HPX-RTE: A LIGHTWEIGHT RUNTIME ENVIRONMENT FOR OPEN MPI

A Thesis

Presented to

the Faculty of the Department of Computer Science

University of Houston

In Partial Fulfillment

of the Requirements for the Degree

Master of Science

By

Hadi Montakhabi

December 2015

HPX-RTE: A LIGHTWEIGHT RUNTIME ENVIRONMENT FOR OPEN MPI

Hadi Montakhabi

APPROVED:

Dr. Edgar Gabriel
Dept. of Computer Science, University of Houston

Dr. Jaspal Subhlok
Dept. of Computer Science, University of Houston

Dr. Rakhi Anand
Intel Corporation

Dean, College of Natural Sciences and Mathematics

HPX-RTE: A LIGHTWEIGHT RUNTIME ENVIRONMENT FOR OPEN MPI

An Abstract of a Thesis

Presented to

the Faculty of the Department of Computer Science

University of Houston

In Partial Fulfillment

of the Requirements for the Degree

Master of Science

By

Hadi Montakhabi

December 2015

Abstract

High-performance computing systems are growing toward hundreds-of-thousands to million-node machines, utilizing the computing power of billions of cores. Running parallel applications on such large machines efficiently will require optimized runtime environments that are scalable and resilient. Multi- and many-core chip architectures in large-scale supercomputers pose several new challenges to designers of operating systems and runtime environments.

ParalleX is a general-purpose parallel-execution model aiming to overcome the limitations imposed by the current hardware and the way we write applications today. High-Performance ParalleX (HPX) is an experimental runtime system for ParalleX.

The majority of scientific and commercial applications in HPC are written in MPI. In order to facilitate the transition from MPI model to ParalleX, there is a need for a compatibility mechanism between the two. Currently, this mechanism does not exist. This thesis provides a compatibility mechanism for MPI applications to use the HPX runtime system. This is achieved by developing a new runtime system for the Open MPI project, an open source implementation of MPI. This new runtime system is called HPX-RTE.

HPX-RTE is a new, lightweight, and open-source runtime system specifically designed for the emerging exascale computing environment. The system is designed relying on HPX project advanced features to allow for easy extension and transparent scalability. HPX-RTE provides full compatibility for current MPI applications to run on HPX runtime system. HPX-RTE provides an easy and simple path for transition from MPI to HPX. It also paves the way for future hybrid programming models such as HPX-MPI and integration of more features from HPX into Open MPI.

Contents

1 Introduction **1**

1.1 Parallel Architectures 2

1.2 Parallel Computer Memory Architectures 3

1.2.1 Shared Memory 3

1.2.2 Distributed Memory 4

1.2.3 Hybrid Distributed-Shared Memory 4

1.3 Parallel Programming Models 5

1.4 Message Passing Interface (MPI) 7

1.5 Runtime Environments and MPI Jobs 10

1.6 Challenges for Runtime Environments 10

1.7 The Goal of this Thesis 11

1.8 Contributions 11

1.9 Organization of the Document 12

2 Background **13**

2.1 Open MPI 13

2.1.1 The Architectire of Open MPI 14

2.2 Open Runtime Environment (ORTE) 17
2.3 ParalleX 21
2.4 High-Performance ParalleX (HPX) 24
2.4.1 HPX Design Principles 25
2.4.2 HPX Implementation Features 26
2.4.3 HPX Hello World Example 30

3 HPX-RTE 34

3.1 Design Principles 36
3.2 Architecture 38
3.3 Runtime Environement Requirements 39
3.3.1 Process Name Objects and Operations 39
3.3.2 Collective Objects and Operations 41
3.3.3 Process Information Structure 42
3.3.4 Error-Handling Objects and Operations 43
3.3.5 Initializing and Finalizing Objects and Operations 44
3.3.6 Database Operations 45
3.4 Implementation 46
3.4.1 HPX-RTE Requirements 46
3.4.2 Features 46

