its main goal is to improve efficiency and scalability compared to the conventional programming practices such as MPI. This goal is achieved in ParalleX model by reducing synchronization and increasing resource utilization through providing asynchronization and employing adaptive scheduling.

HPX is the first implementation of ParallelX model and it is a parallel C++ runtime system that facilitates distributed operations and enables fine-grained task parallelism resulting in a better load balancing [28, 29]. It provides an efficient scalable parallelism by significantly reducing processor starvation and effective latencies while controlling overheads [30]. HPX [19] helps overcoming difficulties faced using conventional techniques by exposing a programming model that intrinsically reduces the SLOW factors [22, 37]. SLOW represents the main factors which inhibit the scalability of applications and they can be described as follow:

A. **S**tarvation: poor utilization of the resources caused by a lack of available work.

B. **L**atencies: time-distance delay of accessing remote resources.

C. **O**verhead: costs for managing parallel actions.

D. **W**aiting: costs imposed by oversubscription of the shared resources.

Also, HPX uses the light-weight threads, which have extremely short context switching times that results in reduced latencies even for very short operations.

Concurrency is defined to have several simultaneously concurrent computations touching same data, while on the other hand, by parallelism we refer to simultaneous execution of independent tasks [19]. HPX's design focuses on parallelism rather than concurrency. As a results, HPX is able to expose both time and spatial parallelization [37], which enables asynchronous task execution.

In this research different dynamic optimizations are proposed for improving the performance of code generated by the OP2 compiler that are implemented using HPX runtime system, which has been developed to overcome limitations such as global barriers and poor latency hiding [29, 30] by embracing new ways of coordinating parallel execution, controlling synchronization, and implementing latency hiding utilizing Local Control Objects (LCO) [25, 38]. These objects have the ability to create, resume, or suspend a thread when triggered by one or more events. LCOs provide traditional concurrency control mechanisms such as various types of mutexes, semaphores, spinlocks, condition variables and barriers in HPX. These objects improve the efficiency of an application by permitting highly dynamic flow control as they organize the execution flow, omit global barriers, and enable thread execution to proceed as far as possible without waiting. More details about LCO design and its performance can be found in [22, 30, 39].

The two implementations of LCOs most relevant to this research are the $future$ construct and the *dataflow* template. HPX provides a multi-threaded, message-driven, split-phase transaction, and distributed shared memory programming model using $futures$ and *dataflow* based synchronization on the large distributed system architectures, which are explained in the following sections.

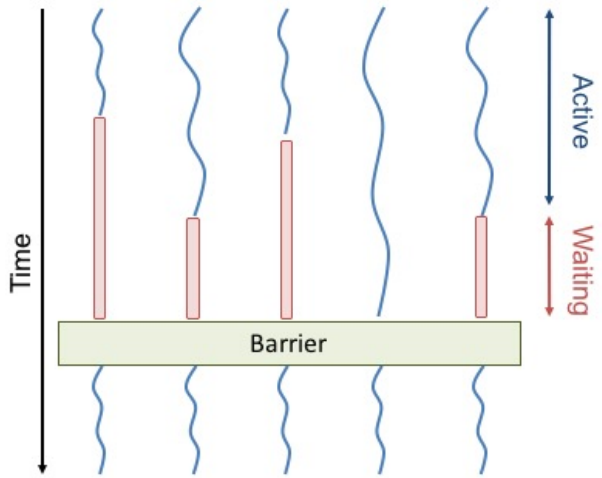


FIGURE 2.3. The global barrier prevents the progress of computation until all threads reaches that barrier.

2.3.1 Future

One of the main challenges faced in the conventional parallelization techniques is the existence of the implicit or explicit global barriers. This barrier prevents the progress of computation until all threads reaches that barrier. Then the reduction will be performed on one of those threads as shown in Fig. 2.3. HPX solves this challenge by eliminating unnecessary global barriers with implementing $future$-based parallelization techniques.

$future$ is a computational result that is initially unknown but becomes available at a later time [31]. The goal of using $future$ is to let every computation proceed as far as possible. Using $future$ enables threads to continue their executions without waiting for the results of the previous steps to be completed, which eliminates the implicit global barrier at the end of the execution of an OpenMP parallel loop. $future$ based parallelization provides the rich semantics for exploiting higher level parallelism available within each application that may significantly improve its scalability.

Figure 3.7 shows the scheme of the $future$ performance with 2 *localities*, where a *locality* is a collection of processing units (PUs) that have access to the same main memory. It illustrates that the other threads do not stop their progress even if the thread, which waits for the value to be computed, is suspended. Threads access a $future$ value by performing $future.get()$. When the

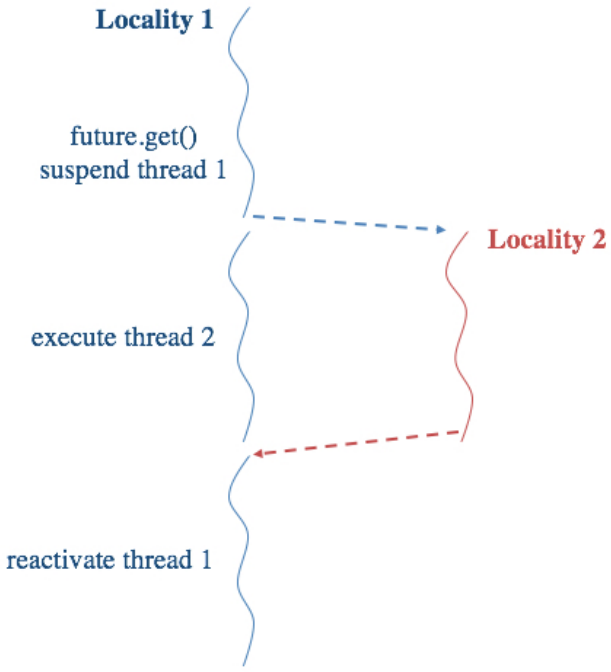


FIGURE 2.4. The principle of the operation of the $future$ in HPX. Thread 1 is suspended only if the results from locality 2 are not readily available. Thread 1 accesses the $future$ value by performing $future.get()$. If results are available, Tread 1 continues to complete the execution.

result becomes available, the $future$ resumes all HPX suspended threads waiting for that value. It can be seen that this process eliminates the global barrier synchronizations, as only those threads that depend on the $future$ value are suspended. With this scheme, HPX allows asynchronous execution of the threads.

2.3.2 Dataflow Object

dataflow object provides a powerful mechanism for managing data dependencies without the use of global barriers [28, 40]. Figure 2.5 shows the schematic of a *dataflow* object, which encapsulates a function $F(in_1, in_2, ..., in_n)$ with n future or non-future inputs from different data resources. If an input is a future, then the invocation of the function will be delayed. Non-future inputs are passed through. A *dataflow* object waits for a set of $futures$ to become ready and as soon as the last input argument has been received, the function F is scheduled for the execution [22]. Because the *dataflow* object returns a $future$, its result can be fed to other objects in the system including other *dataflows*. These chained $futures$, by their nature, represent a dependency

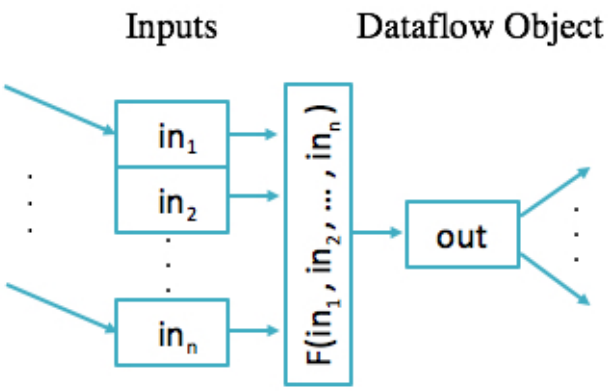


FIGURE 2.5. A *dataflow* object encapsulates a function $F(in_1, in_2, ..., in_n)$ with n inputs from different data resources. As soon as the last input argument has been received, the function F is scheduled for an execution.

tree that automatically generates an execution graph. This graph is executed by the runtime system as each nodes dependencies are meet. As a result, *dataflow* minimizes the total synchronization by scheduling new tasks as soon as they can be run instead of waiting for entire blocks of tasks to finish computation.

2.4 HPX Implementation on an Airfoil application

As described in section 2.2, the Airfoil application [34] uses an unstructured grid and consists of five parallel loops: *op_par_loop_save_soln*, *op_par_loop_adt_calc*, *op_par_loop_res_calc*, *op_par_loop_bres_calc*, *op_par_loop_update*, of which *op_par_loop_save_soln* and *op_par_loop_update* loops are $direct$ loops and the others are $indirect$ loops, shown in figure 2.6. Saving old data values, applying computation on each data value and updating them are implemented within these five loops. Each loop iterates over a specified data set and performs the operations with the user's kernels defined in a header file for each loop: *save_soln.h*, *adt_calc.h*, *res_calc.h*, *bres_calc.h* and *update.h*.

Figure 2.7 demonstrates *op_par_loop_save_soln* that applies *save_soln* on the cells based on the arguments generated with *op_arg_dat* using *p_q* and *p_qold* data values. The function *op_arg_dat* creates an OP2 argument based on the information passed to it and the function *op_par_loop* creates a parallel loop for the computations over the sets through for each loop.

```
op_par_loop_save_soln("save_soln", cells,
  op_arg_dat(data_a0,...),...,
  op_arg_dat(data_an,...);

op_par_loop_adt_calc("adt_calc",cells,
  op_arg_dat(data_b0,...),...,
  op_arg_dat(data_bn,...);

op_par_loop_res_calc("res_calc",edges,
  op_arg_dat(data_c0,...),...,
  op_arg_dat(data_cn,...);

op_par_loop_bres_calc("bres_calc",bedges,
  op_arg_dat(data_d0,...),...,
  op_arg_dat(data_dn,...);

op_par_loop_update("update",cells,
  op_arg_dat(data_e0,...),...,
  op_arg_dat(data_en,...);
```

FIGURE 2.6. Five loops are used in *Airfoil.cpp* for saving old data values, applying computation, and updating each data value. *save_soln* and *update* loops are $direct$ loops and the others are $indirect$ one.

```
op_par_loop_save_soln("save_soln", cells,
op_arg_dat(p_q,-1,OP_ID,4,"double",OP_READ),
op_arg_dat(p_qold,-1,OP_ID,4,"double",OP_WRITE));
```

FIGURE 2.7. *op_par_loop_save_soln* represents one of the loops used in an Airfoil application.

```
void op_par_loop_save_soln(char const *name,
  op_set set, op_arg arg0, op_arg arg1)
{
.
.
.
#pragma omp parallel for
for(int blockIdx=0; blockIdx<nblocks; blockIdx++)
{
  int blockId  = //based on the blockIdx
  int nelem    = //based on the blockId
  int offset_b = //based on the blockId

  for (int n=offset_b; n<offset_b+nelem; n++)
  {
    .
    .
    .
    save_soln(...); //user's kernel
  }}
 }
```

FIGURE 2.8. *#pragma omp parallel for* is used for a loop parallelization for an Airfoil application.

The loop parsed with OP2 in figure 2.8 illustrates how each cell updates its data value by accessing *blockId*, *offset_b*, and *nelem* data elements. The arguments are passed to the *save_soln* user kernel subroutine, which does the computation for each iteration of an inner loop from *offset_b* to *offset_b+nelem* of each iteration of an outer loop from *0* to *nblocks*. Also it illustrates that OpenMP is used for the parallel processing within a node. It is important to note that the outputs of the computations shown in figure 2.6 cannot be passed to the outside of the loop, therefore, the current OP2 design doesn't provide a method for interleaving loops together. This creates implicit global barrier after each loop as the threads inside the loop must wait to synchronize before exiting the loop [13]. Barriers, naturally, impede optimal parallelization by causing the parallel threads and processes to wait. In order to solve this problem, this research sets out to optimize the performance of code generated by the OP2 compiler using the HPX runtime, which are three different HPX parallelization methods:

1. using parallel *hpx::for_each* for parallelizing loops generated with OP2,
2. using parallel *hpx::for_each* while enabling asynchronous task execution, and
3. using *dataflow* object.

The comparison results of these methods with OpenMP are studied in Section 2.5.

2.4.1 *hpx::for_each*

The recently published C++ Extensions for Parallelism TS [41] specifies a comprehensive set of parallel algorithms for inclusion into the C++ standard library and applies the proven design concepts of the Standard Template Library to the parallel loop executions [19]. Almost all of these parallel algorithms are implemented with HPX, which have been extended to support the full spectrum of execution policies, combined with the full set of executors and executor parameters [37]. HPX provides the higher-level abstractions based on and derived from the current C++ standard.

hpx::for_each is one of the HPX parallel algorithms that is able to automatically control the chunk size during the execution. The *auto_partitioner* executor exposed by the *hpx::for_each*

TABLE 2.1. The execution policies defined by the Parallelism TS and implemented in HPX [41].

Policy	Description	Implemented by
seq	sequential execution	Parallelism TS, HPX
par	parallel execution	Parallelism TS, HPX
par_vec	parallel and vectorized execution	Parallelism TS
seq(task)	sequential and asynchronous execution	HPX
par(task)	parallel and asynchronous execution	HPX

algorithm is able to automatically control the amount of work spawned by each chunk by sequentially executing 1% of the loop. Therefore *hpx::for_each* helps creating sufficient parallelism by determining number of the iterations to be run on each HPX thread. This optimization can be used for dynamically determining efficient chunk sizes of the loops generated with *op_par_loop* by OP2. Moreover, HPX is able to execute loops in sequential or in parallel by applying *execution_policies*, which are briefly described in Table 2.1 [22]. The concept of the *execution_policy* developed in HPX is used to specify the execution restrictions of the work items, in which calling with a sequential execution policy makes the algorithm to be run sequentially and calling with a parallel execution policy allows the algorithm to be run in parallel [38].

In this section, we use *par* as an execution policy while executing *hpx::for_each* to make the loops shown in figure 2.6 executing in parallel. We modified the OP2 source-to-source translator with Python to automatically produce *hpx::for_each* instead of using *#pragma omp parallel for* for the loop parallelization. In this method *Airfoil.cpp* (figure 2.6) and the OP2 API remain unchanged. Figure 2.9 shows the loop of *op_par_loop_adt_calc* function parsed with OP2 using HPX for the loop parallelization.

It should be considered that if the computational time of a loop is too small, using auto-partitioner algorithm within HPX will not be efficient. Since for the small loops, 1% execution time of the loop used for determining grain size will affect the application's scalability. For solving this problem, HPX provides another way to avoid degrading scalability while using *hpx::for_each*. Grain size can be specified as the static chunk size with *hpx::for_each(par.with(scs))*

```
void op_par_loop_save_soln(char const *name,
  op_set set, op_arg arg0, op_arg arg1)
{
.
.
.
auto r = boost::irange(0, nblocks);
hpx::parallel::for_each(par, //execution policy
  r.begin(), r.end(), [&](std::size_t blockIdx) {

  int blockId  = //based on the blockIdx in OP2 API
  int nelem    = //based on the blockId
  int offset_b = //based on the blockId

  for ( int n=offset_b; n<offset_b+nelem; n++ ){
    .
    .
    .
    save_soln(...); //user's kernel
  }});
 }
```

FIGURE 2.9. Implementing *hpx::for_each* for the loop parallelization in OP2. HPX is able to control the grain size with this method. As a result, it helps in reducing processor starvation caused by the fork-join barrier at the end of the execution of the parallel loops.

```
void op_par_loop_save_soln(char const *name,
  op_set set, op_arg arg0, op_arg arg1)
{
.
.
.
static_chunk_size scs(SIZE);
auto r=boost::irange(0, nblocks);
hpx::parallel::for_each(par.with(scs),
  r.begin(), r.end(), [&](std::size_t blockIdx) {

  int blockId  = //based on the blockIdx in OP2 API
  int nelem    = //based on the blockId
  int offset_b = //based on the blockId

  for ( int n=offset_b; n<offset_b+nelem; n++ ){
   .
   .
   .
   save_soln(...); //user's kernel
  }});
 }
```

FIGURE 2.10. Implementing *hpx::for_each* for loop parallelization in OP2. HPX is able to avoid degrading scalability for small loops by defining static grain size with *dynamic_chunk_size scs(SIZE)* before the parallel loop execution.

before executing loop; *scs* is defined by *static_chunk_size scs(SIZE)*. Figure 2.10 shows the loop of *op_par_loop_adt_calc* function with static *chunk_size* implemented with *static_chunk_size scs(SIZE)*. In section 2.5 the performance of these two methods are studied.

So, what we achieve here by implementing *hpx::for_each* within the loop execution is enabling compiler to control the chunk sizes dynamically. However, this method exposes the same disadvantage as OpenMP implementation, which is representation of the fork-join parallelism that introduces the global barriers at the end of a parallel loop. In the next sections, we show how we overcome to this problem.

2.4.2 *async* and *hpx::for_each*

In this section we solve the problem described in the previous section by using asynchronous task execution while controlling chunk sizes dynamically within the loop executions. Asynchronous task execution means that a new HPX-thread will be scheduled. As a result it eliminates the global barrier synchronizations when using *hpx::for_each*. For this purpose, we implement two different parallelization methods for the loops, based on their types:

1. For the $direct$ loops, *async* and *hpx::for_each* with *par* as an execution policy is implemented.
2. For the $indirect$ loops, *hpx::for_each* with *par(task)* as an execution policy is implemented.

The calls to *async* and *par(task)* provide a new $future$ instance, which represents the result of the function execution, making invocation of the loop asynchronous. *future* based parallelization provides rich semantics for exploiting higher level parallelism available within each application that may significantly improve the scaling.

OP2 source-to-source translator with Python is modified and it automatically produces *async* and *hpx::for_each* with *par* for each direct loops and *hpx::for_each* with *par(task)* for each $indirect$ loops within an Airfoil application. For example, in figure 2.11, *async* and *hpx::for_each* with *par* is used for *op_par_loop_save_soln*, which is a $direct$ loop and returns a $future$ representing result of a function. Also, in figure 2.12, *hpx::for_each* with *par(task)* is used for *op_par_loop_adt_calc*, which is an $indirect$ loop and returns a $future$ representing result of a function. The *futures* returned from all $direct$ and $indirect$ loops allow asynchronous execution of the loops. In this method, OP2 API is not changed but *Airfoil.cpp* is changed as shown Figure 2.13. Each kernel function within *op_par_loop* returns a *future* stored in a *new_data*. Each *future* depends on the *future* in a previous step. So, *new_data.get()* is used to get all $futures$ ready before the next steps.

So, what we achieve here by implementing *async* within the loop execution is enabling compiler to:

```
hpx::shared_future<op_dat> op_par_loop_save_soln(
char const *name, op_set set, op_arg arg0, op_arg arg1) {
.
.
.
return async(hpx::launch::async,
  [adt_calc,set,arg0,...,argn1](){

    auto r=boost::irange(0, nthreads);
    hpx::parallel::for_each(par,
    r.begin(), r.end(),[&](std::size_t thr){

    int start  =//based on the number of threads;
    int finish =//based on the number of threads;

    for ( int n=start; n<finish; n++ ){
     save_soln(...);
    }
  });
}
```

FIGURE 2.11. Implementing *async* and *hpx::for_each* with *par* for a $direct$ loop parallelization in OP2. The returned $future$ representing result of a function.

```
hpx::shared_future<op_dat> op_par_loop_adt_calc(
char const *name, op_set set, op_arg arg0, op_arg arg1,
op_arg arg2, op_arg arg3, op_arg arg4, op_arg arg5) {
.
.
.
auto r=boost::irange(0, nblocks);
hpx::future<void> new_data;
new_data=hpx::parallel::for_each(par(task),
r.begin(), r.end(),[&](std::size_t blockIdx){

  int blockId  =//based on the blockIdx in OP2 API
  int nelem    =//based on the blockId
  int offset_b =//based on the blockId

  for ( int n=offset_b; n<offset_b+nelem; n++ ){
    .
    .
    .

    adt_calc(...);
  }
 });
 return new_data;
}
```

FIGURE 2.12. Implementing *hpx::for_each* with *par(task)* for an $indirect$ loop parallelization in OP2. The returned $future$ representing result of a function.

```
new_data1=op_par_loop_save_soln("save_soln",cells,
  op_arg_dat(data_a0,...),...,
  op_arg_dat(data_an,...);

new_data2=op_par_loop_adt_calc_("adt_calc",cells,
  op_arg_dat(data_b0,...),...,
  op_arg_dat(data_bn,...);

new_data2.get();

new_data3=op_par_loop_res_calc("res_calc",edges,
  op_arg_dat(data_c0,...),...,
  op_arg_dat(data_cn,...);

new_data4=op_par_loop_bres_calc("bres_calc",bedges,
  op_arg_dat(data_d0,...),...,
  op_arg_dat(data_dn,...);

new_data3.get();
new_data4.get();

new_data5=op_par_loop_update("update",cells,
  op_arg_dat(data_e0,...),...,
  op_arg_dat(data_en,...);

new_data1.get();
new_data5.get();
```

FIGURE 2.13. *Airfoil.cpp* is changed while using *async* and *par(task)* for loop parallelization in OP2. *new_data* is returned from each kernel function after calling *op_par_loop* and *new_data.get()* is used to get $futures$ ready before the next steps.

```
using hpx::lcos::local::dataflow;
using hpx::util::unwrapped;

//automatically returns the argument as a future
return dataflow(unwrapped([&](data_a,...){
    //same as original op_arg_dat
    return arg;
    }
  }),data_a,...);
```

FIGURE 2.14. *op_arg_dat* is modified to create an argument as a $future$ that is passed to a function through *op_par_loop* shown in figure 2.6.

✓ set the chunk sizes dynamically, and

✓ produce an asynchronous loop execution.