4 Evaluation 53

4.1 Test Environment 53
4.1.1 Crill Cluster Hardware Specification 53
4.1.2 Crill Cluster Nodes Software Specification 54

4.2 Configuration, Compilation, and Execution 55
4.2.1 Open MPI Configuration Parameteres 55
4.2.2 Compiling MPI Applications 56
4.2.3 Running MPI Applications 56
4.3 Evaluation 57
4.3.1 Parallel Computational Fluid Dynamics 58
4.3.2 MPI Hello World 61
4.3.3 Parallel Smoothing 63
4.3.4 Code Size 66

5 Conclusion **67**

5.1 Performance 68
5.2 Future Extensions 68

Bibliography **70**

List of Figures

1.1 Shared Memory (UMA) 4

1.2 Shared Memory (NUMA) 5

1.3 Distributed Memory 6

1.4 Hybrid Distributed-Shared Memory (Multi-core nodes) 7

1.5 Hybrid Distributed-Shared Memory (Multi-core nodes with GPUs) . 8

2.1 MCA, component frameworks, and the components 15

2.2 Open MPI Layers 17

2.3 The ORTE architecture 20

2.4 Modular Structure of HPX Implementation 27

2.5 Overview of the main API exposed by HPX 28

2.6 Schematic of a Future Execution 29

3.1 The OpenX Software Architecture 35

3.2 Modular Component Architecure with rte Framework 37

3.3 Open MPI Layers using HPX-RTE 38

4.1 CFD Running Time - 100x100x100 59

4.2 CFD Application Reported Time - 100x100x100 59

4.3 CFD Running Time - 80x80x80 60
4.4 CFD Application Reported Time - 80x80x80 60
4.5 Time - Hello World 62
4.6 Application Reported Time - Hello World 62
4.7 Time - Parallel Smoothing - 1024x1024 64
4.8 Application Reported Time - Parallel Smoothing - 1024x1024 64
4.9 Time - Parallel Smoothing - 2048x2048 65
4.10 Application Reported Time - Parallel Smoothing - 2048x2048 65
4.11 Code Size Comparison - HPX-RTE and ORTE 66

List of Tables

Chapter 1

Introduction

Traditionally, software is written for sequential computation. A problem is broken down into a series of instrcutions. Those instructions are executed sequentially on a single processor. In this scenario, only one instruction is executed at any given moment in time [1].

Parallel computing is a type of computation in which calculations are carried out simultaneously to solve a computational problem [2]. In parallel computing, a problem is broken down into discrete parts that can be solved concurrently. Each part includes a series of instructions. Instructions for each part of the problem are executed in parallel on different processors. In this scenario, a control mechanism among all the processors is needed.

1.1 Parallel Architectures

Parallel computers can be classified in different ways. Flynn's taxonomy is one of the more widely used classification of different computer architectures proposed by Michael J. Flynn in 1966 [3, 4]. This classification is based on the number of concurrent instruction streams and data streams available in the architecture. There are four categories defined:

1. **Single Instruction, Single Data Stream (SISD)** In this architecture, a single processor executes a single instruction stream and only one data stream is available as an input during any given clock cycle. This is basically a serial computer.

2. **Single Instruction, Multiple Data Streams (SIMD)** This architecture describes computers with multiple processing elements which execute the same instruction at any given clock cycle. Each processing unit can use a different data stream.

3. **Multiple Instruction, Single Data Stream (MISD)** In this architecture, multiple processing units operate on a single data stream, each one having a separate set of instructions. This architecture is almost non-existent.

4. **Multiple Instruction, Multiple Data Streams (MIMD)** In this architecture, multiple independent processors execute different instructions on different data streams at the same time. Execution can be synchronous or asynchronous. This is the most generic category. Most current supercomputers fall under this category.

1.2 Parallel Computer Memory Architectures

Main memory in a parallel computer is either shared or distributed [5].

1.2.1 Shared Memory

In a shared-memory parallel computer, the memory is shared between all processing elements as a single global address space. Multiple processors can share the same memory resources and operate independently. Changes in a memory location by one processor are visible to all other processors.