The place of *new_data.get()* depends on the application and the programmer should put them manually in the correct place by considering data dependencies between the loops. In order to completely generate a code automatically for implementing this method without using programmer's help, we change the OP2 API and we implement a $dataflow$ method in HPX, which is studied in more details in the next section.

2.4.3 Dataflow

In this section, the new method is proposed for parallelizing loops generated with OP2, which is based on *dataflow* implementation that solves the current challenges of OP2. In this method, the OP2 API is modified in such a way that *op_arg_dat* used in each loop in figure 2.6 produces an argument as a $future$ for *dataflow* object inputs. Figure 2.14 shows the modified *op_arg_dat*, in which *data_a,...* expressed at the last line of the code invokes a function only once all of them get ready. *unwrapped* is a helper function in HPX, which unwraps the $futures$ and passes along the actual results. This implementation also generates an output argument as a $future$ and as a result, all of the arguments of each loop in figure 2.6 are passed as a $future$ to the kernel function through *op_par_loop*.

2.4.3.1 Parallelizing Loops Using *for_each*

Parallelizing loops and controlling chunk sizes are implemented by using *for_each* algorithm and *persistent_auto_chunk_size* as an *execution_policy* respectively. In figure 2.15, *dataflow* is implemented with *for_each* for the loop in figure 2.8, that aids to parallelize the outer loop. It also illustrates that $arg0$ and $arg1$, which are created as a $future$ with *op_arg_dat* using *p_q* and *p_qold* respectively, are passed as a $future$ within a loop. This loop will be executed only if these arguments get ready. Then, the output argument, which is $arg1$ in this example, is passed as a $future$ to the outside of the loop and it is stored within *p_qold* shown in figure 2.16. This method is implemented to all of the loops in figure 2.6, and as a result, each kernel function returns an output argument as a $future$. The loop execution may depend on the results of the other previous loops. So by this method, the results of the loops can be passed as $future$ inputs to the other loops, which makes OP2 able to interleave different loops. For example, *p_qold* value updated in *op_par_loop_save_soln* is used as an input argument for *op_par_loop_update* as shown in figure 2.17, which using this proposed method makes it able to interleave this two loops together by passing output of *op_par_loop_save_soln* as an input argument for *op_par_loop_update*.

Figure 2.18 shows generally that by implementing proposed method, the $future$ output of each loop passed as an input of the other loops makes OP2 able to interleave different loops together at runtime. As a result, if the loops are not dependent on each other, they can be executed without waiting for the previous loops to complete their tasks, however, if they depend on the parameters from the previous loops, they will wait until the previous loops complete their processes. This proposed method removes the unnecessary barrier synchronizations between different loops and execute them asynchronously.

2.4.3.2 Controlling Chunk Sizes

As it is explained in section 2.4.3.1, figure 2.18 shows how *dataflow* provides a way of interleaving execution of different loops together. In a case of having dependent loops, the execution of each chunk in a loop depends on the execution of the chunks in the previous loop. By using

```
hpx::shared_future<op_dat> op_par_loop_save_soln(char const * name,
    op_set set,
     hpx::future<op_arg> arg0,
     hpx::future<op_arg> arg1)
{
using hpx::lcos::local::dataflow;
using hpx::util::unwrapped;
//automatically returns output as a future
return dataflow(unwrapped([&save_soln]
(op_arg arg0, op_arg arg1){
    .
    .
    .
  auto r=boost::irange(0, nblocks);
  hpx::parallel::for_each(policy,
  r.begin(), r.end(),
  [&](std::size_t blockIdx){

  for (int n=offset_b; n<offset_b+nelem; n++)
  {
    .
    .
    .
    save_soln(...);
  }
  return arg1;
}),arg0,arg1);
}
```

FIGURE 2.15. Implementing *for_each* within *dataflow* for the loop parallelization in OP2 for the loop in figure 2.8. It makes the invocation of a loop asynchronous by returning output as a $future$. *dataflow* allows automatically creating execution graph, which represents a dependency tree.

```
p_qold =   op_par_loop_save_soln("save_soln",cells,
           op_arg_dat(p_q,-1,OP_ID,4,"double",OP_READ),
           op_arg_dat(p_qold,-1,OP_ID,4,"double",OP_WRITE));
```

FIGURE 2.16. *Airfoil.cpp* is changed while using *dataflow* for the loop parallelization in OP2. *p_qold* is returned as a $future$ from each kernel function after calling *op_par_loop*.

```
p_qold = op_par_loop_save_soln("save_soln",cells,
           op_arg_dat(p_q,-1,OP_ID, 4,"double",OP_READ),
           op_arg_dat(p_qold,-1,OP_ID,4,"double",OP_WRITE));

p_q = op_par_loop_update("update",cells,
           op_arg_dat(p_qold,-1,OP_ID,4,"double",OP_READ),
           op_arg_dat(p_q,-1,OP_ID,4,"double",OP_WRITE),
           op_arg_dat(p_res,-1,OP_ID,4,"double",OP_RW),
           op_arg_dat(p_adt,-1,OP_ID,1,"double",OP_READ),
           op_arg_gbl(&rms,1,"double",OP_INC));
```

FIGURE 2.17. The proposed method makes OP2 able to interleave thses two loops together by passing *p_qold* output of *op_par_loop_save_soln* as an input argument for *op_par_loop_update.*

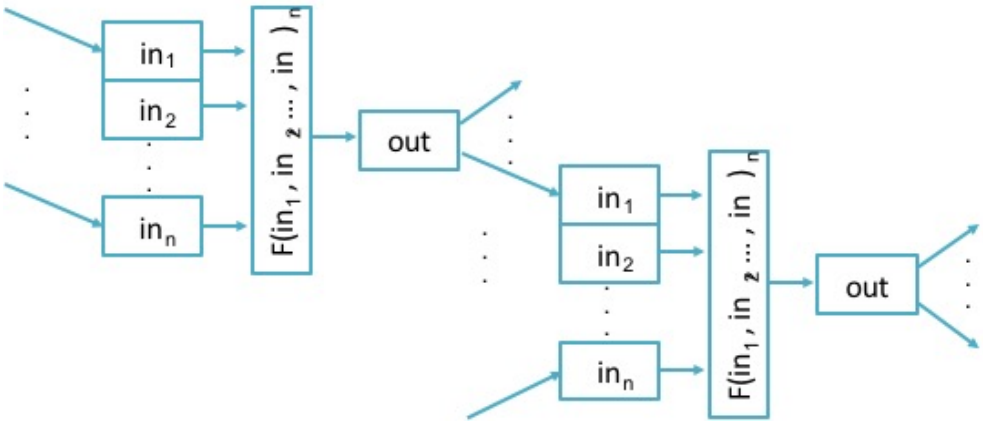
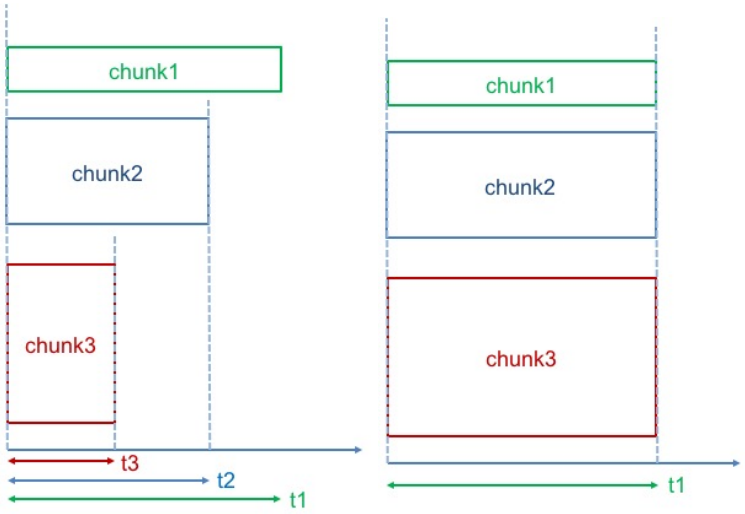


FIGURE 2.18. *dataflow* provides a way for interleaving execution of different loops together by generating output as a $future$ and passing all inputs as $future$s as well.



(a) Chunk sizes with different execution time

(b) Chunk sizes with the same execution time

FIGURE 2.19. Setting chunk sizes of different dependent loops based on each other.

par as an execution policy, different chunks with different execution time regardless of the chunk sizes of the other loops are determined for each loop shown in figure 2.19a, which may increase the waiting time between them. So for decreasing this waiting time, the execution time of each chunk in these dependent loops should be the same. For this purpose, the new execution policy is proposed in this section, named *persistent_auto_chunk_size*, that makes all chunk sizes of different loops having same execution time as shown in figure 2.19b. In this policy, the chunk size of the first loop is determined automatically with *for_each* algorithm. Then the chunk sizes of each second and third loops are determined based on the execution time of the chunk in the first loop. As a result, all chunks of all these three loops will have the same execution time. It should be note that *chunk1*, *chunk2* and *chunk3* have different sizes but with the same execution time.

So, what we achieve here by implementing *dataflow* within the loop execution is enabling compiler to:

✓ set the chunk sizes dynamically,

✓ controlling chunk sizes of different dependent loops based on each other,

✓ produce an asynchronous loop execution, and

✓ interleave $direct$ and $indirect$ loops together without using programmer's help.

Although this method successfully provides an asynchronous task execution framework which increases the on-node parallelism available for the parallel applications generated with OP2, this method also increases the complexity of the memory hierarchy. One solution for this problem is data prefetching, which effectively reduces the memory accesses latencies. In this research, we add the generic prefetching scheme as a new feature to HPX, which is explained in more details in the next Section.

2.4.4 Prefetching Iterator Implemented in HPX

Data prefetching is one of the methods for reducing memory accesses latencies by calling data required for the next step into the cache [21]. The simplest form of the cache prefetching can be implemented by prefetching cache line of the next iteration as soon as the current cache line is referenced [20, 42]. Hardware, software and thread prefetching are different traditional techniques for this purpose.

Various hardware prefetching methods has been proposed that one of them is using one-block-lookahead (OBL) scheme [43]. In this method, the blocks $i+1$, $i+2$, ..., and $i+n$ are prefetched whenever the block i is brought to the cache that results in reducing cache misses significantly. Creating reference prediction table [44, 45] is another method to limit unnecessary prefetching and to predict the future memory references. However, one of the big challenges exists in most of these hardware prefetching methods is that the prefetcher uses the past access pattern by considering data stream, which cannot handle an irregular access pattern.

In the software prefetcher method, the prefetching data is implemented by using prefetch directives in the code. One of the problem of this method is that these prefetching instructions are inserted with programmer or compiler into the applications, which has the high probability of the cache miss occurrences. Another problem is introducing additional overhead for executing these prefetch instructions. There has been many developments proposed for optimizing this technique that mostly are obtained by prefetching pointer-based data structures [43, 46]. Mowry's algorithm [47] is one of the recent prefetching optimization that defines the affine array-references

as the prefetching candidates within an inner-most loop, performs the loop unrolling, and creates the multiple memory references within a loop. As a result, the exact missing instance is prefetched, which avoids the unnecessary prefetching and reduces prefetching overheads. Jump pointer prefetching [44, 45] is another proposed software prefetching approach, which is implemented by inserting additional pointers into a dynamic data structure for connecting non-consecutive elements within a loop. This technique allows prefetching data by creating pointer chain and results in overlapping fetching process of multiple elements simultaneously. However, this technique also has the difficulty in handling sequences of the irregular data accesses [20].

Thread based prefetching method is usually preferred over the software / hardware prefetching methods, since it precomputes the load addresses accurately and it is able to follow more complex patterns compared to the other methods [42]. This technique executes an application in the prefetcher thread context and brings data of the next cache line into the shared cache before the main thread accesses it. However, the scaling can be degraded with this method because of

1. cache misses: the prefetcher could make slower progress than the main thread, and
2. global barriers: a global barrier is needed to synchronize the prefetcher with the main thread [21, 42, 43].

In this section, the new prefetching method is introduced in HPX that combines a thread based prefetching with an asynchronous task execution. The main goal of this method is not only to reduce the memory accesses latencies, but also to relax the global barriers, which results in a better parallel performance. Figure 2.20 shows the scheme of using $future$ and the proposed prefetching iterator, which makes HPX to have the asynchronous execution while prefetching data of all the containers within a loop of the next step in to the cache memory in each iteration. Moreover, HPX is able to prefetch data in sequential or in parallel with applying *execution_policy* described in Table 2.1. This method is added to the method explained in section 2.4.3.1 to decrease the memory access latencies while parallelizing loops.

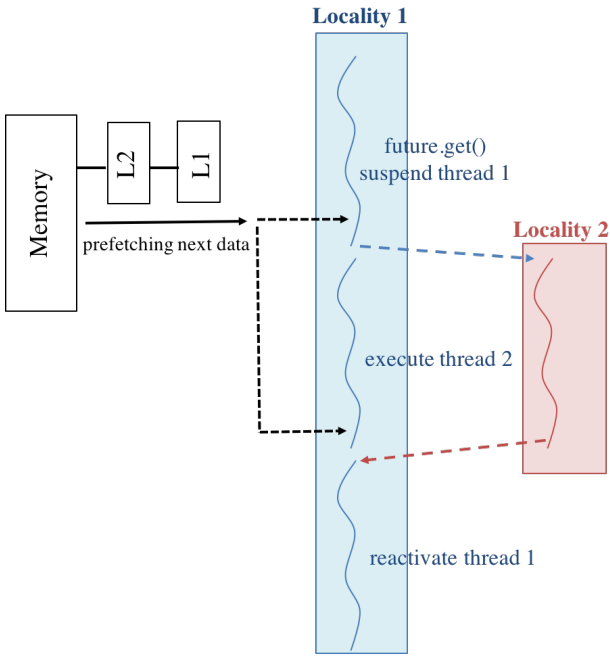


FIGURE 2.20. Data of the next iteration step is prefetched into the cache memory with the prefetching iterator called in each iteration within the *for_each*

```
hpx::parallel::for_each(hpx::parallel::execution::
    make_prefetcher_policy(policy, prefetch_distance_factor,
  container_1, container_2, ..., container_n)
        loop_range.begin(), loop_range.end(),
        [&](std::size_t i)
        {
            ...
        });
```

FIGURE 2.21. The prefetching method used in *for_each*. The prefetching iterator called with *for_each* is the struct that references to all container in the loop.

Figure 2.21 shows the details of the prefetching method implementation within *for_each*. The program execution is divided into several chunks within *for_each* and its iterator is developed to prefetch data of the next chunk size in either sequential or in parallel. The prefetching iterator is initialized and executed with using *make_prefetcher_policy*. This prefetching policy make *for_each* to act as a struct that references to all container in the loop, which enables it to prefetch data of these containers. *loop_range* is the range, in which the loop is executed. One of the feature of this prefetcher is that it works with any data types even in a case of having different type for each container. The distance between each two prefetching operations is computed based on the value of *prefetch_distance_factor*. In order to increase the effectiveness of the prefetcher and to decrease the relative cost, *prefetch_distance_factor* is designed to be determined based on the length of the cache line. As a result, within each prefetcher distance, data of all containers of the next time step are prefetched in each iteration by calling this prefetching iterator.

So, what we achieve here by implementing *dataflow* within the loop execution is enabling compiler to:

✓ set the chunk sizes dynamically,

✓ controlling chunk sizes of different dependent loops based on each other,

✓ produce an asynchronous loop execution,

✓ interleave $direct$ and $indirect$ loops together without using programmer's help, and

✓ automatically prefetch data of the next chunk size while executing loops.

The experimental results of optimizing OP2 performance with HPX discussed in this chapter are presented in the next section.

2.5 Experimental Results

In this section, we evaluate the experimental results of our work by comparing our proposed framework to OP2's current design. The main goal of this section is to illustrate that dynamic information obtained at runtime and static information obtained at compile time are both necessary to provide sufficient optimizations for optimal performance. The proposed methods studied in the

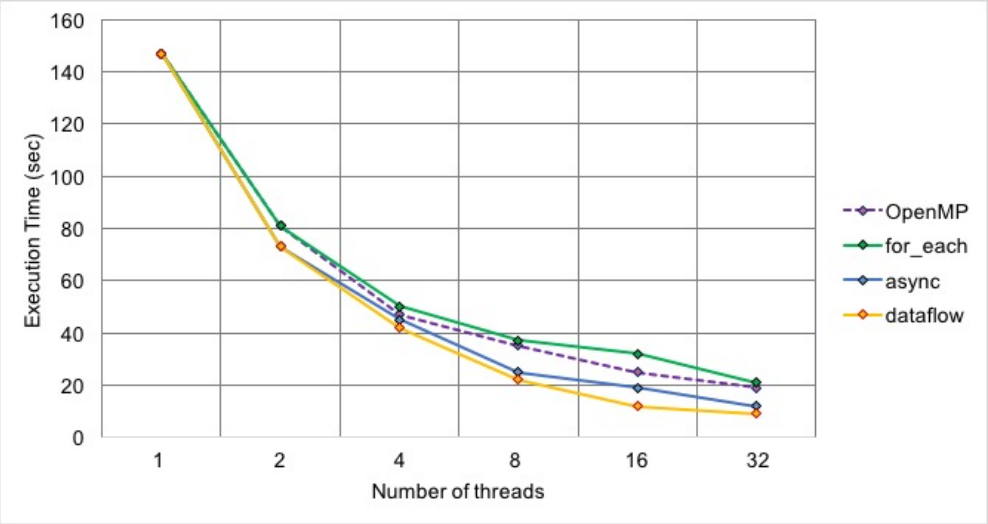


FIGURE 2.22. Comparison results of the execution time between *#pragma omp parallel for*, *hpx::for_each*, *async* and *dataflow* used for an Airfoil application.

previous sections are evaluated here. The experiments are executed on the test machine with two Intel Xeon E5-2630 processors, each with 8 cores clocked at 2.4GHZ and 65GB. Hyper-threading is enabled. The OS used by the shared memory system is 32 bit Linux Mint 17.2. and HPX 0.9.99 is used here.

2.5.1 Asynchronous Task Execution

Figure 3.9 shows the execution time of an Airfoil application using *#pragma omp parallel for*, *hpx::for_each*, *async* and *dataflow*, which illustrates that HPX and OpenMP has approximately the same performance on 1 thread. We are however able to improve parallel performance in using *async* and *dataflow* for more number of threads.

To evaluate the HPX performance of the loop parallelization generated with OP2, we perform the strong scaling and weak scaling experiments. For the speedup analysis, we use strong scaling, for which the problem size is kept the same as the number of cores are increased. Figure 3.4 shows shows the strong scaling comparison results for the following three loop parallelization methods: *#pragma omp parallel for*, *hpx::for_each(par)* with automatic chunk sizing for all loops and *hpx::for_each(par)* with the static chunk size for the small loops for an Airfoil application. It shows that although *hpx::for_each* helps creating sufficient parallelism by determining number

of the iterations to run on each thread, however using *hpx::for_each(par)* with the static chunk size for the small loops has better performance than automatically determining chunk sizes for all loops. Since, the computational time of the loop is so small, using *auto_partitioner* algorithm within HPX will not be efficient for determining chunk sizes. Using this method, OpenMP still performs better than HPX.

Figure 3.6 shows the strong scaling comparison results between *#pragma omp parallel for* and *async* and *hpx::for_each(par(task))* from section 2.4.2. The performance is better for *async* with *hpx::for_each(par(task))*, which is the result of the asynchronous execution of loops through the use of $futures$ that eliminates the global barrier synchronizations when using *hpx::for_each*. It proofs that the HPX *future* based parallelization method provides the rich semantics for exploiting higher level parallelism available within each application that improves the scaling.

Figure 3.5 shows the strong scaling comparison results between *#pragma omp parallel for* and *dataflow*. *hpx::for_each(par)* is used for the loop parallelization in this method as discussed in section 2.4.3. It illustrates a better performance for *dataflow* due to the asynchronous task execution and interleaving different dependent loops together. As described in section 2.3, *dataflow* automatically generated an (implicit) execution tree, which represents a dependency graph that results in removing unnecessary global barriers and improving scalability of the parallel applications.

By considering the above results, we can see the improvement in the performance over the OP2 (initial) version. For 32 threads in figure 3.6, *async* with *hpx::for_each(par(task))* improves the scalability by around 16% and in figure 3.5, *dataflow* improves the scalability by around 33% compared to using *#pragma omp parallel for*.

To study the effects of the communication latencies, we perform the weak scaling experiments, where the problem size is increased in proportion to the increase of the number of cores. Figure 3.8 shows the weak scaling in terms of the efficiency relative to the one core case using *#pragma omp parallel for*, *hpx::for_each(par)*, *async* with *hpx::for_each(par(task))* and *dataflow*. It is shown

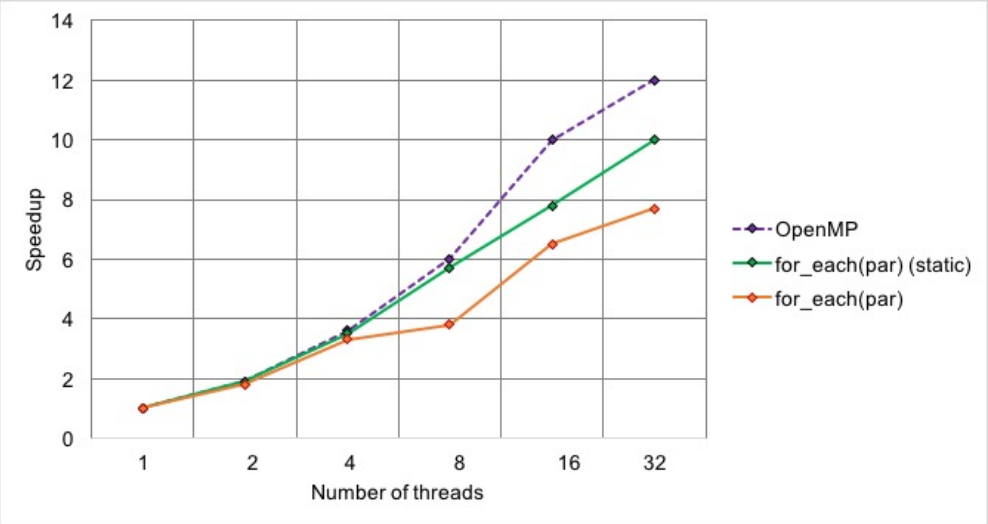


FIGURE 2.23. Comparison results of the strong scaling between *#pragma omp parallel for* and *hpx::for_each(par)* with the static and auto *chunk_size* used for an Airfoil application with up to 32 threads. HPX allows controlling grain size while using *hpx::for_each* to improve scalability. It shows a better performance for *hpx::for_each* with the static *chunk_size* compared to the auto *chunk_size* for small loops. Hyperthreading is enabled after 16 threads.