Furthermore, shared-memory architectures can be classified as symmetric multiprocessors with uniform memory access (UMA) or as non-uniform memory access (NUMA) architectures based on memory access time [6, 7].

- **Uniform Memory Access (UMA)**

  This category is most known by Symmetric Multiprocessor (SMP) machines. SMPs have identical processors with equal access time to memory. Figure 1.1 demonstrates a typical UMA architecture.

- **Non-Uniform Memory Access (NUMA)**

  Typically, these are made by physically connecting two or more SMPs. This way, one SMP can access memory of another SMP directly. However, access time to all memories is not equal to all processors.

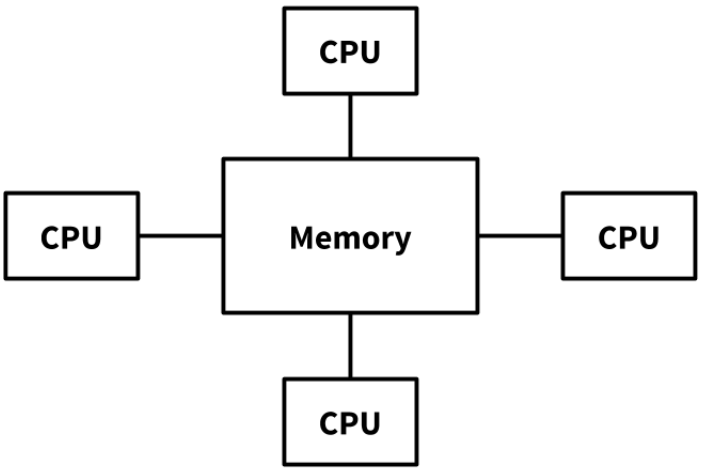

Figure 1.1: Shared Memory (UMA)

1.2.2 Distributed Memory

Each processing element has its own local memory. Therefore, there is no global address space. To connect inter-process memory units, a communication network is needed. Each processor operates independently. Changes that a processor makes to its local memory have no effect on the memory of other processors. If a processor needs to access data in another processor's memory, the programmer needs to explicitly define the communication mechanism.

1.2.3 Hybrid Distributed-Shared Memory

By combining a shared-memory and distributed-memory model, we will have a hybrid distributed-shared memory system. An example for this category could be a cluster of nodes with each node having multiple cores that share the same memory (Figure 1.4), or a cluster of nodes with each having multiple GPUs and cores (Figure 1.5).

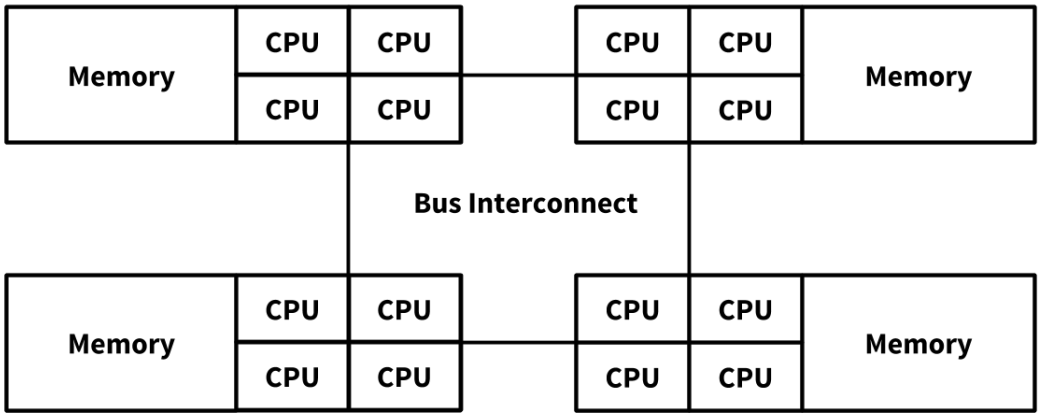

Figure 1.2: Shared Memory (NUMA)

1.3 Parallel Programming Models

The programming model is an abstraction that defines the hardware and the form of programming language or API used for writing parallel programs. Parallel programming models can be divided into two broad areas: Process interaction and problem decomposition. [8]

- **Process Interaction**

  Parallel processes need to communicate with each other. The mechanism that they use for the communication is described as process interation. Two main types of process interaction are shared memory and message passing. The interaction could also be implicit.