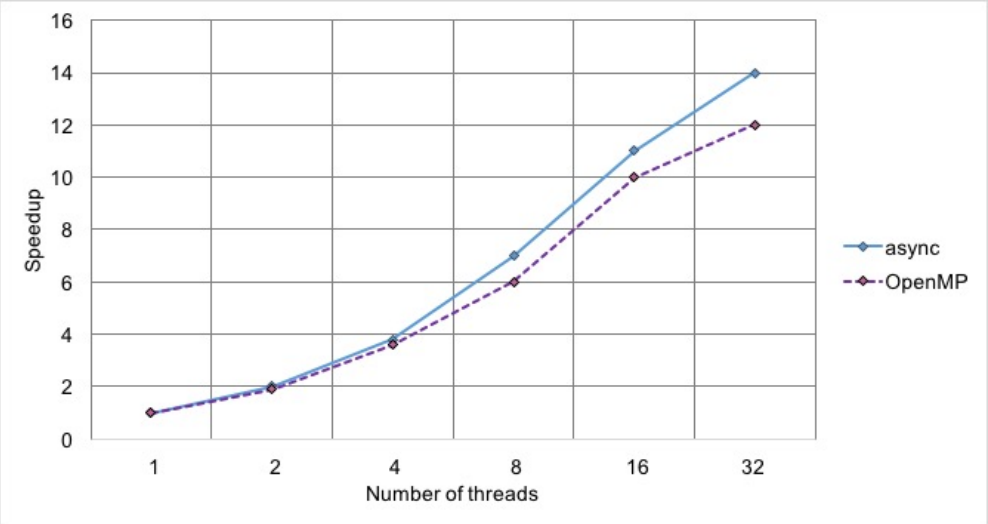


FIGURE 2.24. Comparison results of the strong scaling between *#pragma omp parallel for* and *async* with *hpx::for_each(par(task))* used for an Airfoil application with up to 32 threads. It shows a better performance for *async* due to the asynchronous task execution. Hyperthreading is enabled after 16 threads.

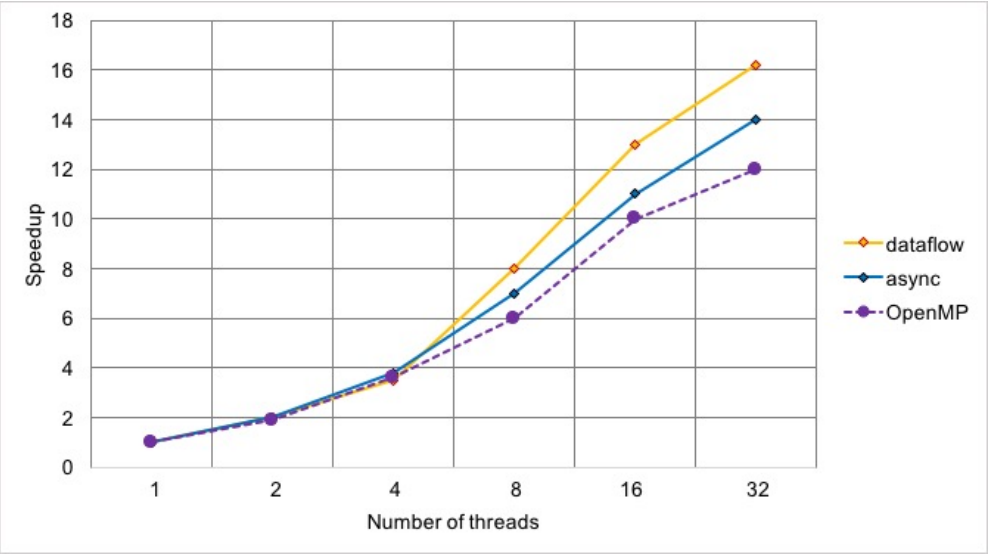


FIGURE 2.25. Comparison results of the strong scaling between *dataflow* and *#pragma omp parallel for* used for an Airfoil application with up to 32 threads. It illustrates a better performance for *dataflow* for a larger number of threads, which is due to the asynchronous task execution. *dataflow* automatically generates an execution tree, which represents a dependency graph and allows an asynchronous execution of the functions. Hyperthreading is enabled after 16 threads.

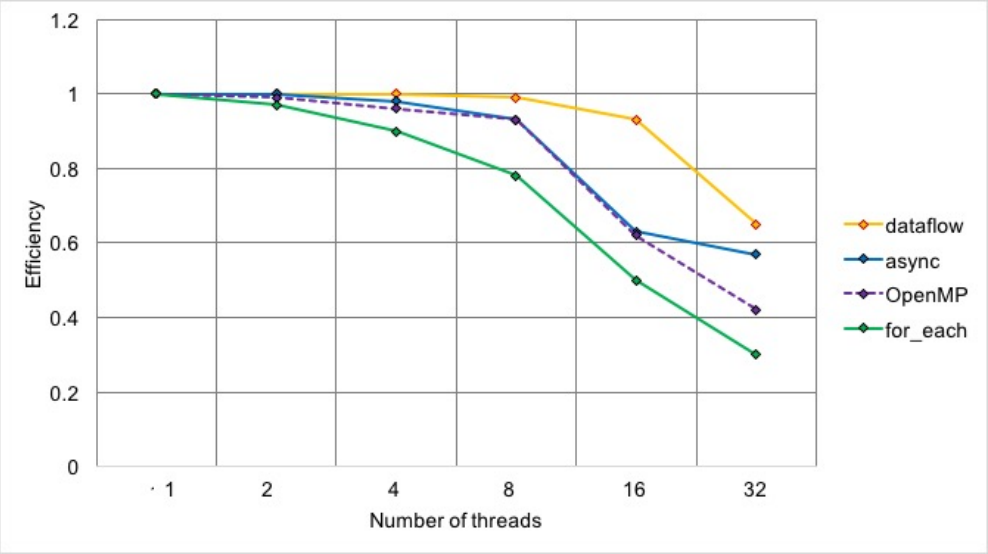


FIGURE 2.26. Comparison results of the weak scaling between *#pragma omp parallel for*, *hpx::for_each*, *async* and *dataflow* used for an Airfoil application. It illustrates a better performance for *dataflow*, which shows the perfect overlap of the computations with the communications enabled by HPX. Hyperthreading is enabled after 16 threads.

that *dataflow* has a better parallelization efficiency and parallel scalability compared to the other methods, illustrating better overlap of the communication with the computation enabled by HPX.

The results show that we are able to optimize the application parallelization by dynamically controlling chunk sizes, interleaving different $direct$ and $indirect$ loops together and asynchronous scheduling tasks at runtime. For further improvements, the chunk size of each loop is determined by considering chunk sizes of the previous loops as described in section 2.4.3 and its results are evaluated in the next section.

2.5.2 Controlling Chunk Sizes

In this section, the chunk sizes of different loops are set by considering chunk sizes determined in the previous loops as described in section 2.4.3. Since *dataflow* enables the compiler to interleave different $direct$ and $indirect$ loops together, the execution of each chunk in each loop depends on the execution of the chunks in the previous loops. So using *persistent_auto_chunk_size* makes the execution time of each chunks in these loops to be the same, which decreases the waiting time between chunks in each loops. Figure 2.27 shows the improvement in the performance of *dataflow* method by using *persistent_auto_chunk_size* as an execution policy within the loops. For an instance, with 32 threads, the improvement is obtained by about 40%.

For further parallelization performance improvements, the data prefetching proposed in section 3.4.4.3 is implemented in the *dataflow* method and its results are evaluated in the next section.

2.5.3 Prefetching Data

The proposed prefetching method is applied on the *dataflow* method and its performance is shown in figure 2.28. This method takes advantage of the asynchronous execution while prefetching data within a loop of the next step in to the cache memory in each iteration step. These results illustrate that the parallel performance of *for_each* is improved by an average of 45%, which confirms the successful process of avoiding cache misses with implementing HPX prefetcher iterator. The bandwidth rate comparison of these results are also shown in figure 2.29.

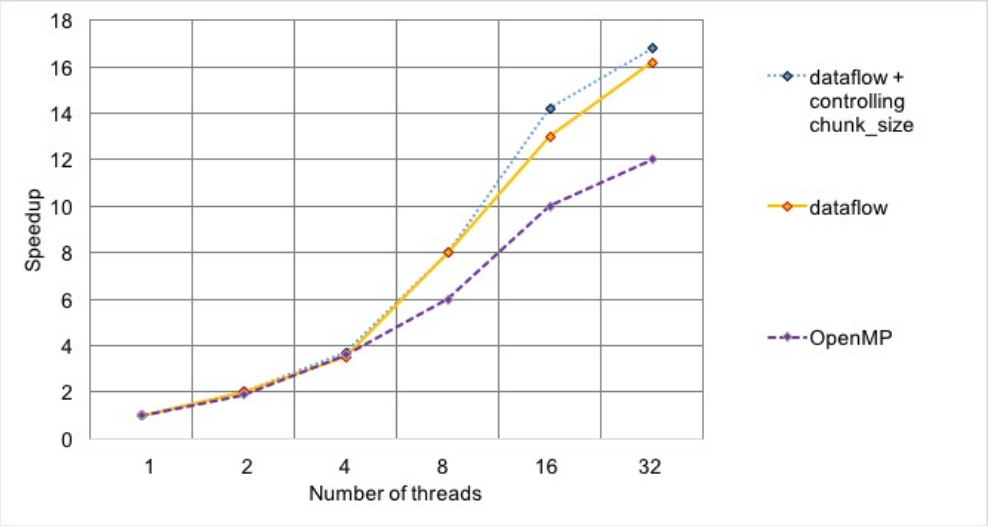


FIGURE 2.27. Comparison results of the strong scaling using *dataflow* with/without setting chunk sizes of different dependent loops based on each other.

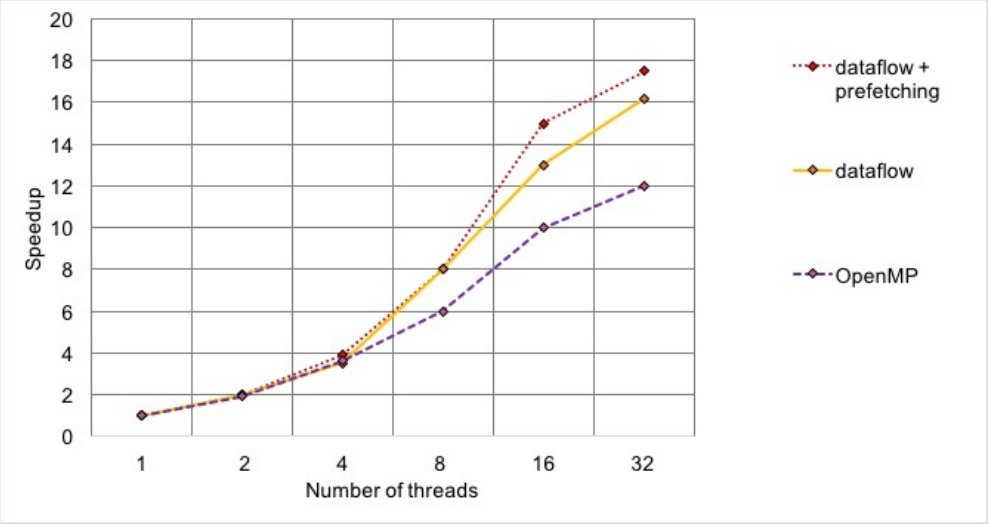


FIGURE 2.28. Comparison results of a *dataflow* performance by using proposed prefetching method. It shows that the speedup is increased by around 45% with prefetching data within a loop.

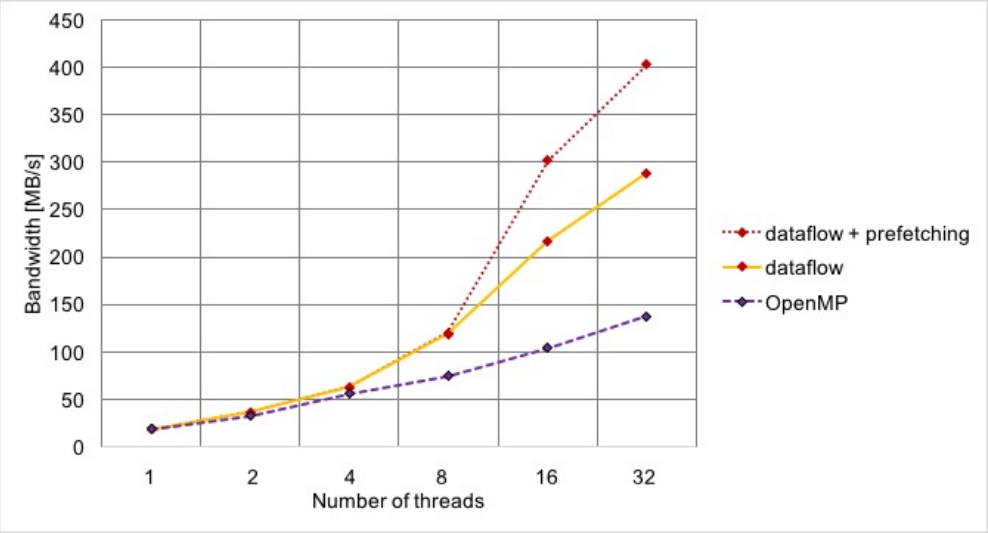


FIGURE 2.29. The data transfer rate of implementing hpx::for_each using standard random access iterator versus prefetching iterator within a *dataflow*.

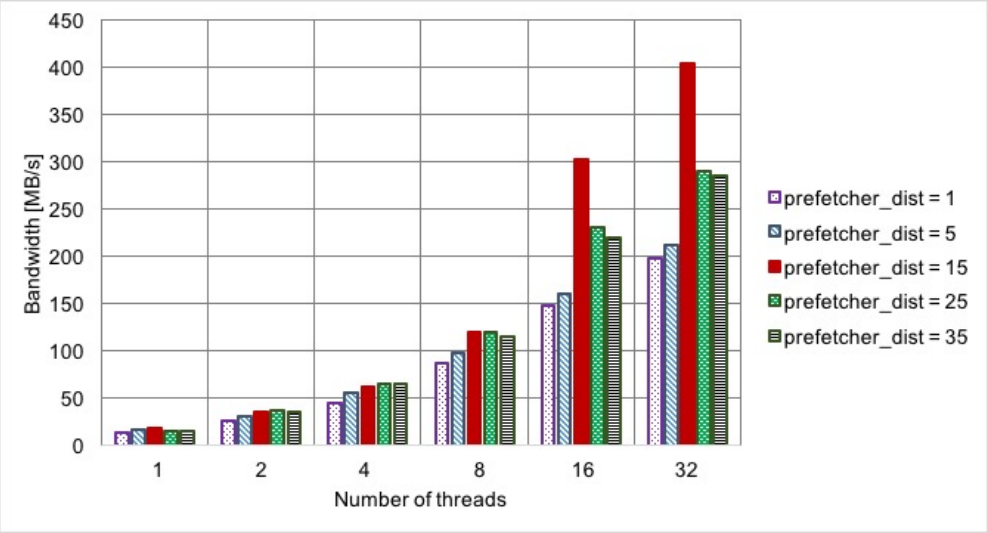


FIGURE 2.30. The data transfer rate of using prefetching iterator for different prefetching distances.

The results of the parallel performance of the prefetching iterator measurements with different *prefetch_distance_factor* are shown in figure 2.30. It can be seen that for the very large distances, data prefetching cannot improve the parallel performance. On the other hand, very small prefetcher distances causes more data to be prefetched, which becomes more expensive. This cost dominates the gains from prefetching and impedes scaling. It is illustrated that $prefetch\_distance\_factor = 15$ for an Airfoil application improves the parallel performance significantly. These results show the good scalability achieved by HPX and indicates that it has the potential to continue to scale on a larger number of threads.

2.6 Conclusion

OP2 provides the framework for the parallel execution of the unstructured grid applications as the abstract for the users to parallelize their application in the high parallelism level without worrying about the architecture specific optimizations. It is designed to achieve a near-optimal scaling on multi-core processors. However its performance is hindered while using conventional techniques. In this research, we present an implementation of the OP2 compiler that employs HPX runtime techniques to efficiently and automatically parallelize unstructured grid applications to achieve desired parallel scalability. The results illustrate that using both dynamic information provided at runtime and the static information provided at compile time are necessary to obtain a higher parallelism level in the applications.

In the proposed framework, OP2 is able to automatically produce data dependencies based on arguments that are passed into the loops at compile time and, by using HPX parallelism methods, the generated loops can be executed asynchronously. In this framework, we propose different optimization methods that make OP2 execute tasks asynchronously, interleave different loops together, efficiently control the chunk sizes of different dependent loops based on each other, and prefetch data into the cache before its actual access.

The performance of the proposed framework is evaluated on an Airfoil simulation, which shows that Airfoil had the same performance using HPX and OpenMP running on 1 thread, but we are

able to obtain 16% scalability improvement in using *async* and 33% scalability improvement in using *dataflow* for the loop parallelization compared with OpenMP. Efficiently controlling chunk sizes and prefetching data improved the overall performance of an Airfoil application by around $40 - 50\%$.

Although these experimental results show an improvement in an application scalability, however the static information are missed in those proposed dynamic techniques. More optimizations could be achieved by utilizing static information extracted at compile time in the runtime system. Since only trusting on the runtime information doesn't guarantee maximal parallel performance, so we believe that collecting information at compile time could significantly improve the HPX parallel applications performances. In the next Chapter, we improve the HPX runtime techniques using static information provided with Clang compiler.

Chapter 3

HPX Intelligent Parallel Algorithms

The performance of many parallel applications depends on loop-level parallelism. However, manually parallelizing all loops may result in degrading parallel performance, as some of them cannot scale desirably to a large number of threads. In addition, the overheads of manually tuning loop parameters might prevent an application from reaching its maximum parallel performance. We illustrate how machine learning techniques can be applied to address these challenges. In this research, we develop a framework that is able to automatically capture the static and dynamic information of a loop. Moreover, we advocate a novel method for determining the execution policy, chunk size, and prefetching distance of an HPX loop to achieve best possible performance by feeding static information captured during compilation and runtime-based dynamic information to our learning model .

3.1 Introduction

Runtime information is often speculative. While runtime adaptive methods have been shown to be very effective – especially for highly dynamic scenarios – solely relying on them doesn't guarantee maximal parallel performance, since the performance of an application depends on both the values measured at runtime and the related transformations performed at compile time. Collecting the outcome of the static analysis performed by the compiler could significantly improve runtime decisions and therefore application performance [9, 25, 26, 48–50].

The goal of this Chapter is to optimize an HPX application's performance by predicting optimum parameters for its parallel algorithms by considering both static and dynamic information and to avoid unnecessary compilation. As all of the HPX parallel algorithms perform based on the dynamic analysis provided at runtime, this technique is unable to achieve the maximum possible parallel efficiency in some cases:

- In [19, 30] different policies for executing HPX parallel algorithms are studied. However these policies should be manually selected for each algorithm within an application, which may not be an optimum approach, as a user should execute each parallel algorithm of his application with different execution policies to find the efficient one for that algorithm.
- Determining chunk size is another challenge in the existing version of the HPX algorithms. Chunk size is the amount of work performed by each task [24, 37] that is determined by an *auto_partitioner* exposed by the HPX algorithms or is passed by using *static/dynamic_chunk_size* as an execution policy's parameter [30]. However,
    1. the experimental results in [50] and [49] showed that the overheads of determining chunk size by using the *auto_partitioner* negatively effected the application's scalability in some cases;
    2. the policy written by the user will often not be able to determine the optimum chunk size either due to the limit of runtime information.
- In [26], we proposed the HPX prefetching method which aids prefetching that not only reduces the memory accesses latency, but also relaxes the global barrier. Although it results in better parallel performance for the HPX algorithms, however, a distance between each two prefetching operations should also be manually chosen by a user for each new program.

Automating these mentioned parameters selections by considering loops characteristics implemented in a learning model can optimize the HPX parallel applications performances. To the best of our knowledge, we present the first attempt to implement a learning model for predicting optimum loop parameters at runtime, wherein the learning model captures features both from static compile time information and from runtime introspection.

In this research, we introduce a new ClangTool *ForEachCallHandler* using LibTooling [51] as a custom compiler pass to be executed by the Clang compiler, which is intended to collect the static features at compile time. The logistic regression model is implemented in this paper as a

learning model that considers these captured features for predicting efficient parameters for an HPX loop. For implementing this learning model on a loop, we propose new execution policies and parameters that – when used on a parallel loop – instructs the compiler to apply this *ForEachCallHandler* tool on that loop. As a results, the loop's features will automatically be included in the prediction process implemented with that learning model. One of the advantages of this approach in utilizing HPX policies is that in practice it enables us to change the algorithms internal structure at runtime and therefore we do not have to compile the code again after the code transformation step.