  - **Shared Memory**

    In this model, parallel processes share a global address space which they all read and write to. Since these operations are asynchronous, protection

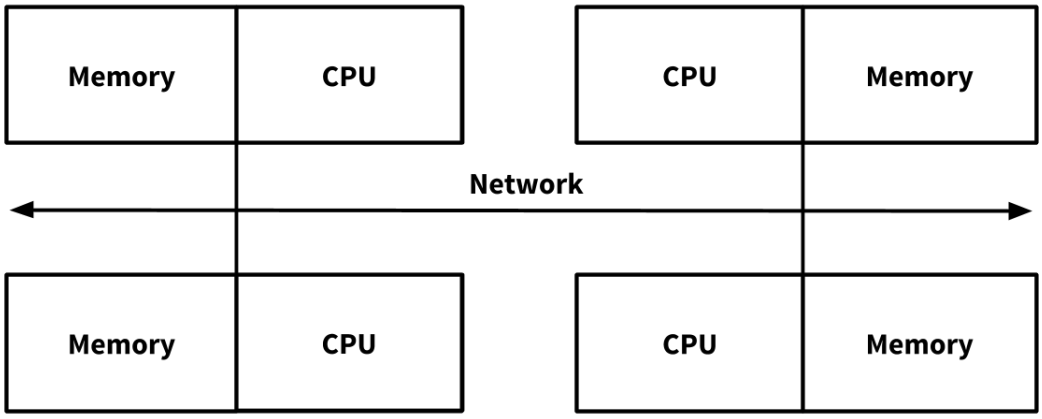

Figure 1.3: Distributed Memory

mechanisms like locks and semaphores are needed to control concurrent access.

- **Message Passing**

  In this model, parallel tasks exchange data through sending and receiving messages to and from one another. These communications can be synchronous or asynchronus.

- **Implicit**

  In this model, process interactions are not visible to the programmer. Usually, the compiler or the runtime is responsible for performing the interations.

- **Problem Decomposition**

  Problem decomposition describes the way processes are expressed to break down a large problem [9].

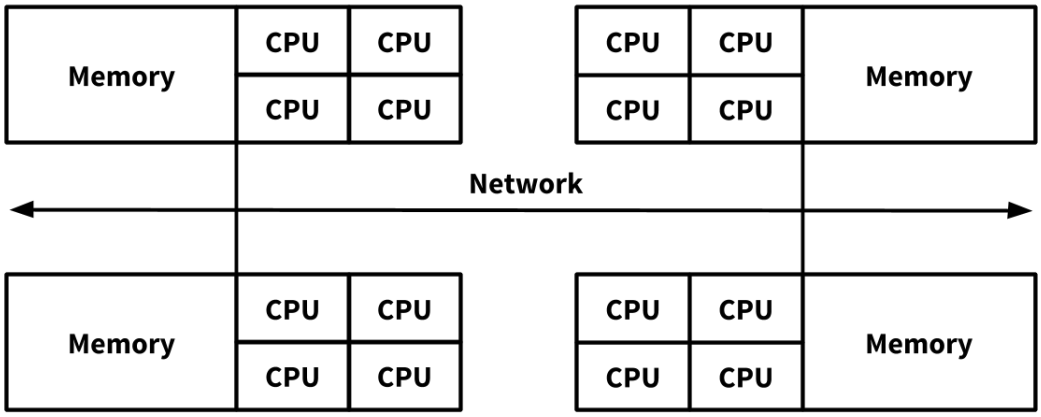

Figure 1.4: Hybrid Distributed-Shared Memory (Multi-core nodes)

– **Task Parallelism**

  The main focus of a task based parallelism is on processes or threads as individual units of execution. This is a natural way to express message passing communication.

– **Data Parallelism**

  In task-parallel models the data is usually structured in an array. A set of tasks will operate independently on separate partitions of the data.

1.4 Message Passing Interface (MPI)

Message Passing Interface (MPI) [10] is a language-independent communication application programming interface (API) used for programming in parallel environments. "MPI is a specification for a standard library for message passing that was defined by the MPI Forum, a broadly based group of parallel computer vendors, library writers, and applications specialists." [11] MPI has become a de-facto standard

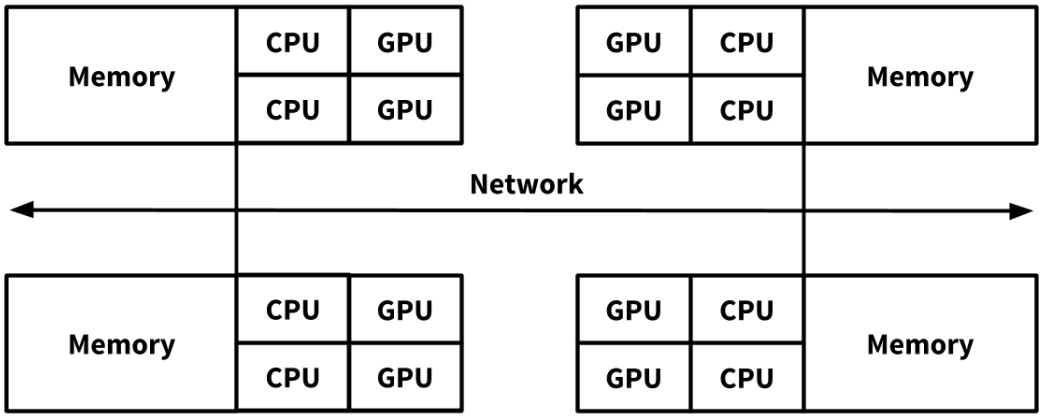

Figure 1.5: Hybrid Distributed-Shared Memory (Multi-core nodes with GPUs)

for communication among processes running on a distributed memory system.

Message Passing Interface aims to develop an efficient, portable, and flexible standard that will be widely used for writing message passing applications. MPI is the first standardized, vendor independent, message passing library. "MPI is not an IEEE or ISO standard, but has in fact, become the "industry standard" for writing message passing programs on HPC platforms." [12]

Some of the main concepts of MPI include [13–16]:

- **Communicators**

  A communicator can be thought of as a handle to a process group. A group is an ordered set of processes. Each process is associated with a rank. Ranks are contiguous and start from zero.

- **Point-to-point Communication**

  MPI includes routines for communication between two specific processes. For

instance, `MPI_Send` sends a message from one specified process to another specified process. MPI specifies mechanisms for both blocking and non-blocking point-to-point communication.

- **Collective Communication**

  All processes in a process group participate in a collective funtion. For example, `MPI_Bcast` sends data from one process to all the processes in a group.

- **One-sided Communication**

  MPI one-sided communication functions allow a process to access another process address space without explicit participation in that communication operation by the remote process. This can reduce synchronization and therefore improve performance in some cases.

- **Derived Datatypes**

  MPI provides a mechanism to create derived datatypes that are built from simple datatypes. These are useful in situations that the communication involves non-contiguous data or data that is comprised of multiple types.

- **MPI-IO**

  MPI-IO is a set of functions to carry out parallel I/O operations. These functions allow files to be easily accessed in a patterned way using the existing derived datatype functionality.

There are several implementations of MPI, including some that are in the public domain like MPICH [17] and Open MPI [18], and some commercial implementations from companies like HP, Intel, and Microsoft, and Fujitsu.

1.5 Runtime Environments and MPI Jobs

High-performance computing systems are growing toward hundreds-of-thousands to million-node machines, utilizing the computing power of billions of cores. Running parallel applications on such large machines efficiently will require optimized runtime environments that are scalable and resilient.