This technique is able to use high-level programming abstractions and machine learning to relieve the programmer of difficult and tedious decisions that can significantly affect program behavior and performance. Our results show that combining machine learning, compiler optimizations and runtime adaptation helps us to maximally utilize available resources. This improves application performance by around $12\% - 35\%$ for the Matrix Multiplication, Stream, $2D$ Stencil and NBody applications compared to setting their HPX loop's execution policy/parameters manually or using HPX auto-parallelization techniques.

The remainder of this chapter is structured as follows: related works are studied briefly in 3.2; the machine learning algorithms that are used to study the learning models are discussed in section 3.3; the proposed model is discussed is more details in section 3.4, and section 3.5 provides the experimental results of this proposed technique. Conclusion and future works are explained in section 3.6.

3.2 Literature Review

There are several existing publications on automatically choosing optimizations parameters based on static information extracted at compile time. For example in [52, 53], a new runtime technique is proposed that determines an optimum parallel execution schedule for a loop, which requires using synchronization after each iteration for ensuring correct order execution. In these studies, the new *inspector* and *scheduler* are presented for performing preprocessing of the loop's access

pattern and scheduling the loop iterations without using synchronization. Their implementation are based on analyzing data dependencies captured at compile time that are utilized for breaking iterations into smaller subsets that can be executed in parallel. Their evaluated results show that this proposed method is capable of achieving speedups. As another example, in [48, 54, 55] the parallel performance was improved by considering static information captured at compile time and iteratively applying the compilation with different settings for choosing the best sequence of the compiler options. Their implementation resulted in proposing a new compiler framework that is able to adopt its behavior to the user's application, target machine and available transformations. This method optimizes a compiler in choosing an optimum set of transformations and their usage order.

However, one of the challenges in these studies is the need to repeat their proposed methods for each new program, which in general is not desirable, as it requires extra execution time for each application for such parameters determination. Moreover, manually tuning parameters becomes ineffective and almost impossible when too many features are given to the program. Hence, many researches have extensively studied machine learning algorithms which optimize such parameters automatically. This section provides a summary of a number of related works that use machine learning based techniques for improving application performance based of the compile time captured information.

3.2.1 Joshi et al. (2006)

In Joshi et al. [12] clustering algorithm is implemented for examining different benchmarks for their similarities based on their microarchitecture independent characteristics of interest. Their method resulted in finding a representative subset of the applications that can be used for estimating the behavior of the other similar benchmarks with saving simulation time and without loosing accuracy. Clustering algorithm is implemented in their proposed technique on a feature set included instruction mix and control flow behavior statistics. Their method can be easily used during designing benchmark to make it suitable for a group of the candidate programs.

3.2.2 Calder et al. (1997)

Branch prediction is important for the compilers and computer architectures. It is used for predicting whether a branch will be taken before it is actually executed. In Calder et al. [8] neural network and decision tree are applied to predict the branch behavior in a new program. This technique is implemented based on the static branch behaviors captured at compile time.

3.2.3 Stephenson et al. (2005), Stephenson et al. (2009)

Loop unrolling is one of the well known transformation that replicates a loop body a number of times which reduces the overheads of some of the branch operations as well as allowing further compiler optimizations to be applied on a loop. In Stephenson et al. [9] and [11] nearest neighbors and support vector machines are used for predicting unroll factors for different nested loops based on the extracted static features. This feature set includes 38 features determined and extracted at compile time. Then the Mutual Information Scoring method is used to select the smaller subset of this set. In nearest neighbors method, training data are classified into n different classes in which they reduce the overall calculated Euclidian distance. However, their experimental results showed an increasing in complexity with increasing in the number of features. On the other hand, Support Vector Machine showed fastest process. Their evaluated results on different benchmarks showed that their learned classifier is able to predict the loop unrolling factors with good precision.

3.2.4 Agakov et al. (2006)

Generating predictive models applied on the static information is used in Agakov et al. [56] for iterative compiler optimizations. The technique proposed in this study is able to speedup iterative optimization by reducing the number of evaluations of the application performed by a compiler.

3.2.5 Dubach et al. (2007)

Authors in Dubach et al. [57] studied about executing applications on a specific processor and predicting execution time of a new application without actually executing it on real hardware.

Their proposed techniques uses a set of features of an application in Neural Network for predicting application performance.

3.2.6 Cavazos et al. (2006), Pekhimenko et al. (2011)

In Cavazos et al. and Pekhimenko et al. [10, 58], the logistic regression model is used to derive a learning model, which resulted in a significant speedup in compilation time of their studied benchmarks. In [58] authors proposed a new machine learning technique that makes a compiler to decide which optimizations passes to apply on an application and to decide which set of the transformations is needed for that application. This selection is based on the application's reactions that provides the information of the application performance comparisons by applying different transformations. Their studies resulted in developing a framework suitable for a heavily tuned commercial optimizer. In [10], a new method-specific technique is proposed that automatically selects an optimum set of optimizations for different parts of an application. This technique also focuses on reducing the complexity of the process of determining set of the compiler transformations.

3.2.7 Akihiro et al. (2015)

In Akihiro et al. [59], the selection between CPUs and GPUs as an application's hardware device is implemented using Support Vector Machine. The datasets and application's features effect its parallelization performance, so a proper system should be chosen for achieving desired speedup. Their studies resulted in developing a compiler that is able to detect a parallel stream API and records its features such as the parallel loop range, the number of its instructions and etc. Then these features are fed to a learning model that enhances the scalability of an application. Their proposed techniques is developed for the application written with Java and it is trained over 291 samples with 28 features generated by their compiler.

3.2.8 Pellegrini et al. (2009)

Customizing MPI environment to be suitable for the specific applications or hardwares allows improving distributed parallel performance efficiently. In Pellegrini et al. [60], estimating optimum runtime parameters setting for MPI applications on any hardware architecture is implemented using decision trees and artificial neural network. In their proposed method, a new inspector is introduced that is able to select an efficient subset of runtime parameters for any MPI input application by considering static information captured at compile time, number of cores, amount of private/shared cache, and etc.

3.2.9 Planned Contribution

Most of these optimization techniques require users to compile their application twice, first compilation for extracting static information and the second one for recompiling application based on those extracted data. None of these considers both static and dynamic information. The goal of this chapter is to optimize an application's performance by predicting optimum parameters for its parallel algorithms by considering both static and dynamic information and to avoid unnecessary compilation.

3.3 Learning Algorithm

In this research we use the binary and multinomial logistic regression models [27] to select the optimum execution policy, chunk size, and prefetching distance for certain HPX loops based on both, static and dynamic information, with the goal of minimizing execution time. Logistic regression model has been used in several previous works [58, 61], and it is shown to be able to predict such parameters accurately. We will show later that the performance of these learning models has high accuracy for about 98% and 95% for the binary and multinomial logistic regression models respectively on the studied problems. Also, compared to the other learning models such as artificial neural networks (ANNs), the implemented logistic regression model has lower computational complexity. Moreover, since the chunk size values can be seen as a categorical variables,

this makes the logistic regression models well-suited for our problem. We briefly discuss these learning models in the following sections.

3.3.1 Binary Logistic Regression Model

In order to select the optimum execution policy (sequential or parallel) for a loop, we implemented a binary logistic regression model [27] in HPX which analyzes the static information extracted from the loop by the compiler and the dynamic information as provided by the runtime. In this model, the weights parameters having k features $W^T = [\omega_1, \omega_2, ..., \omega_k]$ are determined by considering features values $x_r(i)$ of each experiment $X_i = [1, x_1(i), ..., x_k(i)]^T$ which minimize the log-likelihood of the Bernoulli distribution value as follow:

$$\mu_i = 1/(1 + e^{-W^T X_i}). \tag{3.1}$$

The values of ω are updated in each step t as follows:

$$\omega_{t+1} = (X^T S_t X)^{-1} X^T (S_t X \omega_t + y - \mu_t) \tag{3.2}$$

, where S is a diagonal matrix with $S(i,i) = \mu_i(1 - \mu_i)$. The output is determined by considering the following decision rule:

$$y(x) = 1 \longleftrightarrow p(y = 1|x) > 0.5 \tag{3.3}$$

3.3.2 Multinomial Logistic Regression Model

In order to predict the optimum values for the chunk size and the prefetching distance, we implemented a multinomial logistic regression model [27] in HPX which also analyzes the static information extracted from the loop by the compiler and the dynamic information as provided by the runtime. If we have N experiments that are classified in C classes and each has K features, the

posterior probabilities are computed by using a softmax transformation of the feature variables linear functions for an experiment n with a class c as follow:

$$y_{nc} = y_c(X_n) = \frac{exp(W_c^T X_n)}{\sum_{i=1}^{C} exp(W_i^T X_n)} \tag{3.4}$$

The cross entropy error function is defined as follows:

$$E(\omega_1, \omega_2, ..., \omega_C) = -\sum_{n=1}^{N}\sum_{c=1}^{C} t_{nc} ln y_{nc} \tag{3.5}$$

, where T is a $N \times C$ matrix of target variables with t_{nc} elements. The gradient of E is computed as follows:

$$\nabla_{\omega_c} E(\omega_1, \omega_2, ..., \omega_C) = \sum_{n=1}^{N} (y_{nc} - t_{nc}) X_n \tag{3.6}$$

In this method, we use the Newton-Raphson method [62] to update the weights values in each step:

$$\omega_{new} = \omega_{old} - H^{-1} \nabla E(\omega) \tag{3.7}$$

where H is the Hessian matrix defined as follows:

$$\nabla_{\omega_i} \nabla_{\omega_j} E(\omega_1, \omega_2, ..., \omega_C) = \sum_{n=1}^{N} y_{ni}(I_{ij} - y_{nj}) X_n X_n^T \tag{3.8}$$

Since the Hessian matrix for this regression model is positive definite, its error function has a unique minimum. At the end of this iterative process, a set of weights is determined for a learning model that gives the best classification on a given set of training data. More details can be found in [63].

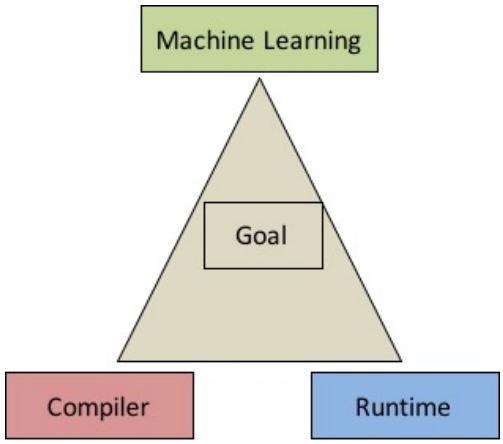


FIGURE 3.1. The goal of this research is to combine machine learning methods, compiler transformations, and runtime introspection in order to maximize the use of available resources and to minimize execution time of the loops.

3.4 Proposed Model

In this section, we propose a new technique for applying the learning models discussed in section 3.3 to HPX loops. The goal of this technique is to combine machine learning methods, compiler transformations, and runtime introspection in order to maximize the use of available resources and to minimize execution time of the loops (see Fig. 3.1). Its design and implementation has several steps categorized as follow[1]:

A. Special Execution Policies and Parameters

B. Feature Extraction

C. Designing Learning Model

D. Learning Model Implementation

Fig. 3.2 shows the scheme of the workflow of this model, in which the static information about the loop body (such as the number of operations, see Table 3.1) collected by the compiler and the dynamic information (such as the number of cores used to execute the loop) as provided by the

[1]This technique with its installation instructions are publicly available at `https://github.com/STEllAR-GROUP/hpxML`. Feel free to join our IRC channel $\#ste||ar$ if you need any help.

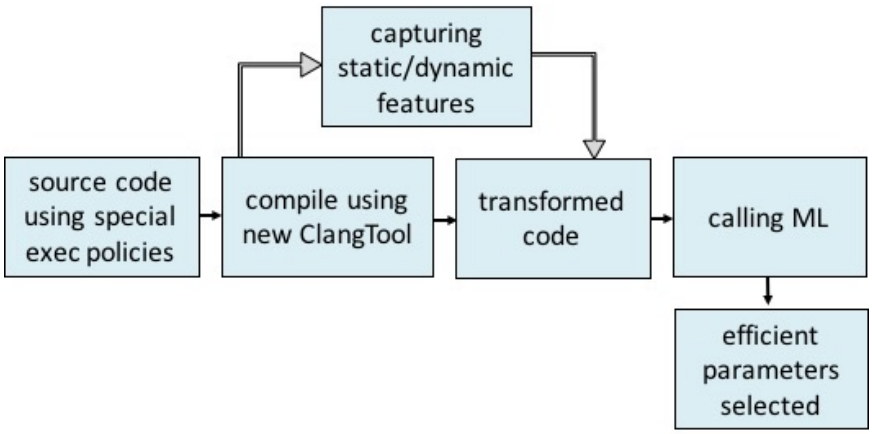


FIGURE 3.2. The scheme of the workflow of the proposed model.

runtime system is used to feed a logistic regression model enabling a runtime decision to obtain highest possible performance of the loop under consideration. The presented method relies on a compiler-based source-to-source transformation. The compiler transforms certain loops which were annotated by the user by providing special execution policies – discussed later in section 3.4.1 – into code controlling runtime behavior. This transformed code instructs the runtime system to apply a logistic regression model and to select either an appropriate code path (e.g. parallel or sequential loop execution) or certain parameters for the loop execution itself (e.g. chunk size or prefetching distance).

3.4.1 Special Execution Policies and Parameter

We introduce two new HPX execution policies and one new HPX execution policy parameter, which enables the weights gathered by the learning model to be applied on the loop: *par_if* and *make_prefetcher_policy*. These policies instrument executors to be able to consume the weights produced by a binary logistic regression model, which is used to select the execution policy corresponding to the optimal code path to execute (sequential or parallel), and a multinomial logistic regression model, which is used to determine an efficient prefetching distance. Additionally we created an new execution policy parameter, *adaptive_chunk_size*, which uses a multinomial logistic regression model to determine an efficient chunk size. Fig.3.3 shows three loops defined with these new execution policies and the new execution parameter that apply a *lambda* function over

```
//This loop chooses an optimum execution policy(seq/par)
for_each(par_if, range.begin(), range.end(), lambda);

//This loop chooses an efficient chunk size
for_each(policy.with(adaptive_chunk_size())
          ,range.begin(),range.end(),lambda);

//This loop chooses an efficient prefetching distance for
   prefetching array1 and array2
for_each(make_prefetcher_policy(policy,
          prefetching_distance_factor,
          container_1,..., container_n),
          range.begin(),range.end(),lambda);
```

FIGURE 3.3. Loops using the new execution policies and parameters. These policies and parameters are instrumented to allow HPX to consider the weights produced by the learning models when executing the loops.

a *range*. We have created a new special compiler pass for clang which recognizes these annotated loops and transform them into equivalent code which instructs the runtime to apply the described regression models.

3.4.2 Feature Extraction

Initially, we selected 10 static features to be collected at compile time and 2 dynamic features to be determined at runtime to be evaluated by our learning model. These features are listed in Table 3.1. Although it may not be the best possible set, it is very similar to those considered in the other works [9, 48, 58], which in their results indicated that the set is sufficient to design a learning model for this type of problem. In order to collect static information at compile time, we introduce a new ClangTool named *LoopConvert* in the Clang compiler as shown in Fig.3.4. This tool locates in the user source code instances of loops which use *par_if* or *make_prefetcher_policy* as their execution policy or *adaptive_chunk_size* as an execution policy parameter. Once identified, the loop body is then extracted from the $lambda$ function by applying *getBody()* on a $lambda$ operator *getLambdaCallOperator()*. The value of each of the listed static features is then recorded by passing $lambda$ to *analyze_statement*. In order to capture dynamic features at runtime, the compiler

TABLE 3.1. Collected static and dynamic features. First 6 features marked with red* have been selected for our model using the decision tree classification technique [64, 65] to avoid overfitting the model.

Type	Information
dynamic	number of threads∗
dynamic	number of iterations∗
static	number of total operations per iteration∗
static	number of float operations per iteration∗
static	number of comparison operations per iteration∗
static	deepest loop level∗
static	number of integer variables
static	number of float variables
static	number of if statements
static	number of if statements within inner loops
static	number of function calls
static	number of function calls within inner loops

inserts hooks (HPX API function calls) which are invoked by the runtime. In this instance the compiler will insert the call *hpx::get_os_thread_count()* and *std::distance(range.begin(), range.end())* which will return the number of OS threads as well as the number of iterations that the loop will run over, respectively.

To avoid overfitting the model, we chose 6 critical features marked with red∗ color in Table 3.1 to include in the actual decision tree classification technique[64, 65]. This algorithm starts with considering entire features set, then it selects the best feature that yields maximum information for a better classification. It reduces the initial feature set in a tree building process based on the information gain value to decide which feature to be selected for splitting data at each step in a tree building process. This information gain is computed for each feature in each step, and the one with the highest value will be chosen as the test feature for that set of training data. This value for each feature A with v distinct values and having m distinct classes is computed as follow:

$$Information_Gain(A) = I(s_1, s_2, ..., s_m) - E(A) \tag{3.9}$$

, where $E(A)$ is an entropy that is given by

```
class LoopConvert:
     public RecursiveASTVisitor<LoopConvert>
{
  ...
  // Visit every call expression
  bool VisitCallExpr(const CallExpr *call)
  {
    ...
    SourceManager &SM = m_context->getSourceManager();

    //check if a call is an hpx algorithm
    const clang::FunctionDecl* FD=call->getDirectCallee();
    string func_string=FD->getQualifiedNameAsString();
    if (func_string.find("hpx::parallel::") !=string::npos)
    {

      //Capturing lambda function from a loop
      const CXXMethodDecl* lambda_callop =
             lambda_record->getLambdaCallOperator();
      Stmt* lambda_body = lambda_callop->getBody();

      //Capturing policy
      SourceRange policy(call->getArg(0)->getExprLoc(),
       call->getArg(1)->getExprLoc().getLocWithOffset(-2));

      string policy_string = Lexer::getSourceText(
          CharSourceRange::getCharRange(policy), *SM,
          LangOptions()).str();

      //Determining policy if a current policy is par_if
      if (policy_string.find("par_if") != string::npos)
      {
        //Extracting static information from lambda function
         analyze_statement(lambda_body);
         policy_determination(call, SM);
      }

      //Determining chunk size if a current policy's parameter is adaptive_chunk_size
      if(policy_string.find("adaptive_chunk_size")!=string::npos)
      {
        //Extracting static information from lambda function
         analyze_statement(lambda_body);
         chunk_size_determination(call, SM);
      }

      //Determining prefetching distance if a current policy is make_prefetcher_policy
      if(policy_string.find("make_prefetcher_policy")!=string::npos)
      {
        //Extracting static information from lambda function
         analyze_statement(lambda_body);
         prefetching_distance_determination(call, SM);
      }
    }
}}
```

FIGURE 3.4. In the Clang compiler, we propose to add the ClangTool *LoopConvert* which will collect static information of each for loop and to implement a learning model based on the current execution policy:

- *par_if.*
- *policy.with(adaptive_chunk_size).*
- *make_prefetcher_policy(policy, ...).*

$$E(A) = \sum_{j=1}^{v} \frac{s_{1j} + s_{2j} + ... + s_{mj}}{s} I(s_1, s_2, ..., s_m) \tag{3.10}$$

, that I is computed as follow:

$$I(s_1, s_2, ..., s_m) = -\sum_{k=1}^{m} p_k log_2(p_k) \tag{3.11}$$

, in which p_k is the probability that an arbitrary data belongs to class k and is given by

$$p_k = \frac{s_k}{s} \tag{3.12}$$

This classifier generation stops whenever all the data of a leaf belong to the same class. More information about this technique can be found in [64, 66, 67].

3.4.3 Designing Learning Model

To design an efficient learning model that could be able to cover various cases, we collected over 300 training data sets by analyzing Matrix multiplication application with different problem sizes that implements *par_if*, *adaptive_chunk_size* or *make_prefetcher_policy* on its loops. The experimental results evaluated in Section 3.5 show that these training data[1] are enough to predict the HPX loop's parameters accurately for the studied applications: Matrix multiplication, Stream, $2D$ Stencil and NBody applications.

Both the binary and multinomial logistic regression models are implemented in C++[1]. These models are designed based on the collected data, in which the values of ω from eq.3.14 and eq.3.7 are determined whenever the sum of square errors reaches its minimum value. Then they are stored in an output file named as *weights.dat* that will be used for predicting the optimal

[1]The characteristics of the loops of these training data are available at `https://github.com/STEllAR-GROUP/hpxML/blob/master/logisticRegressionModel/algorithms/inputs`.

execution policy, chunk size, and prefetching distance at runtime. This learning step can be done offline, which also doesn't add any overhead at compile time nor does at runtime.

It should be noted that the multinomial logistic regression model must be initialized with the allowed boundaries for the chunk size and prefetching distance in order to choose an efficient value. In this study we selected 0.1%, 1%, 10%, or 50% of the iterations of a loop as chunk size candidates and 1, 5, 10, 100 and 500 cache lines as prefetching distance candidates. These candidates are validated with different tests and based on their results, they are selected. Also, they are already included in the implementation of the proposed technique discussed in the next section and it is not required for the users to include them manually.

3.4.4 Learning Model Implementation

In this study, we propose three new techniques that are able to implement binary and multinomial logistic regression models at runtime:

1. Implementing Binary Logistic Regression Model (Execution Policy).
2. Implementing Multinomial Logistic Regression Model (Chunk Size).
3. Implementing Multinomial Logistic Regression Model (Prefetching Distance).

These new techniques are discussed as follow.

3.4.4.1 Implementing Binary Logistic Regression Model (Execution Policy)

For this purpose, we propose a new function *seq_par* that passes the extracted features for a loop that uses *par_if* as its execution policy. In this technique, a Clang compiler automatically adds extra lines within a user's code as shown in Fig.3.5b that allows the runtime system to decide whether execute a loop sequentially or in parallel based on the return value of *seq_par* from Eq.3.15. If the output is $false$ the loop will execute sequentially and if the output is $true$ the loop will execute in parallel. This function takes the weights extracted during compilation

```
for_each(par_if,range.begin(),range.end(),lambda);
```

(a) Before compilation.

```
if(seq_par({extracted_static/dynamic_features}))
  for_each(seq, range.begin(),range.end(),lambda);
else
  for_each(par, range.begin(),range.end(),lambda);
```

(b) After compilation.

```
bool seq_par(F &&features)
{
  ...
  // from external file
   W wights = retrieving_weights("weights.dat");
  return policy_costs_fnc(features, wights);
}
```

(c) Determining execution policy at runtime using features and weights values.

FIGURE 3.5. The proposed *seq_par* for implementing binary logistic regression model at runtime.

and the values polled at runtime as inputs. Fig.3.5c shows the *seq_par* function implemented for determining policy by computing cost function with features and weights values as its inputs.

3.4.4.2 Implementing Multinomial Logistic Regression Model (Chunk Size)

For this purpose, we propose a new function *chunk_size_determination* that passes the extracted features for a loop that uses *adaptive_chunk_size* as its execution policy's parameter. In this technique, a Clang compiler changes a user's code automatically as shown in Fig.3.6b that makes runtime system to choose an optimum chunk size based on the output of *chunk_size_determination* from Eq.3.16 that is based on the chunk size candidate's probability and it is computed using the values of the studied loop's features and the learning model's weights. Fig.3.6c shows *chunk_size_determination* function implemented for determining chunk size by computing cost function with features and weights values as its inputs.

```
for_each(policy.with(adaptive_chunk_size()),
         range.begin(),range.end(),lambda);
```

(a) Before compilation.

```
for_each(policy.with(chunk_size_determination({extracted_static/
    dynamic_features}))),
         range.begin(),range.end(),lambda);
```

(b) After compilation.

```
dynamic_chunk_size chunk_size_determination(F &&features)
{
  ...
  // from external file
   W wights = retrieving_weights("weights.dat");
  return chunk_size_costs_fnc(features, wights);
}
```

(c) Determining chunk size at runtime using features and weights values.

FIGURE 3.6. The proposed *chunk_size_determination* for implementing multinomial logistic regression model at runtime.

3.4.4.3 Implementing Multinomial Logistic Regression Model (Prefetching Distance)

For this purpose, we propose a new function *prefetching_distance_determination* that passes the extracted features for a loop that uses *make_prefetcher_policy* as its execution policy. In this technique, a Clang compiler changes a user's code automatically as shown in Fig.3.7b that makes runtime system to choose an optimum prefetching distance based on the output of *prefetching_distance_determination*. This function also computes the outputs by implementing Eq.3.16 using the values of the studied loop's features and the learning model's weights. Fig.3.7c shows *prefetching_distance_determination* function implemented for determining prefetching distance by computing cost function with features and weights values as its inputs.

As we can see, these proposed techniques consider both, the static and the dynamic information for determining an efficient execution policy, chunk size, and prefetching distance for a loop. In addition, this decision process is performed at runtime by computing outputs of *seq_par*, *chunk_size_determination* and *prefetching_distance_determination*, which avoids an extra compilation step. In other words, static information is collected during compilation and the decisions

```
for_each(make_prefetcher_policy(policy,
            prefetching_distance_factor,
            container_1,..., container_n),
            range.begin(),range.end(),lambda);
```

(a) Before compilation.

```
for_each(make_prefetcher_policy(policy,
            prefetching_distance_determination({extracted_static/
                dynamic_features}),
            container_1,..., container_n),
            range.begin(),range.end(),lambda);
```

(b) After compilation.

```
std::size_t prefetching_distance_determination(F &&features)
{
  ...
  // from external file
   W wights = retrieving_weights("weights.dat");
  return prefetching_distance_costs_fnc(features, wights);
}
```

(c) Determining prefetching distance at runtime using features and weights values.

FIGURE 3.7. The proposed *prefetching_distance_determination* for implementing multinomial logistic regression model at runtime.

aiming at optimum parameters are made at runtime while taking into account additional runtime information. One of the other advantages of this method are that other parameters and executors attached to the current execution policy can be also reattached to the generated execution policy as shown in Fig.3.8. Moreover, all of all of these new execution policies and parameters can be used together by simply defining a loop policy to be "*make_prefetcher_policy(par_if, ...).with(adaptive_chunk_size())*". The experimental results of our proposed learning techniques discussed are presented in the next section.

3.5 Experimental Results

In this section, we evaluate the performance of our proposed technique using Clang $4.0.0$ and HPX $V0.9.99$ on the test machine with two Intel Xeon E5-2630 processors, each with 8 cores clocked at $2.4GHZ$ and $65GB$ of main memory. The main goal here is to illustrate that dynamic information obtained at runtime and static information obtained at compile time are both necessary to provide sufficient parallel performance and the proposed techniques are able to predict the optimum parameters for HPX loops based on these information. We illustrate the performance of our proposed model in the following three sections[1]:

1. Evaluating Model Accuracy
2. Artificial Test Cases
3. Real Benchmarks

3.5.1 Evaluating Model Accuracy

As discussed in Section 3.3, binary logistic regression model and multinomial logistic regression model are implemented for determining execution policy, chunk size and prefetching distance of a loop. As mentioned in 3.4.4, chunk size candidates are 0.1%, 1%, 10%, or 50% of the iterations of a loop and prefetching candidates are 1, 5, 10, 100 and 500 cache lines.

[1]Applications evaluated in this Section are publicly available at `https://github.com/STEllAR-GROUP/hpxML/tree/master/examples`.

```
for_each(par_if.with(parameter).on(executor),
         range.begin(),range.end(),lambda);


for_each(policy.on(executor).with(adaptive_chunk_size()),
         range.begin(),range.end(),lambda);


for_each(make_prefetcher_policy(
         policy.with(parameter).on(executor),
         prefetching_distance_factor,
         container_1,..., container_n),
         range.begin(),range.end(),lambda);
```

(a) Before compilation.

```
if(seq_par({extracted_static/dynamic_features}))
  for_each(seq.with(parameter).on(executor),
           range.begin(),range.end(),lambda);
else
  for_each(par.with(parameter).on(executor),
           range.begin(),range.end(),lambda);


for_each(policy.on(executor).with(chunk_size_determination({
   extracted_static/dynamic_features}))),
         range.begin(),range.end(),lambda);


for_each(make_prefetcher_policy(
         policy.with(parameter).on(executor),
         prefetching_distance_determination({extracted_static/
            dynamic_features}),
         container_1,..., container_n),
         range.begin(),range.end(),lambda);
```

(b) After compilation.

FIGURE 3.8. Reattaching policy's parameters and executors to the final determined execution policy.

TABLE 3.2. Execution policy, chunk size and prefetching distance determined by *seq_par*, *chunk_size_determination* and *perfecting_distance_determination* implementation based on the static and dynamic information extracted for each loop and the weights provided by the learning models.

Test	Loop	Iterations	Total opr.	Float opr.	Comparison opr.	Loop level	Policy (Threads)	Chunk size%	Pref. dist.
1	l_1	10000	400100	200000	101010	2	par (8)	0.1	5
	l_2	20000	450026	250000	150503	2	par (8)	0.1	5
	l_3	20000	502040	250000	103051	2	par (8)	0.1	1
	l_4	500	550402	200000	150102	1	par (8)	10	5
2	l_1	150000	350106	101010	500	2	par (8)	0.1	10
	l_2	100	10050016	5000000	2505013	3	seq	10	1
	l_3	100	25000000	3010204	1500204	3	seq	10	1
	l_4	50000	4000450	200000	100150	1	par (8)	1	5
3	l_1	500	4504030	250000	150300	2	par (8)	1	10
	l_2	400	3502020	200000	100405	1	par (8)	1	10
	l_3	2000	250033	150000	103040	3	seq	10	5
	l_4	2500	350400	150000	100600	3	seq	10	5
4	l_1	20000	204002	100000	10320	2	par (8)	0.1	1
	l_2	30000	400000	150102	10000	2	par (8)	0.1	1
	l_3	300	550000	44000	20030	3	seq	10	5
	l_4	400	450000	50400	10602	3	seq	10	10
5	l_1	200	4502001	150000	101004	3	par (8)	1	1
	l_2	700	400020	300000	150006	3	par (8)	1	5
	l_3	300	302020	20000	14005	2	par (8)	1	5
	l_4	100	50400	20000	10110	2	seq	10	10

The characteristics of these models are derived from a training set of over 300 test cases. In order to derive the fidelity of the model, we train the algorithm using 80% of the test cases and use the remaining 20% as a trials to see how accurate the predictions are. Our results show that the binary logistic regression model is accurate in 98% of the trials and the multinomial logistic regression model is accurate in 95% of them.

3.5.2 Evaluating Proposed Techniques on Different Test Cases

In this section, we evaluate the performance of the proposed techniques from section 3.4 over 5 test cases shown in Table 3.2, in which each of them includes 4 loops with different characteristics. Each of these loops of each test cases is a Matrix multiplication computation with different problem sizes included in this table. The main purpose of these evaluations is to show the effectiveness of each proposed method on an HPX parallel performance.

3.5.2.1 *par_if*

Parallelizing all loops within an application may not result in a best possible parallelization, as some of the loops cannot scale desirably on more number of threads. For evaluating the effec-

tiveness of the proposed *seq_par* function exposed by a new execution policy *par_if* discussed in section 3.4, we study its implementation on the described 5 test cases. These test cases are selected to show that in case of having several loops within a parallel application, some of these loops should be executed in sequential to achieve a better parallel performance. Each of these test cases is executed three times by setting execution policies of the outer loops to be *seq*, *par*, or *par_if* in each time. The static and dynamic characteristics of each loop in each test are listed in Table 3.2. The execution policies determined by using *par_if* policy for each loop are also included in the column *Policy* of this Table.

Fig.3.9 shows the execution time for each test case and it illustrates that in most of them using the execution policy *par_if* will outperform the basic policy *par*. The main reason of this improvement is that by considering the determined execution policy included in Table 3.2, as the execution policy *seq* is determined for some of the loops that cannot scale desirably on more number of threads, this technique results in outperforming manually parallelized code by around $15\% - 20\%$ for these test cases expect the first one. In this test case, however, the total execution time of the loops took slightly longer when invoked with *par_if*. This is due to the overhead generated during the invocation of the *policy_costs_fnc* cost function, manually setting their execution policy as *par* resulted in having a better performance.

3.5.2.2 *adaptive_chunk_size*

As discussed in Section 3.4, the proposed *chunk_size_determination* function exposed by a new execution policy's parameter *adaptive_chunk_size* enables the runtime system to choose an efficient chunk size for a loop by considering static and dynamic features of that loop. As mentioned in section 3.4.3, this method selects between chunk sizes of 0.1%, 1%, 10%, or 50% of the iterations of a loop by comparing their probabilities in the multinomial logistic regression model's cost function

Fig.3.10 shows the execution time for each test case in Table 3.2 by setting optimal chunk size of each loop. The chunk size determined by the algorithm for each loop are also included in the

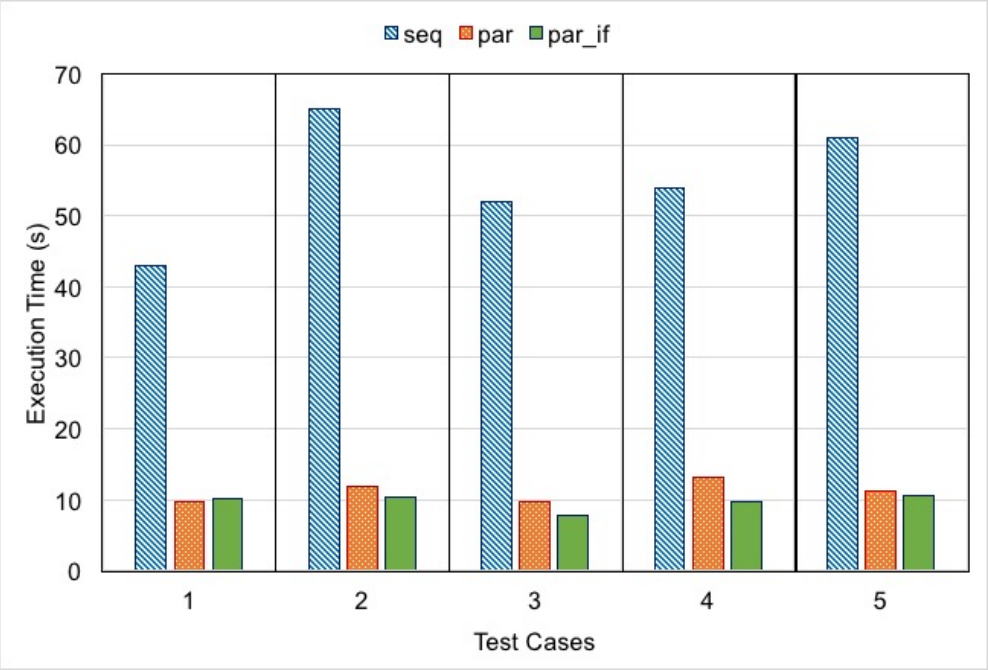


FIGURE 3.9. The execution time comparisons between setting execution policy of the loops to be *seq*, *par*, or *par_if*.

column *Chunk size%* of the Table 3.2. The overall performance of these cases show by an average of about 31%, 15%, 17% and 38% improvement over setting chunks to be 0.1%, 1%, 10%, or 50% of the iterations of a loop. The main reason of this improvement is that efficient chunk size helps in having even amount of work on each number of threads that results in reducing total overheads and latencies. These results also illustrate the importance of the chunk size's effect on an application's scalability and the capability of this method in improving parallel performance of an application by choosing efficient chunk size for each loop.

3.5.2.3 *make_perfetcher_policy*

As discussed in section 3.4, the proposed *perfecting_distance_determination* function exposed by a new execution policy *make_perfetcher_policy* allows the runtime system to choose an efficient prefetching distance for a loop by considering static and dynamic features of that loop. As it mentioned in section 3.4.4, this method chooses between prefetching distances of 1, 5, 10, 100

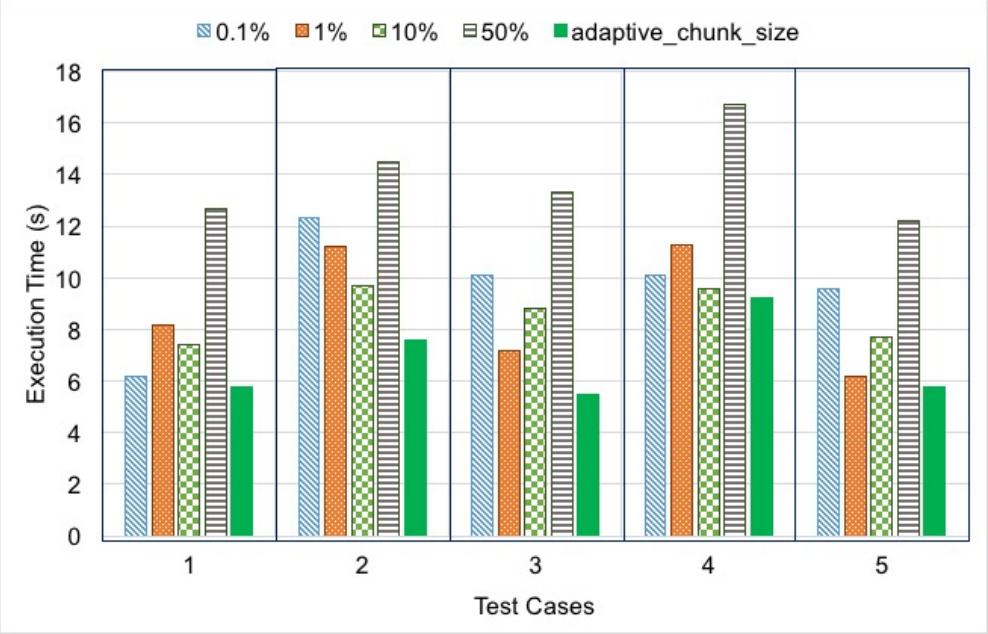


FIGURE 3.10. The execution time comparisons between setting chunk size of the loops to be 0.1%, 1%, 10%, or 50% of the iterations of a loop and the chunk size determined by using *adaptive_chunk_size*.

and 500 cache lines by comparing their probabilities in the *prefetching_distance_costs_fnc* cost function from fig.3.7c.

Fig.3.11 shows the execution time for each prefetching size in each test case in Table 3.2. The prefetching distance determined by the algorithm for each loop are also included in the last column of the Table 3.2. The overall performance of these cases show by an average of about 25%, 19%, 14%, 33%, 24%, and 47% improvement over setting prefetching distances to be 1, 5, 10, 100, or 500 cache lines. The main reason of this improvement is that using efficient prefetching distance resulted in better cache usage that reduced the total overheads.

3.5.3 Real Benchmarks

In the previous section, we demonstrated the effectiveness of each proposed on an HPX parallel performance on 5 different test cases in which each of them includes 4 different loops for a matrix multiplication computation. In this section, we apply all of the proposed methods together on four different benchmarks: the Stream, a $2D$ Stencil, a Matrix Multiplication and an NBody

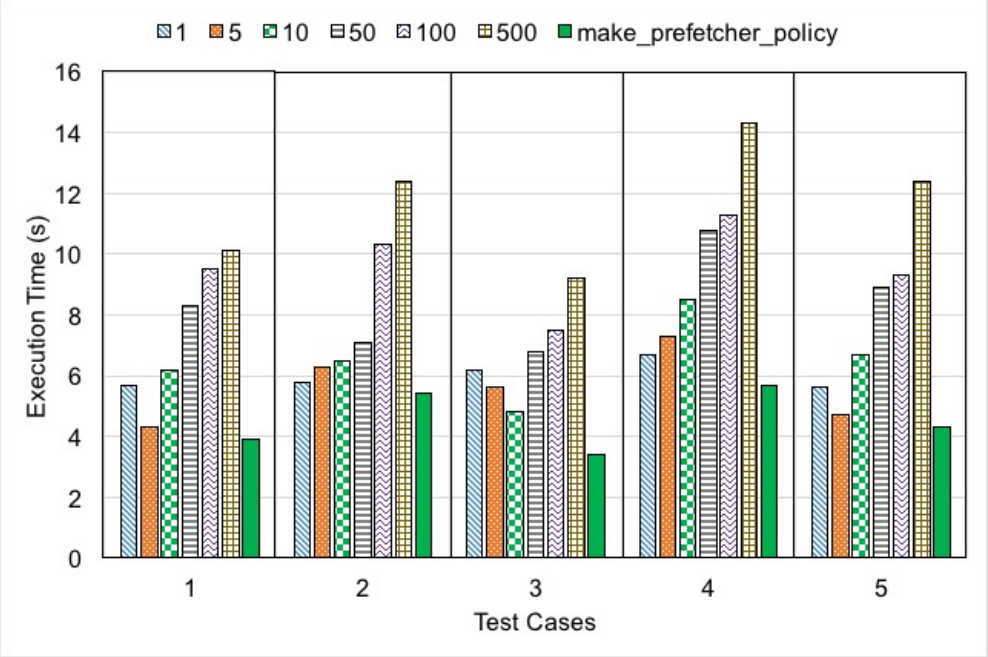


FIGURE 3.11. The execution time comparisons between setting the prefetching distance of the loops to be 1, 5, 10, 100, or 500 cache lines and the determined prefetching distance using *make_perfetcher_policy*.

TABLE 3.3. Dynamic and static features of the loops in Stream, Stencil and Matrix Multiplication Benchmarks.

Test	Iterations	Total opr.	Float opr.	Comp. opr.	Loop level
Stream	50000000	8	8	0	0
Stencil	45	3502	2500	301	1
Mat. Mult.	200	123706	320800	10053	2

benchmarks. The previous training data is also used in the proposed techniques applied on these applications.

3.5.3.1 Stream Benchmark

The Stream benchmark [68, 69] has been widely used for evaluating memory bandwidth of a system. In [30], the HPX executors performance were evaluated on this benchmark with 50 million data points over 10 iterations. As shown in Fig.3.12, this application includes 4 operations over 3 equally sizes arrays (A, B and C) that are: copy ($C = A$), scale ($B = k \times C$), adding ($C = A + B$) and triad ($A = B + K \times C$). The characteristic information of this loop is included in Table 3.3. All

```
for_each(policy, a_begin, a_end,[&](std::size_t i){
                    // copy step
                    c[i] = a[i];

                    // scale step
                    b[i] = k * c[i];

                     // adding step
                    c[i] = a[i] + b[i];

                    // triad step
                    a[i] = b[i] + k * c[i];
                });
```

FIGURE 3.12. Stream Benchmark.

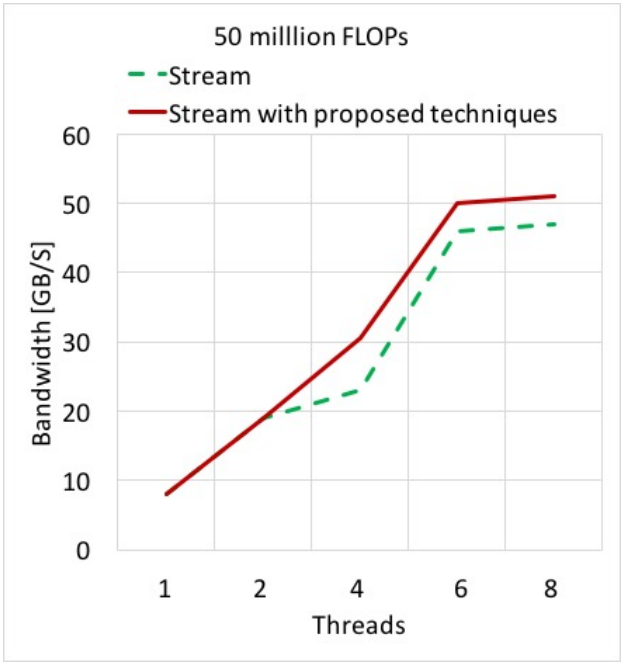


FIGURE 3.13. HPX Stream benchmark's strong scaling using the proposed techniques together compared to implementing without considering static/dynamic information and implementing machine learning technique.