Considering a future where MPI remains a major programming paradigm, the MPI implementations will have to seamlessly adapt to launching and managing large scale applications on resources considerably larger than today's. [19]

1.6 Challenges for Runtime Environments

Multi- and many-core chip architectures in large-scale supercomputers pose several new challenges to designers of operating systems and runtime environments.
"Operating systems and runtime environments on supercomputers have similar goals: both seek to provide an environment for executing applications in a scalable and high-performing way. Achieving this goal often requires minimizing the layers of indirection between the application and the architecture." [20]

Developing a software environment to support high-performance computing applications in today's distributed systems poses a significant challenge. The runtime environment (RTE) must be capable of supporting heterogeneous operations, scale from one to large numbers of processors in an efficient manner, and provide strategies for dealing with fault scenarios that are expected of computing systems effectively.

Furthermore, the runtime must be easy to use, providing users with a transparent interface to the computing environment in a manner that avoids the need to customize applications when moving between specific computing resources. [21]

1.7 The Goal of this Thesis

ParalleX [22] is a new (and still experimental) general purpose parallel execution model aiming to overcome the limitations imposed by the current hardware and the way we write applications today. High Performance ParalleX (HPX) [23] is an experimental runtime system for ParalleX.

The majority of scientific and commercial applications in HPC are written in MPI. In order to facilitate the transition from MPI model to ParalleX, there is a need for a compatibility mechansim between the two. Currently, this mechanism does not exist.

The goal of this thesis is to provide a compatibility mechanism for MPI applications to use the HPX runtime system. This is achieved by developing a new runtime system for Open MPI. We call this new runtime system HPX-RTE.

1.8 Contributions

Design and implementation of HPX-RTE, the new runtime environment for Open MPI, is the main contribution of this thesis. HPX-RTE provides a compatibility layer for MPI applications to run on HPX runtime environment without any modification to the applications written in MPI. HPX-RTE facilitates the transition from MPI

to ParalleX programming model. HPX-RTE makes it possible for new programming paradigms such as MPI-HPX hybrid model to emerge. This thesis also presents a set of performance comparisons between HPX-RTE and ORTE (Open MPI's default runtime environment).

1.9 Organization of the Document

Chapter 2 provides a background and the state-of-the-art approaches to parallel runtime enviroment challenges. Open MPI as an implementation of choice will be introduced. We will discuss the project features and its architecture. Open Runtime Environment (ORTE), the current runtime environment for Open MPI project, and its features will be discussed in detail in secion 2.2. ParalleX and High-Perfomance ParalleX (HPX) will be presented in section 2.4. Chapter 3 covers the design principles, architecture, and implementation of our runtime environment for Open MPI (HPX-RTE) in sections 3.1, 3.2, and 3.4 accordingly. Chapter 4 is dedicated to evaluation of our developed runtime environment and its performance compared to the current runtime environment in Open MPI (ORTE). And finally, Chapter 5 concludes this document.

Chapter 2

Background

2.1 Open MPI

The Open MPI Project is an open source implementation of the Message-Passing Interface (MPI). Open MPI is developed by a consortium of academic, research, and industry partners to combine the expertise, technologies, and resources from the High Performance Computing community and build this MPI library.

Some of the Open MPI features are [24]:

- Open-source license
- Full conformance with MPI-3 standard
- Spawning processes dynamically
- Concurrency and thread safety
- Support for network heterogeneity

- Fault tolerance for network and processes
- Support for various job schedulers
- Support for all networks in a single library
- Portability and maintainability
- Run-time instrumentation
- Support for different operating systems (32 and 64 bit)
- Production-quality software
- Tunable by installers and end-users
- Modular component architecture
- Documentation for APIs

2.1.1 The Architectire of Open MPI

Open MPI is built based on a component architecture called the Modular Component Architecture (MCA). Component based archtirecture makes large software projects extensible and maintainable [25, 26]. It also allows users to build their own costumized components and integrate them into Open MPI. Component based architectures are popular in the high-performance computing community [27, 28].