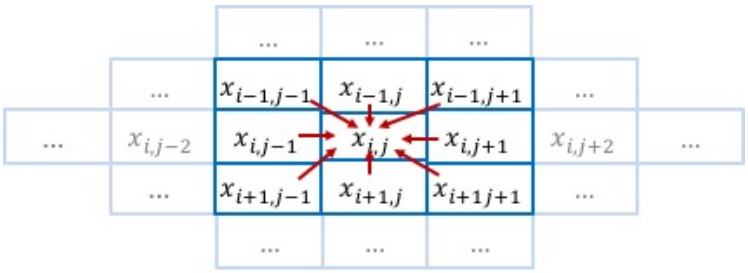


FIGURE 3.14. Heat Distribution Benchmark, HPX Stencil [23].

three proposed techniques – *seq_par*, *adaptive_chunk_size* and *make_prefetcher_policy* – are applied on this loop to make HPX to choose an execution policy, chunk size and prefetching distance efficiently. The speedup comparison results of the data transform measurements with/without using proposed techniques are illustrated in Fig.3.13. As we can see, using the proposed techniques together on this benchmark improves HPX performance by an average of about 13% compared to using HPX auto-parallelization techniques without considering static/dynamic information and implementing machine learning technique.

3.5.3.2 Stencil Benchmark

The performance of different HPX scheduling policies on a $2D$ Stencil benchmark is studied in [19]. This application is a two dimensional heat distribution shown in Fig.3.14, in which the temperature of each point is computed based on the temperature of its neighbors. The characteristic information of this loop is also included in Table 3.3. Same as the Stream Benchmark, all three proposed techniques – *seq_par*, *adaptive_chunk_size* and *make_prefetcher_policy* – are also applied on this loop to make HPX to choose an execution policy, chunk size and prefetching distance efficiently. The speedup comparison results of HPX performance with/without using proposed technique are illustrated in Fig.3.18. It shows HPX performance improvement by an average of about 22% by using the proposed techniques together on this loop compared to using HPX auto-parallelization techniques without considering static/dynamic information and implementing machine learning technique.

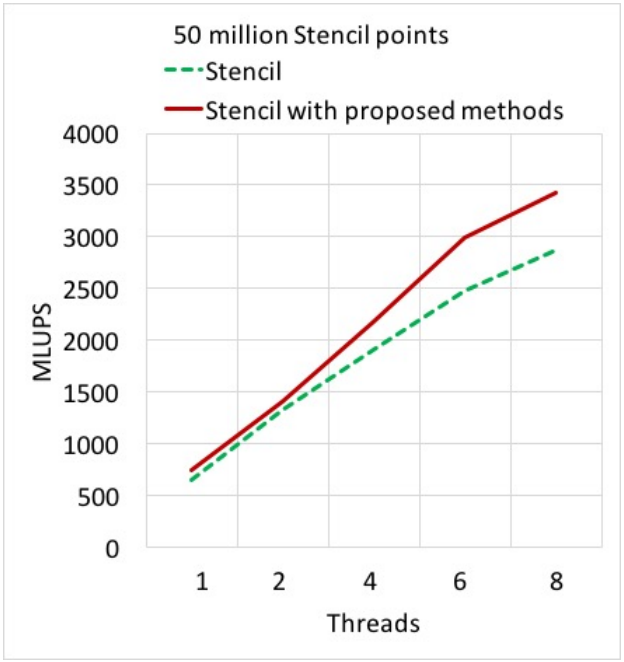


FIGURE 3.15. HPX $2D$ Stencil benchmark's strong scaling using the proposed techniques together compared to using HPX auto-parallelization techniques without considering static/dynamic information and implementing machine learning technique.

3.5.3.3 Matrix Multiplication Benchmark

HPX matrix multiplication performance using the proposed techniques is evaluated in this section. The characteristic information of this loop is also included in Table 3.3. Same as the Stream and Stencil Benchmarks, all three proposed techniques – *seq_par*, *adaptive_chunk_size* and *make_prefetcher_policy* – are also applied on this loop to make HPX to choose an execution policy, chunk size and prefetching distance efficiently. Fig. 3.16 illustrates the speedup comparison results of the benchmark's performance with/without using proposed technique. As we can see, using the proposed techniques together improve the benchmark's parallel performance by an average about 37% compared to using HPX auto-parallelization techniques without considering static/dynamic information and implementing machine learning technique.

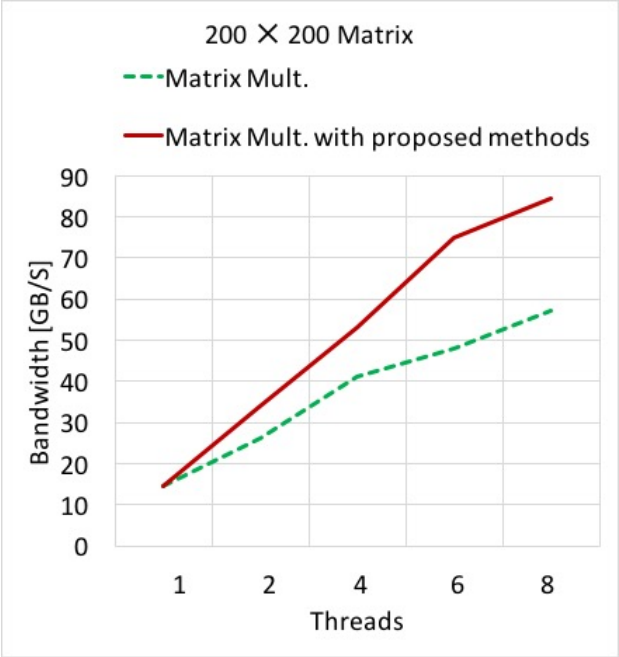


FIGURE 3.16. Matrix Multiplication benchmark's strong scaling using the proposed techniques together compared to using HPX auto-parallelization techniques without considering static/dynamic information and implementing machine learning technique.

3.5.3.4 NBody Benchmark

In an NBody problem [70, 71], updating the information of each of the particles requires accessing the information about the other particles. The brute-force NBody algorithm which computes a new position of a particles by considering the forces felt by all other N particles in each time step has an $O(N^2)$ time complexity. In order to improve the parallel scalability of this application, various algorithms have been proposed, e.g. Barnes-Hut and Fast Multipole Method (FMM) with $O(NlogN)$ and $O(log(1/\epsilon)N)$ time complexity respectively [72–74].

In [26], we showed performance improvement by using features of the HPX parallel algorithms to decrease the communication latencies in a Barnes-Hut algorithm when compared to using a hybrid model (OpenMP + MPI). The experimental results evaluated in that paper showed that HPX helped this application to achieve a desired scalability by providing task-based parallelism and asynchronous task execution. It was illustrated that the *future* concepts implemented by the HPX algorithms allowed the tasks related to a single particle in an NBody application to execute

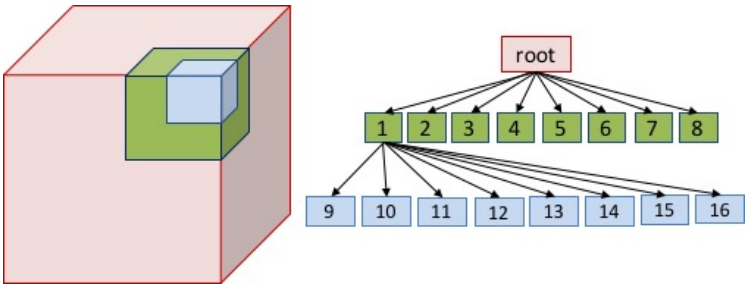


FIGURE 3.17. The Octree data structure used in an NBody application.

when its dependencies were satisfied without having to wait for the resolution of dependencies for all other particles. This effectively circumvents the global barriers that are imposed by the hybrid model implementation where all particles were not able to continue to the next timestep until all particles had reached that timestep. The NBody application used in this research employs the Barnes-Hut algorithm. This algorithm has the following three stages:

1. **Octree Construction**: Fig.3.17 shows the Octree data structure applied in the NBody application [73, 75]. The maximum number of particles within each cube is considered to be $N_{threshold}$. This implies that each cube will be subdivided into eight equally sized sub-cubes if it has more than $N_{threshold}$ number of particles. The Octree is adaptive and is able to be modified automatically if the positions of the particles are changed during the simulation. In order to name each cube, a unique id is assigned to each cube in an Octree as follows: the id of a cube, c, is created by taking the id of the cube's parent, p, and computing: $id_c = id_p + 8 \times i$, where i varies from 1 to 9 (the sequence of that parent's sub-cubes). For example, as it is illustrated in Fig.3.17, a cube with $id = 1$, has eight sub-cubes with $9 <= id <= 16$. This technique enables each cube to confirm the ids of its neighbors. The center of mass of each of these cubes is computed using Eq. 3.14 based on the positions and the masses of the particles in that cube [76]:

$$center\_of\_mass_i = mass_{total} \times \sum_{i=1}^{n} r_{p_i} \times mass_{p_i} \tag{3.13}$$

, where $mass_{total}$ is the summation of all of those particles masses.

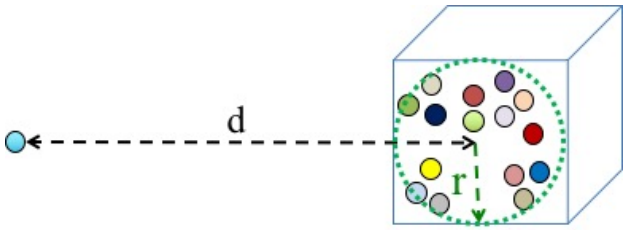


FIGURE 3.18. The gravitational potential of a distant group of particles is approximated as a single particle that is located in the center of mass of those particles.

2. **Interaction List Creation**: After partitioning particles into an Octree, the interaction list for each particle is created by an Octree traversal. In [**?**], we illustrated how we implemented the FMM [77, 78] method in the Barnes-Hut algorithm which reduced the stage time complexity from $O(NlogN)$ to $O(log(1/\epsilon)N)$. This interaction list contains the particles that are close enough to an object to be individually considered as well as a cubes which are collections of particles that are far away and therefore modeled as one object. These remote cubes are treated as a single particle if d is greater than r/θ value; where d is the distance between the cube and that particle, r is the radius of that cluster of those particles, and θ controls the error of the approximation (see Fig.3.18):

   - A larger θ produces more accurate results but increases the application's execution time.
   - A smaller θ produces less accurate results but decreases the application's execution time.

   This parameter should be assigned based on the level of accuracy required for this application. We evaluated the performance of our proposed technique on an NBody application using different θ values in this section.

3. **Force Computation**: In an NBody application, each particle m_i with an initial position r_i and an initial velocity v_i moves under the influence of the gravitational attraction. These gravitational force applied on each is computed by using Newton's law of gravity, Eq. 3.14,

by considering all particles and sub-cubes included in the interaction list of that particle as follow:

$$F_{ij} = \frac{G\, m_j\, m_i\, (r_i - r_j)}{||\, r_j - r_i\, ||^3} \tag{3.14}$$

Also, the velocity and the new position of that particle is computed in Eq.3.15 and Eq.3.16 respectively,

$$v_i = v_i + \frac{F_{ij} \times t}{m_i} \tag{3.15}$$

$$r_i = r_i + v_i \times t + \frac{0.5 \times t^2 \times F_{ij}}{m_i} \tag{3.16}$$

, where t is a time step.

In [26], the performance of an NBody application was evaluated when using HPX parallel loops. It was shown that HPX *future*-based techniques enable the application to continue its progress without waiting for all of the computations in a previous time step to be completed. This effectively results in removing an unnecessary global barrier after each time step. However, as all of the HPX parallel algorithms perform based on the dynamic analysis provided at runtime, this technique is often unable to achieve the maximum possible parallel efficiency. As a result of this challenge, an HPX application's performance may suffer.

To demonstrate the importance of this effect on an application's scalability, we show the performance of an NBody application with $1,000,000$ particles when assigning different chunk sizes in Fig.3.19. The chunk sizes include: 0.1%, 0.2%, 1%, 2%, 10%, 20%, or 50% of the particles. We used the maximum bandwidth measured to determine the most efficient setting for the application. For each series on the graph, we marked the highest value with a "◆" and the lowest value with a "○". As illustrated, the most performant setting was to use 2 NUMA Domains and 16 threads with chunk size $= 0.1\%$ of the particles, which is approximately a 53% percentage difference over the

worst case with chunk size $=$ 50% of the particles. This underscores the fact that choosing an optimum chunk size for each number of threads is necessary for the NBody application to achieve to its highest possible speedup.

Both effective static and dynamic loop's characteristics should be considered for determining its chunk size. We illustrate the performance of our proposed model in the following scenarios:

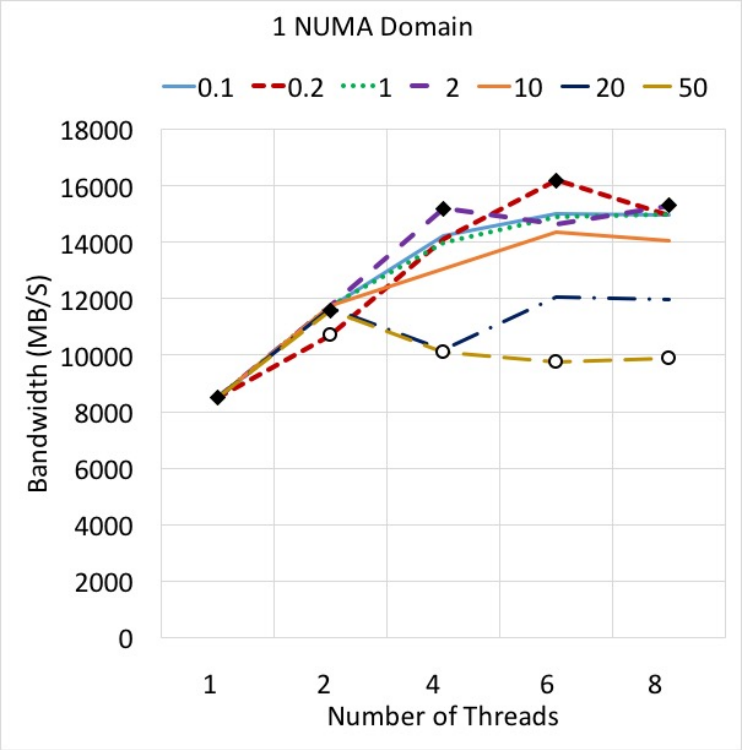
A. Using fixed θ value.

B. Using different θ values.

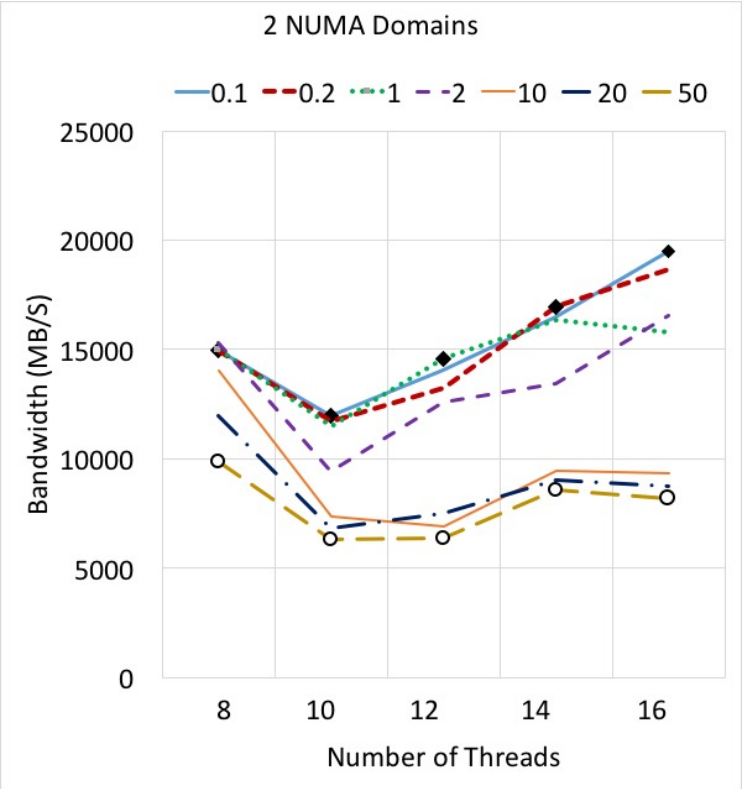
A. Using fixed θ value

In this section we are going to evaluate the performance of the proposed technique on an NBody application using Barnes-Hut algorithm with a fixed value of θ to be 0.01 for different number of the particles. As discussed in Section 3.4.4, the proposed *chunk_size_determination* function exposed by the new execution policy's parameter *adaptive_chunk_size* enables the runtime system to choose an efficient chunk size for the loop by considering static and dynamic features of that loop. Fig.3.20a and 3.20b show the bandwidth rates for an NBody application with different problem sizes – $100,000$, $1,000,000$, and $10,000,000$ particles– on 1 NUMA and 2 NUMA domains respectively. These results compare assigning chunk sizes automatically by using the HPX facility *auto_partitioner* and using the proposed technique discussed in Section 3.4 by annotating the loop with *adaptive_chunk_size*. The results shown with prefix *"(ML)"* are the one with using the proposed technique.

In this case, the application gained up to 33%, 17%, and 19% improvement with application sizes of $100,000$, $1,000,000$, and $10,000,000$ particles respectively. The main reason of this improvement is that efficient chunk size helps in having even amount of work on each number of threads that results in reducing total overheads and latencies. These results illustrate the capability of this method to improve parallel performance of an application simply by intelligently choosing efficient chunk size for the HPX loop.

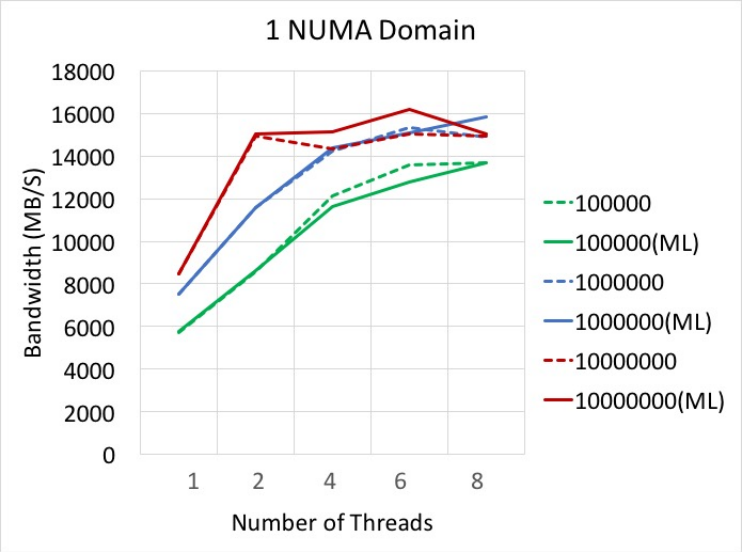


(a)

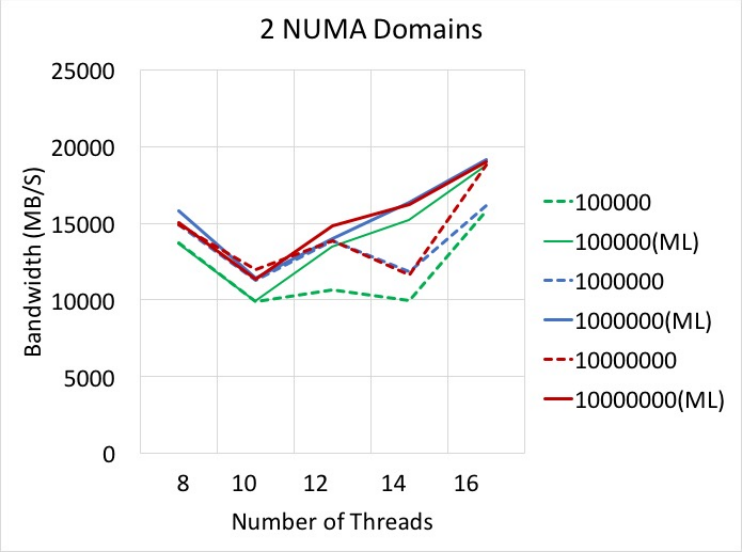


(b)

FIGURE 3.19. NBody parallel performance with $1,000,000$ particles using chunk sizes of: 0.1%, 0.2%, 1%, 2%, 10%, 20%, or 50% of the particles.



(a)



(b)

FIGURE 3.20. NBody parallel performances with $100,000$, $1,000,000$, and $10,000,000$ particles with assigning chunk sizes automatically by an HPX and using the proposed technique discussed in Section 3.4 by annotating loop with *adaptive_chunk_size*. The results shown with prefix *"(ML)"* are the series which use the proposed technique.

B. Using different θ values

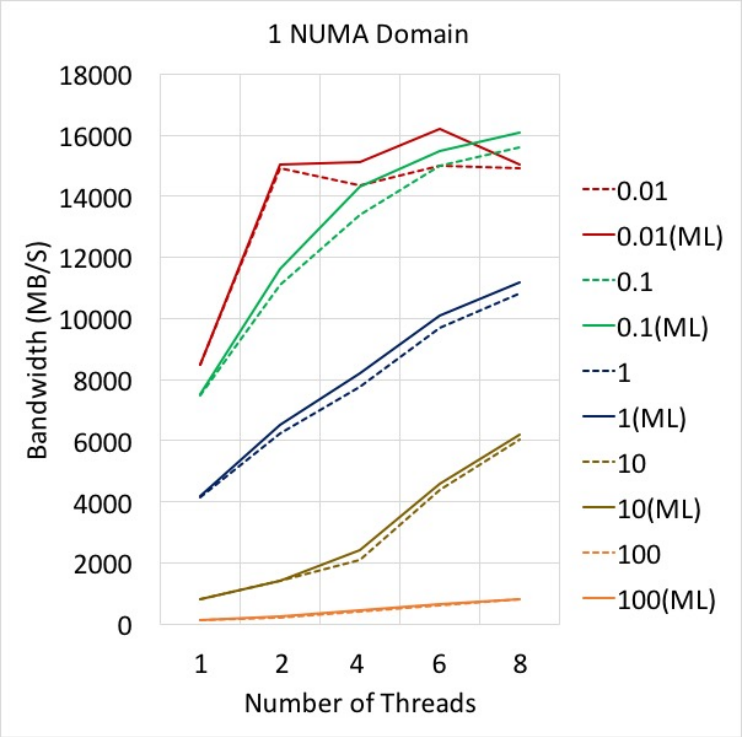
As mentioned in before, θ controls the error of the approximation. For further evaluation, in this section, the performance of the proposed technique is studied when simulating NBody application using different values for θ. Fig.3.21a and 3.21b show the bandwidth rate for an NBody application with $10,000,000$ particles on 1 NUMA and 2 NUMA domains respectively. The results shown with prefix *"(ML)"* are the series which use the proposed technique.

These results compare how *adaptive_chunk_size* adapts to changing values of θ. Larger values of θ implies that more particles are considered to be near objects, which results in a more accurate model. On the other hand, smaller values of θ results in less particles being treated as near objects. Therefore, θ controls the amount of calls to *computing_forces_on_each_particle* within a loop, which causes the execution time of the loop to grow when using larger values of θ and smaller execution times of the loop when using smaller values of θ. We can conclude then, that the chunk size selection process will be affected by the value of θ.

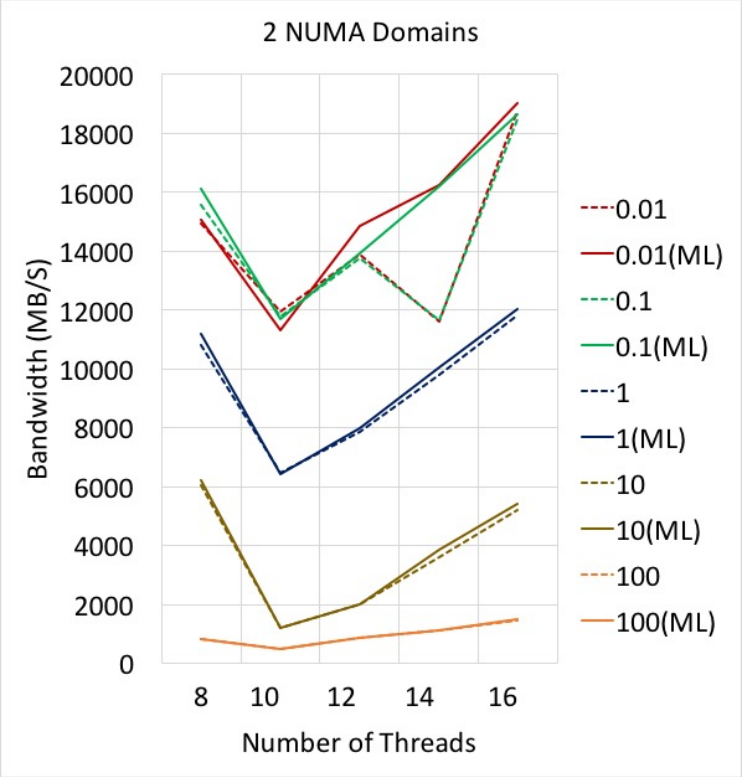
As it is illustrated in these comparison results, the effect of applying the proposed technique to NBody when $\theta = 100$ is negligible. However, this parallel performance is improved up to 17%, 19%, 8% and 3% when θ is changed to 0.01, 0.1, 1 and 10 respectively compared to the existing version of the HPX parallel loop using *auto_partitioner*. It can be concluded that the proposed technique is more effective in improving an NBody parallel performance when it uses a smaller θ value.

3.6 Conclusion

The main goal of this chapter is to illustrate a powerful new set of techniques that can be made available to application developers when compilers, runtime systems, and machine learning algorithms work in concert. These techniques developed here not only greatly improve performance, but users are able to reap their benefit with little cost to themselves. Simply by annotating their code with high level policies, users can see their application's performance increase in a portable



(a)



(b)

FIGURE 3.21. NBody parallel performances with $10,000,000$ particles with assigning different values for θ while using the proposed technique by annotating loop with *adaptive_chunk_size*. The results shown with prefix *"(ML)"* are the series which use the proposed technique.

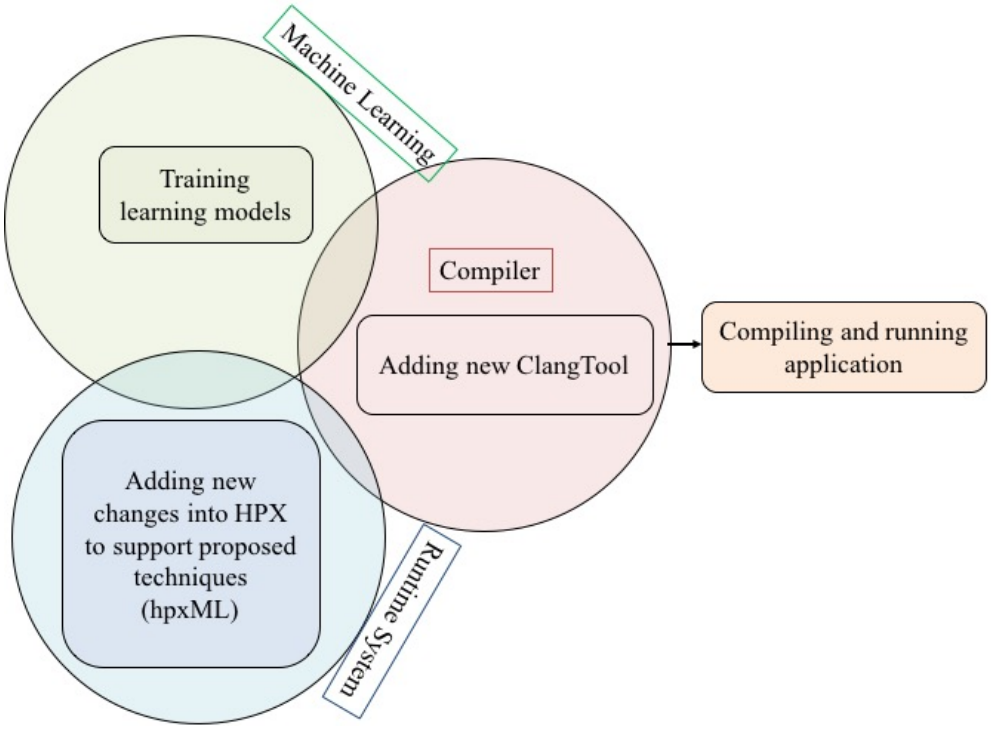


FIGURE 3.22. Shows generally how our proposed methods combines three different fields of machine learning, compiler and runtime together. All of our techniques are publicly available at `https://github.com/STEllAR-GROUP/hpxML`.

way. Fig. 3.22 shows generally how our proposed methods combines three different fields of machine learning, compiler and runtime together.

These results could have broad impact for applications and libraries as well as the maintainers and scientist that use them. The high level annotations increase the usability and therefore accessibility of runtime features that before would have taken a knowledgeable developer to implement. Due to the machine learning element, users will not have to worry about losing performance in different runtime environments that could manifest themselves. Finally, the inclusion of compiler information will allow these performance optimizations to be platform independent. These three features taken together present a notable solution to the challenges presented by an increasingly multi-core and heterogeneous world.

As powerful as these techniques may be, more work is needed to be done in order to fully realize the potential of this work. Notably, the breadth of performance characteristics needs to be

more carefully studied to understand the core features that relate to performance. Additionally more research is needed to ensure that the characteristics measured here also are relevant for other architectures such as the new Knights Landing chipset. On a shorter time scale we intend to investigate extending the number of features for improving the resulting loop's parameters prediction.

In this chapter, we have illustrated that the parallel performance of our test cases were improved by using a machine learning algorithm to determine either an appropriate code path (parallel or sequential loop execution) or certain parameters for the loop execution itself (chunk size or prefetching distance). The speedup results of these test cases and benchmarks showed by around $12\% - 35\%$ improvement compared to selecting execution policy, chunk size and prefetching distance of a loop without using static information and machine learning technique. These results proved that combining machine learning techniques, compiler information, and runtime methods helps an application maximize the available resources.

Chapter 4

Artifact Description

In this chapter, we provide links to the source code for implementing proposed optimization techniques discussed in Chapter 3. The hpxML source code and its build instructions are provided as well as the *LoopConvert* source code. Training data used for designing multinomial logistic regression model are also available. Finally, we include the output results from our benchmarks that one would expect to get.

4.1 Description

4.1.1 Check-list (Artifact Meta Information)

- Algorithm: Logistic regression model.
- Program: C++ code.
- Compilation: Clang 4.0.0.
- Data set: Provided in 4.1.5.
- Run-time environment: Linux Mint 17.2.
- Hardware: x86.
- Output: Bandwidth rate (MB/s).
- Experiment workflow: Git clone hpxML; Make hpxML; Get learning models; Download the datasets; Run learning models on those datasets; Rebuild Clang compiler after adding *LoopConvert*; Compile and run the test script; Observe the measurement results (see fig. 4.1).
- Experiment customization: Provided in 4.5.
- Publicly available?: Yes.

4.1.2 How Delivered

All the source code for implementing proposed optimization techniques are provided as follow.

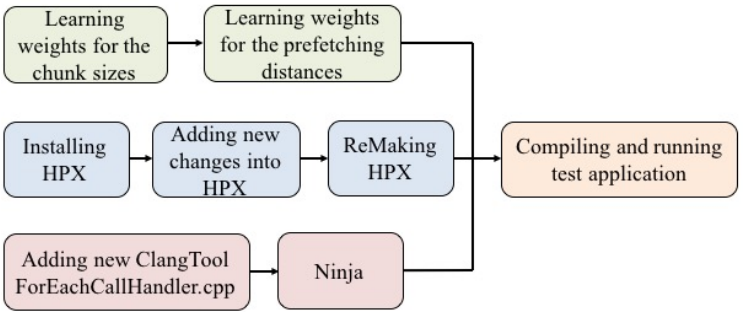


FIGURE 4.1. The general scheme of implementing proposed technique on a test application.

- HPX source code is available from GitHub at `https://github.com/STEllAR-GROUP/HPX`.
- Changes introduced in the current HPX libraries developed in this research for implementing proposed techniques are publicly available at `https://github.com/STEllAR-GROUP/hpxML` with the instructions offering guidance for applying them.
- Multinomial logistic leaning model implemented in C++ is also provided at `https://github.com/STEllAR-GROUP/hpxML/tree/master/logisticRegressionModel`.
- *LoopConvert* source code – our new ClangTool based on Clang's LibTooling – can be found in `https://github.com/STEllAR-GROUP/hpxML/tree/master/ClangTool`.

The general scheme of implementing proposed technique on a test application is shown in fig. 4.1, which illustrates three separate but needed implementations for reproducing the experiments studied in this paper:

−− Steps to design learning model.

−− Steps to prepare HPX runtime system with new changes.

−− Steps to prepare Clang compiler with our ClangTool.

It is described in more details in the following subsections.

4.1.3 Hardware Dependencies

All the new changes and new optimization techniques in this paper are implemented and tested on x86 architectures.

4.1.4 Software Dependencies

On Linux systems, the HPX installation requires Clang 3.3 or newer, CMake 2.8.4 or newer, and Boost 1.50.0 or newer. You can find the full list of HPX prerequisites here: (`http://stellar.cct.lsu.edu/files/hpx-0.9.9/html/hpx/manual/build_system/prerequisites.html`). In this study, the proposed techniques are implemented using Clang 4.0.0, CMake 3.7.1, and Boost 1.63.0.

4.1.5 Datasets

Training data which are used to prime the learning model for predicting execution policy, efficient chunk sizes and prefetching distances are publicly available at `https://github.com/STEllAR-GROUP/hpxML/tree/master/logisticRegressionModel/algorithms/inputs`.

4.2 Installation

4.2.1 Setup HPX with New Changes

First, install Boost 1.63.0 using Clang 4.0.0. Then, follow the build instructions on hpxML's GitHub page (4.1.2). To use Clang 4.0.0 as the compiler add the additional cmake flags below:

```
-DCMAKE_CXX_COMPILER=/path/to/clang++\
-DCMAKE_BUILD_TYPE=Release
```

At this stage, HPX is tooled to support the learning techniques.

4.2.2 Set Up Clang with *LoopConvert*

Add a new ClangTool *LoopConvert* (4.1.2) in a new directory *loop-convert* as follow:

```
$ cd /path/to/clang-llvm/llvm/
$ mkdir tools/clang/tools/extra/loop-convert
$ mv /path/to/LoopConvert.cpp tools/clang/tools/extra/loop-
   convert/
$ echo 'add_subdirectory(loop-convert)' >> tools/extra/
   CMakeLists.txt
```

Compile our new tool as follow. We recommend using *Ninja* for this process.

```
$ cd /path/to/clang-llvm/build
$ ninja loop-convert
```

More details about building tools using LibTooling and LibASTMatchers can be found in `https://clang.llvm.org/docs/LibASTMatchersTutorial.html`.

4.3 Experiment Workflow

4.3.1 Designing The Learning Model

The multinomial logistic learning model can be trained on the data provided in 4.1.5. We have prepared this learning model using a separate C++ code (see 4.1.2) using the Eigen library (`https://eigen.tuxfamily.org/dox/index.html`). To determine the coefficients to be used for the learning model, compile and run the source code available at `https://github.com/STEllAR-GROUP/hpxML/tree/master/logisticRegressionModel/algorithms` with the command below that calls our C++ multinomial logistic learning model on each training dataset*.

```
$ g++ -std=c++11 -I /path/to/include/eigen3/ main.cpp -o main.o
```

After executing *main.o* for each dataset, the trained weights will be stored in separate output files.

4.3.2 Running The Benchmark Using *LoopConvert*

For the artifact evaluation, we supply an instructions to run on our Stream, Stencil, and matrix multiplication benchmark provided at `https://github.com/STEllAR-GROUP/hpxML/tree/master/examples`. Before executing this script, you can find three different loops within each of these applications as follows.

```
for_each(policy, range.begin(), range.end(),f);

for_each(policy.with(adaptive_chunk_size()),range.begin(),range.
   end(),f);

for_each(execution::make_prefetcher_policy(policy,...),range.
   begin(),range.end(),f);
```

After executing this script, *LoopConvert* will be executed on these loops and will modify them as follows.

```
for_each(policy,range.begin(),range.end(),f);
```

[1]For the convenience of this phase, Matlab multinomial logistic learning model (`https://www.mathworks.com/help/stats/mnrfit.html`) is also recommended.

```
for_each(policy.with(chunk_size_determination(
   EXTRACTED_STATICE_DYNAMIC_FEATURES)),range.begin(),range.end
   (),f);

for_each(execution::make_prefetcher_policy(policy,
   prefetching_distance_determination(
   EXTRACTED_STATICE_DYNAMIC_FEATURES),...),range.begin(),range.
   end(),f);
```

At the end, a modified benchmark will be executed utilizing an efficient chunk size and prefetching distance determined from the learning model for these loops.

4.4 Evaluation and Expected Result

The expected results from running the proposed techniques on our applications include bandwidth rates for *for_each* with and without using proposed methods. These results are discussed in more details in section 3.5.3.

4.5 Experiment Customization

In order to reproduce the experimental results discussed in this Appendix, there are two customizations that are already discussed in the previous sections, however we want to restate them here to clear up any confusion:

- The training data provided in 4.1.5 should be normalized before using them to design the learning model, which is considered in our C++ multinomial logistic regression model discussed in 4.3.1.
- Executing provided applications on different numbers of threads is currently included in a provided instruction used in 4.3.2.

Chapter 5

Summary

5.1 Summary of Work

The goal of our study is to propose optimization techniques for implementing scalable parallel applications. This goal is achieved in this research by introducing new techniques for both compiler and runtime system that enable them to contribute with each other and utilize both static and dynamic analysis information to maximize application parallel performance. In the proposed framework, a compiler can implement dynamic runtime methods in its parallelization optimizations and a runtime system can apply static information in its parallelization methods implementation. Our study consists of two projects that are discussed in details in chapters 2 and 3.

In Chapter 2, we propose different optimization methods that provide dynamic information for the code generated by the OP2 compiler. To our knowledge, we present a first attempt of redesigning OP2 to utilize the runtime techniques for improving performance of the parallel unstructured grid applications. In that Chapter, we illustrate the implementation of OP2 compiler that employs the proposed runtime techniques implemented using HPX to efficiently and automatically parallelize the dynamic applications. These optimizations include asynchronous tasking, loop interleaving, dynamic chunk sizing, and data prefetching. The combination of these proposed techniques yield a more portable and performant software stack for unstructured grid applications and enable the applications to properly scale to a higher level of parallelism compared to the existing OP2 implementation.

In Chapter 3 a new technique is proposed that enables HPX runtime system to select its parallel algorithms parameters at runtime by implementing machine learning algorithm on the extracted static and dynamic information. To the best of our knowledge, we present a first attempt in implementing learning models for predicting loop's parameters at runtime, in which designing these runtime techniques and capturing learning model's features are automatically performed at com-

pile time. In that proposed technique, we introduced a new ClangTool using LibTooling as a custom compiler pass to be executed by the Clang compiler, which is intended to collect the static features at compile time. The logistic regression model is implemented as a learning model that considers these captured features for predicting efficient parameters for an HPX loop. For implementing this learning model on a loop, we propose new execution policies that instructs the compiler to apply our ClangTool on that loop. As a results, the loop's features will automatically be included in the prediction process implemented with that learning model. This technique is able to use high-level programming abstractions and machine learning to relieve the programmer of difficult and tedious decisions that can significantly affect program behavior and performance.

As it was illustrated in the experimental results discussed in both chapters, the scalable framework is developed in this research that utilizes both static and dynamic information about the system to make proper optimizations. Both the compiler and the runtime exchange parameters that can be used to optimize their performance and achieve a safe parallelization. In next chapter, we discuss a brief summary of our planned contributions and present some interesting future research ideas.

Chapter 6

Future Work

In this chapter, we discuss a brief summary of our planned contributions and present some interesting future research ideas.

6.1 Improving OP2 Compiler with HPX Runtime System

OP2 framework is presented in chapter 2 and it is shown that how unstructured grids can be implemented in this framework without worrying about the underlying architecture hardware. OP2 is designed to solve an overhead challenge caused by the data exchanged between main memory and processors in a case of having large unstructured grids. All of its optimization techniques, however, are based on the static information and they are insufficient for achieving maximal parallelism level. For obtaining desired scalability using this framework, some optimizations should be applied at runtime, so OP2 could be able to include dynamic information in its optimization techniques.

Higher parallel performance for code generated with the OP2 compiler are achieved in chapter 2 by delaying some compiler optimizations to be managed at runtime. These proposed optimization methods are implemented with the HPX runtime system that makes OP2 to provide asynchronous tasking, loop interleaving, dynamic chunk sizing, and data prefetching for the parallel execution of the unstructured grid applications. The experimental results on an Airfoil application showed up to 50% improvement in its parallel scalability. This improvement is mostly due to the use of the *future* based techniques that allow HPX to relax the global barriers, enable flexibility, and improve the parallel performance. Some interesting future research ideas are as follows.

- All parallelization techniques discussed in chapter 2 are proposed for implementing parallelization within a node, not between nodes. In the current OP2 framework, MPI is used for executing distributed unstructured grids on more than one node. However as discussed

in Chapter 1 and 2, obtaining desired scalability may be hindered because of the following problems.

1. Same as an in-node parallelization, for between nodes parallelization in the current OP2 design, only static information are considered for establishing connection between two different nodes and exchanging data between them. We believe that including dynamic information and runtime techniques in these methods can result in achieving higher performance from parallel and distributed unstructured grids applications.
2. As discussed in Chapter 1, the challenges faced in using MPI for between nodes parallelization such as providing unnecessary global barriers and causing load imbalances between different nodes, may result in having not scalable applications for some unstructured grids. We believe that fine-grained task parallelism and the *future*-based parallelization techniques provided with the HPX runtime system can resolve these challenges and result in having better load balance and relaxing unnecessary global barriers.

- As discussed in section 2.4.3.2, the proposed technique using *dataflow* is able to set different chunk sizes but with the same execution time for the loops that their execution are dependent on each other. As it was shown, in this method *for_each* sets a chunk size of the first loop dynamically by exposing *auto_partitioner* and then the chunk sizes of the rest of the loops will be set by considering the execution time of the chunk size of the first loop. However setting all chunks based on the chunk of the first loop may not be an optimum solution. It should be evaluated in more studies about the chunk size selection and to see which loop should be selected as a base for setting other chunk sizes.

6.2 Improving HPX Runtime System with Clang Compiler

In the current HPX design, most of the parameters of its algorithms should be manually selected by a user at runtime. However this approach doesn't guarantee achieving desired speedup in the

HPX parallel applications. Both static and dynamic features of an application should be considered in assigning such parameters. Collecting static features and related transformations at compile time could result in tuning those parameters efficiently to maximally utilize available resources.