Open MPI is comprised of three main functional areas (Figure 2.1) [26]:

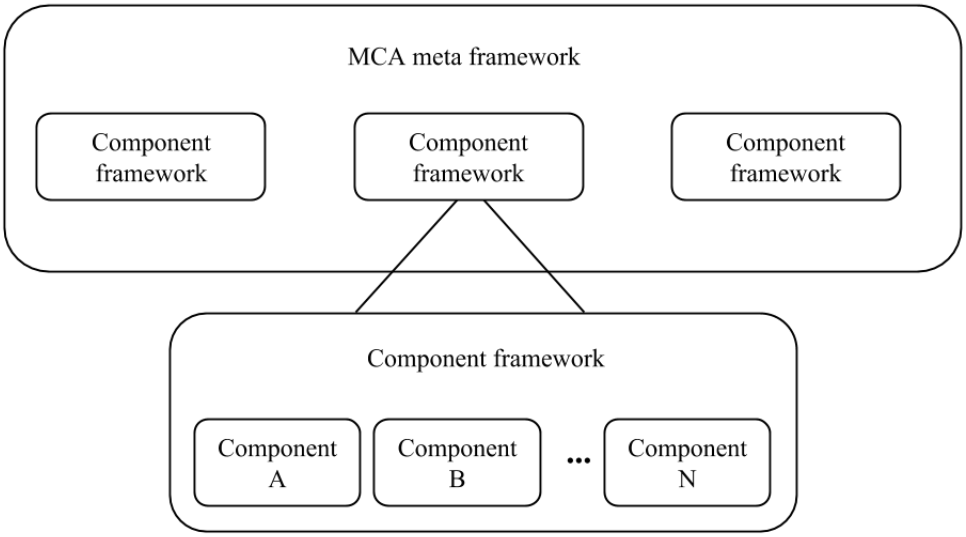

Figure 2.1: MCA, component frameworks, and the components

1. **MCA**

   The backbone modular component architecture that provides management services for all other layers.

   The MCA is responsible for management of the component frameworks and providing them services they use. For instance, the MCA provides the ability to accept run time parameters from higher level abstractions (e.g., mpirun) and pass them down through the component framework to individual components. It also finds components at build time and invokes their corresponding hooks for configuration, building, and installation.

2. **Component frameworks**

   Each major functional area in Open MPI has a corresponding back end component framework, which manages modules.

Each component framework is a construct that is created for a single, targeted task. For example, **btl** (Byte Transfer Layer) framework is used to send and receive data on different types of networks, **allocator** framework is responsible for memory allocation, and **coll** framework is dedicated to MPI collective algorithms. A framework uses MCA's services to discover, load, use, and unload components at run time. Each framework has different policies and use cases; some only use one component at a time while others use all available components simultaneously.

3. **Components**

   Components are self-contained software units that can configure, build, and install themselves. A component is an implementation of a framework's interface. Components are also known as "plugins". Each instance of a component is called a "module".

   The Open MPI software has three classes of components: Open MPI (OMPI) components, Open Runtime Environment (ORTE) components, and Open Portable Access Layer (OPAL) components. (Figure 2.2)

   Frameworks, components, and modules can be either dynamic or static. This means, they can be available as plugins or they can be compiled statically into libraries.