Higher parallel performance for the HPX parallel applications are achieved in chapter 3 by combining machine learning, compiler optimizations and runtime techniques. In these proposed techniques, the HPX loop's parameters are automatically selected with implementing logistic regression model on the loop's characteristics, that its static features are collected by a compiler and its dynamic features are provided by the runtime system. Efficient execution policy, chunk size, and prefetching distance of an HPX loop can be predicted with these techniques. This prediction process is performed at runtime by computing outputs of the learning model, which avoids an extra compilation step. Some interesting future research ideas are as follows.

- Although the evaluated results discussed in section 3.5 show an improvement in an HPX parallel application, however more studies should be done in selecting efficient features set for a learning model to cover more various applications. Collecting architecture's characteristics in this feature set could help the proposed techniques to predict a proper architecture hardware for a particular application.
- The learning models in these proposed methods are designed offline with using collected training data. One of the major difficulty in this offline learning is that this learning model should be set at once, hence, this training data should be general enough to cover various cases. If not sufficient training data is collected, these proposed techniques may not predict the parameters for the future new applications efficiently. On the other hand, online training data could help us to update our learning model in each step for future data. By implementing online machine learning method, the learning algorithms can dynamically adapt their trained weights to new patterns in the data.

- In the proposed methods, executing HPX algorithm sequentially or in parallel is determined automatically at runtime based on the output of a binary logistic regression model using the values of algorithm's characteristics. This method can be extended to use a multinomial logistic regression model for determining loop's execution policy other than just *seq* or *par* and choosing the efficient one between all 5 policies described in Table 2.1.
- These proposed techniques can be implemented on an HPX algorithm by using specific execution policies. However, their performances are evaluated only on an HPX loop. As these specific policies are designed to be used for all of the HPX parallel algorithms, so implementing them on all of these algorithms should be studied for examining their effectiveness on these algorithms performances.

References

[1] G. Mudalige, M. Giles, I. Reguly, C. Bertolli, and P. Kelly, "Op2: An active library framework for solving unstructured mesh-based applications on multi-core and many-core architectures," in *Innovative Parallel Computing (InPar), 2012*. IEEE, 2012, pp. 1–12.

[2] M. B. Giles, G. R. Mudalige, Z. Sharif, G. Markall, and P. H. Kelly, "Performance analysis and optimization of the op2 framework on many-core architectures," *The Computer Journal*, p. bxr062, 2011.

[3] G. Mudalige, M. Giles, B. Spencer, C. Bertolli, and I. Reguly, "Designing op2 for gpu architectures," *Journal of Parallel and Distributed Computing*, 2012.

[4] C. Bertolli, A. Betts, P. H. Kelly, G. R. Mudalige, and M. B. Giles, "Mesh independent loop fusion for unstructured mesh applications," in *Proceedings of the 9th conference on Computing Frontiers*. ACM, 2012, pp. 43–52.

[5] D. Patel and L. Rauchwerger, "Implementation issues of loop-level speculative run-time parallelization," in *International Conference on Compiler Construction*. Springer, 1999, pp. 183–197.

[6] L. Rauchwerger, N. M. Amato, and D. A. Padua, "A scalable method for run-time loop parallelization," *International Journal of Parallel Programming*, vol. 23, no. 6, pp. 537–576, 1995.

[7] ——, "Run-time methods for parallelizing partially parallel loops," in *Proceedings of the 9th international conference on Supercomputing*. ACM, 1995, pp. 137–146.

[8] B. Calder, D. Grunwald, M. Jones, D. Lindsay, J. Martin, M. Mozer, and B. Zorn, "Evidence-based static branch prediction using machine learning," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 19, no. 1, pp. 188–222, 1997.

[9] M. Stephenson and S. Amarasinghe, "Predicting unroll factors using supervised classification," in *Code Generation and Optimization, 2005. CGO 2005. International Symposium on*. IEEE, 2005, pp. 123–134.

[10] J. Cavazos and M. F. O'boyle, "Method-specific dynamic compilation using logistic regression," *ACM SIGPLAN Notices*, vol. 41, no. 10, pp. 229–240, 2006.

[11] M. Stephenson, S. Amarasinghe, M. Martin, and U.-M. O'Reilly, "Meta optimization: improving compiler heuristics with machine learning," in *ACM SIGPLAN Notices*, vol. 38, no. 5. ACM, 2003, pp. 77–90.

[12] A. Joshi, A. Phansalkar, L. Eeckhout, and L. K. John, "Measuring benchmark similarity using inherent program characteristics," *IEEE Transactions on Computers*, vol. 55, no. 6, pp. 769–782, 2006.

[13] L. Smith, "Mixed mode MPI/OpenMP programming," *UK High-End Computing Technology Report*, pp. 1–25, 2000.

[14] R. Rabenseifner, G. Hager, and G. Jost, "Hybrid MPI/OpenMP parallel programming on clusters of multi-core SMP nodes," in *Parallel, Distributed and Network-based Processing, 2009 17th Euromicro International Conference on*. IEEE, 2009, pp. 427–436.

[15] L. Smith and M. Bull, "Development of mixed mode MPI/OpenMP applications," *Scientific Programming*, vol. 9, no. 2-3, pp. 83–98, 2001.

[16] J. M. Bull, "Measuring synchronization and scheduling overheads in OpenMP," in *Proceedings of First European Workshop on OpenMP*, vol. 8, 1999, p. 49.

[17] R. Aversa, B. Di Martino, N. Mazzocca, and S. Venticinque, "Performance analysis of hybrid OpenMP/MPI n-body application," in *Shared Memory Parallel Programming with Open MP*. Springer, 2004, pp. 12–18.

[18] R. Rabenseifner, G. Hager, G. Jost, and R. Keller, "Hybrid MPI and OpenMP parallel programming," in *PVM/MPI*, 2006, p. 11.

[19] H. Kaiser, T. Heller, B. Adelstein-Lelbach, A. Serio, and D. Fey, "HPX: A Task Based Programming Model in a Global Address Space," in *Proceedings of the 8th International Conference on Partitioned Global Address Space Programming Models*. ACM, 2014, p. 6.

[20] I. Ganusov and M. Burtscher, "Efficient emulation of hardware prefetchers via event-driven helper threading," in *Proceedings of the 15th international conference on Parallel architectures and compilation techniques*. ACM, 2006, pp. 144–153.

[21] J. Collins, S. Sair, B. Calder, and D. M. Tullsen, "Pointer cache assisted prefetching," in *Proceedings of the 35th annual ACM/IEEE international symposium on Microarchitecture*. IEEE Computer Society Press, 2002, pp. 62–73.

[22] H. Kaiser, T. Heller, D. Bourgeois, and D. Fey, "Higher-level parallelization for local and distributed asynchronous task-based programming," in *Proceedings of the First International Workshop on Extreme Scale Programming Models and Middleware*. ACM, 2015, pp. 29–37.

[23] R. Raj, "Performance analysis with hpx," Ph.D. dissertation, Louisiana State University, 2014.

[24] H. C. Baker Jr and C. Hewitt, "The incremental garbage collection of processes," in *ACM Sigplan Notices*, vol. 12, no. 8. ACM, 1977, pp. 55–59.

[25] Z. Khatami, H. Kaiser, and J. Ramanujam, "Using hpx and op2 for improving parallel scaling performance of unstructured grid applications," in *Parallel Processing Workshops (ICPPW), 2016 45th International Conference on*. IEEE, 2016, pp. 190–199.

[26] Z. Khatami, H. Kaiser, P. Grubel, A. Serio, and J. Ramanujam, "A massively parallel distributed n-body application implemented with hpx," in *Proceedings of the 7th Workshop on Latest Advances in Scalable Algorithms for Large-Scale Systems*. IEEE Press, 2016, pp. 57–64.

[27] C. Bishop, "Pattern recognition and machine learning (information science and statistics), 1st edn. 2006. corr. 2nd printing edn," *Springer, New York*, 2007.

[28] T. Heller, H. Kaiser, A. Schäfer, and D. Fey, "Using HPX and LibGeoDecomp for scaling HPC applications on heterogeneous supercomputers," in *Proceedings of the Workshop on Latest Advances in Scalable Algorithms for Large-Scale Systems*. ACM, 2013, p. 1.

[29] T. Heller, H. Kaiser, and K. Iglberger, "Application of the ParalleX execution model to stencil-based problems," *Computer Science-Research and Development*, vol. 28, no. 2-3, pp. 253–261, 2013.

[30] P. Grubel, H. Kaiser, J. Cook, and A. Serio, "The Performance Implication of Task Size for Applications on the HPX Runtime System," in *Cluster Computing (CLUSTER), 2015 IEEE International Conference on*. IEEE, 2015, pp. 682–689.

[31] H. C. Baker Jr and C. Hewitt, "The incremental garbage collection of processes," in *ACM Sigplan Notices*, vol. 12, no. 8. ACM, 1977, pp. 55–59.

[32] C. Bertolli, A. Betts, G. Mudalige, M. Giles, and P. Kelly, "Design and performance of the op2 library for unstructured mesh applications," in *Euro-Par 2011: Parallel Processing Workshops*. Springer, 2011, pp. 191–200.

[33] M. B. Giles, G. R. Mudalige, Z. Sharif, G. Markall, and P. H. Kelly, "Performance analysis of the op2 framework on many-core architectures," *ACM SIGMETRICS Performance Evaluation Review*, vol. 38, no. 4, pp. 9–15, 2011.

[34] M. Giles, D. Ghate, and M. Duta, "Using automatic differentiation for adjoint cfd code development," 2005.

[35] H. Kaiser, M. Brodowicz, and T. Sterling, "Parallex an advanced parallel execution model for scaling-impaired applications," in *Parallel Processing Workshops, 2009. ICPPW'09. International Conference on*. IEEE, 2009, pp. 394–401.

[36] P. A. Grubel, "Dynamic adaptation in hpx-a task-based parallel runtime system," Ph.D. dissertation, New Mexico State University, 2016.

[37] H. Kaiser, T. Heller, A. Berge, and B. Adelstein-Lelbach, "HPX V0.9.11: A general purpose C++ runtime system for parallel and distributed applications of any scale," 2015, http://github.com/STEllAR-GROUP/hpx. [Online]. Available: http://dx.doi.org/10.5281/zenodo.33656

[38] T. Heller, H. Kaiser, P. Diehl, D. Fey, and M. A. Schweitzer, "Closing the performance gap with modern c++," in *International Conference on High Performance Computing*. Springer,

2016, pp. 18–31.

[39] C. Dekate, "Extreme scale parallel nbody algorithm with event driven constraint based execution model," Ph.D. dissertation, Citeseer, 2011.

[40] P. Grubel, H. Kaiser, K. Huck, and J. Cook, "Using intrinsic performance counters to assess efficiency in task-based parallel applications."

[41] "N4406: Parallel algorithms need executors. technical report," http://www.openstd.org/jtc1/sc22/wg21/docs/papers/2015/n4406.pdf, accessed: 2015.

[42] J. Lee, C. Jung, D. Lim, and Y. Solihin, "Prefetching with helper threads for loosely coupled multiprocessor systems," *Parallel and Distributed Systems, IEEE Transactions on*, vol. 20, no. 9, pp. 1309–1324, 2009.

[43] A.-H. Badawy, A. Aggarwal, D. Yeung, and C.-W. Tseng, "The efficacy of software prefetching and locality optimizations on future memory systems," *Journal of Instruction-Level Parallelism*, vol. 6, no. 7, 2004.

[44] D. Callahan, K. Kennedy, and A. Porterfield, "Software prefetching," in *ACM SIGARCH Computer Architecture News*, vol. 19, no. 2. ACM, 1991, pp. 40–52.

[45] K. Bala, M. F. Kaashoek, and W. E. Weihl, "Software prefetching and caching for translation lookaside buffers," in *Proceedings of the 1st USENIX conference on Operating Systems Design and Implementation*. USENIX Association, 1994, p. 18.

[46] D. F. Zucker, R. B. Lee, and M. J. Flynn, "Hardware and software cache prefetching techniques for mpeg benchmarks," *IEEE Transactions on Circuits and Systems for Video Technology*, vol. 10, no. 5, pp. 782–796, 2000.

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

[48] K. D. Cooper, D. Subramanian, and L. Torczon, "Adaptive optimizing compilers for the 21st century," *The Journal of Supercomputing*, vol. 23, no. 1, pp. 7–22, 2001.

[49] Z. Khatami, H. Kaiser, and J. Ramanujam, "Hpx data prefetching iterator," *Woman in High Performance Computing*.

[50] ——, "Redesigning op2 compiler to use hpx runtime asynchronous techniques," in *Parallel and Distributed Scientific and Engineering Computing (IPDPSW), 2017 18th IEEE International Workshop on*. IEEE, 2017.

[51] T. C. Team. (2017) Clang 5 documentation, LibTooling. https://clang.llvm.org/docs/LibTooling.html. [Online; accessed 1-Feb-2017].

[52] L. Rauchwerger, N. M. Amato, and D. A. Padua, "A scalable method for run-time loop parallelization," *International Journal of Parallel Programming*, vol. 23, no. 6, pp. 537–576,

1995.

[53] ——, "Run-time methods for parallelizing partially parallel loops," in *Proceedings of the 9th international conference on Supercomputing*. ACM, 1995, pp. 137–146.

[54] K. Seymour, H. You, and J. Dongarra, "A comparison of search heuristics for empirical code optimization," in *Cluster Computing, 2008 IEEE International Conference on*. IEEE, 2008, pp. 421–429.

[55] Z. Pan and R. Eigenmann, "Fast, automatic, procedure-level performance tuning," in *Proceedings of the 15th international conference on Parallel architectures and compilation techniques*. ACM, 2006, pp. 173–181.

[56] F. Agakov, E. Bonilla, J. Cavazos, B. Franke, G. Fursin, M. F. O'Boyle, J. Thomson, M. Toussaint, and C. K. Williams, "Using machine learning to focus iterative optimization," in *Proceedings of the International Symposium on Code Generation and Optimization*. IEEE Computer Society, 2006, pp. 295–305.

[57] C. Dubach, J. Cavazos, B. Franke, G. Fursin, M. F. O'Boyle, and O. Temam, "Fast compiler optimisation evaluation using code-feature based performance prediction," in *Proceedings of the 4th international conference on Computing frontiers*. ACM, 2007, pp. 131–142.

[58] G. Pekhimenko and A. D. Brown, "Efficient program compilation through machine learning techniques," in *Software Automatic Tuning*. Springer, 2011, pp. 335–351.

[59] A. Hayashi, K. Ishizaki, G. Koblents, and V. Sarkar, "Machine-learning-based performance heuristics for runtime cpu/gpu selection," in *Proceedings of the principles and practices of programming on the Java platform*. ACM, 2015, pp. 27–36.

[60] S. Pellegrini, J. Wang, T. Fahringer, and H. Moritsch, "Optimizing mpi runtime parameter settings by using machine learning," in *European Parallel Virtual Machine/Message Passing Interface Users? Group Meeting*. Springer, 2009, pp. 196–206.

[61] "Method-specific dynamic compilation using logistic regression."

[62] T. J. Ypma, "Historical development of the newton–raphson method," *SIAM review*, vol. 37, no. 4, pp. 531–551, 1995.

[63] I. Nabney, *NETLAB: algorithms for pattern recognition*. Springer Science & Business Media, 2002.

[64] C. A. Ratanamahatana and D. Gunopulos, "Scaling up the naive bayesian classifier: Using decision trees for feature selection," 2002.

[65] W.-Y. Loh, "Classification and regression trees," *Wiley Interdisciplinary Reviews: Data Mining and Knowledge Discovery*, vol. 1, no. 1, pp. 14–23, 2011.

[66] L. Rokach and O. Maimon, *Data mining with decision trees: theory and applications*. World scientific, 2014.

[67] W.-Y. Loh, "Classification and regression trees," *Wiley Interdisciplinary Reviews: Data Mining and Knowledge Discovery*, vol. 1, no. 1, pp. 14–23, 2011.

[68] J. McCalpin, "Stream: Sustainable memory bandwidth in high performance computers (2008)," 1991-2007.

[69] ——, "Memory bandwidth and machine balance in current high performance computers," in *Committee on Computer Architecture (TCCA) Newsletter*. IEEE, 1995, pp. 19–25.

[70] Y. Kwon, D. Nunley, J. P. Gardner, M. Balazinska, B. Howe, and S. Loebman, "Scalable clustering algorithm for N-body simulations in a shared-nothing cluster," in *Scientific and Statistical Database Management*. Springer, 2010, pp. 132–150.

[71] K. S. Pedersen and B. Vinter, "Java PastSet: A Structured Distributed Shared Memory System," in *Software, IEE Proceedings-*, vol. 150, no. 2. IET, 2003, pp. 147–153.

[72] R. Speck, D. Ruprecht, R. Krause, M. Emmett, M. Minion, M. Winkel, and P. Gibbon, "A massively space-time parallel N-body solver," in *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*. IEEE Computer Society Press, 2012, p. 92.

[73] J. P. Singh, C. Holt, T. Totsuka, A. Gupta, and J. Hennessy, "Load balancing and data locality in adaptive hierarchical N-body methods: Barnes-Hut, fast multipole, and radiosity," *Journal of Parallel and Distributed Computing*, vol. 27, no. 2, pp. 118–141, 1995.

[74] R. Speck, L. Arnold, and P. Gibbon, "Towards a petascale tree code: Scaling and efficiency of the PEPC library," *Journal of Computational Science*, vol. 2, no. 2, pp. 138–143, 2011.

[75] H. Sundar, R. S. Sampath, and G. Biros, "Bottom-up construction and 2: 1 balance refinement of linear octrees in parallel," *SIAM Journal on Scientific Computing*, vol. 30, no. 5, pp. 2675–2708, 2008.

[76] C. Dekate, M. Anderson, M. Brodowicz, H. Kaiser, B. Adelstein-Lelbach, and T. Sterling, "Improving the scalability of parallel N-body applications with an event-driven constraint-based execution model," *International Journal of High Performance Computing Applications*, 2012.

[77] E. Agullo, B. Bramas, O. Coulaud, E. Darve, M. Messner, and T. Takahashi, "Task-based fmm for multicore architectures," *SIAM Journal on Scientific Computing*, vol. 36, no. 1, pp. C66–C93, 2014.

[78] ——, "Task-based fmm for heterogeneous architectures," *Concurrency and Computation: Practice and Experience*, 2015.

Appendix A: Permissions

Gmail **Zahra Khatami <z.khatami88@gmail.com>**

Case #00359052 - Permission Request [ref:_00D30oeGz._5000c1OEups:ref]

customercare@copyright.com <customercare@copyright.com> Tue, Aug 15, 2017 at 10:20 AM
To: "zkhata1@lsu.edu" <zkhata1@lsu.edu>

Dear Zahra Khatami,

Thank you for contacting Copyright Clearance Center. I understand that you are looking for permission to reuse material from IEEE in your thesis. I have gone to the ordering page, and it appears that IEEE simply includes a statement showing that you have permission, and no formal order needs to be placed:

Copyright Clearance Center RightsLink® Home Account Info Help

IEEE Requesting permission to reuse content from an IEEE publication

Title:	Redesigning OP2 Compiler to Use HPX Runtime Asynchronous Techniques
Conference Proceedings:	Parallel and Distributed Processing Symposium Workshops (IPDPSW), 2017 IEEE International
Author:	Zahra Khatami
Publisher:	IEEE
Date:	May 2017

Copyright © 2017, 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 © 2011 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 on-line.
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.

BACK CLOSE WINDOW

You may save the attachment of this image as proof that you do not need permission for this use. If you have any further questions please don't hesitate to contact a Customer Account Specialist at 855-239-3415 Monday-Friday, 24 hours/day.

Best,

Amy Larocque
Customer Account Specialist
Copyright Clearance Center
222 Rosewood Drive
Danvers, MA 01923
www.copyright.com
+1.855.239.3415

Facebook - Twitter - LinkedIn

ref:_00D30oeGz._5000c1OEups:ref

---------- Original Message ----------
From: zahra khatami [zkhata1@lsu.edu]
Sent: 8/15/2017 12:59 PM
To: customercare@copyright.com
Subject: Permission Request

To whom it may concern,

I am a first author of two papers published in IEE with the title of "Redesigning op2 compiler to use hpx runtime asynchronous techniques" with DOI:10.1109/IPDPSW.2017.14 and "Using hpx and op2 for improving parallel scaling performance of unstructured grid applications" with DOI: 10.1109/ICPPW.2016.39.
I would like to include some parts of these papers in one of the chapters in my PhD dissertation.
I have tried to request for a written permission from IEEE website, however I was not successful.
I am wondering if you could guide me on how to receive such a permission from IEEE.
Thanks in advance.

Best Regards,

Zahra Khatami | PhD Student
Center for Computation & Technology (CCT)
School of Electrical Engineering & Computer Science
Louisiana State University
2027 Digital Media Center (DMC)
Baton Rouge, LA 70803

.

ref:_00D30oeGz._5000c1OEups:ref

IEEE.jpg
352K

Vita

Zahra Khatami received her BSc in Electrical Engineering from the Zanjan University (ZNU) , Zanjan, Iran in 2010. From 2010 to 2013, she worked as a Research Assistant during her M.S. at the Amirkabir University of Technology (AUT), Tehran, Iran. From 2014 to 2017, she studied computer engineering/software as a PhD student at the school of Electrical Engineering and Computer Science (EECS) at the Louisiana State University (LSU), Baton Rouge, LA. She has completed several projects in the field of software development, parallel computing and high performance computing during past years. In those years, she had an opportunity to work as a software developer with Ste||ar Group at Center for Computation and Technology located at LSU for developing HPX, which is a general purpose C++ runtime system for parallel and distributed applications of any scale. Also, she had worked as a software developer for PGX Group at Software Lab located at Oracle for developing PGX.Dist, which is a fast, parallel, in-memory graph analytic framework. Her researches and studies has been published in several conference papers and posters.