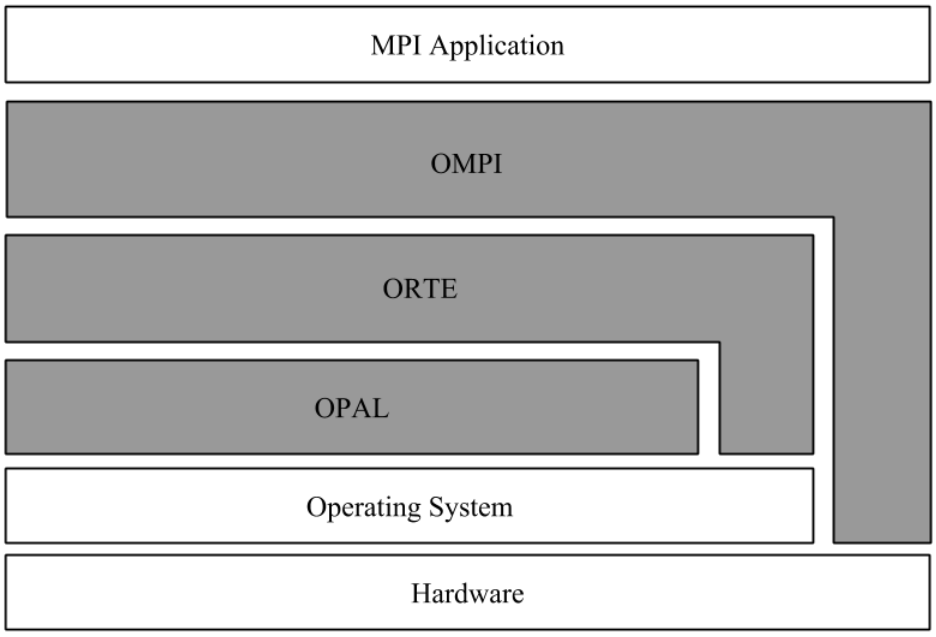

Figure 2.2: Open MPI Layers

2.2 Open Runtime Environment (ORTE)

Developing software environments for high-performance computing applications in heterogenous distributed systems poses a significant challenge. The runtime environment (RTE) must be capable of supporting heterogeneous operations, efficiently scaling from one to large numbers of processors, and providing effective strategies for dealing with fault scenarios that are expected as our systems continue to scale to exaflop systems [29].

There has been a number of studies with different approaches to this challenge. Each approach focuses on a particular aspect of the overall problem. For example,

LAM/MPI emphasis was on ease of portability and performance [30]. LA-MPI focused on data fault tolerance [31], and HARNESS FT-MPI focused on system fault tolerance [32].

The major roles of every runtime are [19]:

1. **Launch**

   This task to launch the processes for an MPI application. This is shared between the runtime and the parallel scheduling/launching mechanism.

2. **Connect**

   The connection information (the URI of a process) may not be known before the MPI processes are launched. Therefore, it is necessary for the runtime to establish the connections between the processes. It is then necessary to distribute this information through an out of band messaging system.

3. **Control**

   The control role is to ensure that the entire environment is gracefully cleaned in case of a crash. Depending on the operating system and implementation, control may also forward signals to the MPI processes and ensure that completion codes are returned to the user command: mpirun.

4. **IO**

   The input/output commands launched do not necessarily run on the same machine as where they are issued in an MPI application. However, users usually expect the information printed on the standard output appear on the standard output of the command they launched. Therefore, it is necessary to forward

the standard input/output information to the machine users launch their application from.

The Open Runtime Environment (ORTE) was developed as a part of the Open MPI project to support distributed high performance computing applications operating in a heterogeneous environment. Implementation of the ORTE is based on the Modular Component Architecture (MCA). The main design objectives of the ORTE are ease of use, resilient operations, scalability, and extensibility. Interprocess communication, resource discovery and allocation, and process launch across different platforms in a transparent manner are main features of the ORTE[33, 34].

The ORTE consists of four major subsystems (Figure 2.3)[21, 33, 35]:

1. **General Purpose Registry (GPR)**

   The GPR is the core subsystem in the ORTE architecture. It provides a mechanism for exchanging of communication connection data, in the form of key-value pairs, among processes. The GPR is also used to synchronize events across the system. It asynchronously notifies subscribers of events such as data changes in the registry or new data being entered to the registry.

2. **Resource Management**

   The resource management subsystem consists of four smaller subsystems: *Resource Discovery Subsytem (RDS)*, *Resource Allocation Subsystem (RAS)*, *Resource Mapping Subsystem*, and *Process Launch Subsystem (PLS)*. These four subsystems together provide services for resource discovery, allocation, mapping, and process launch.

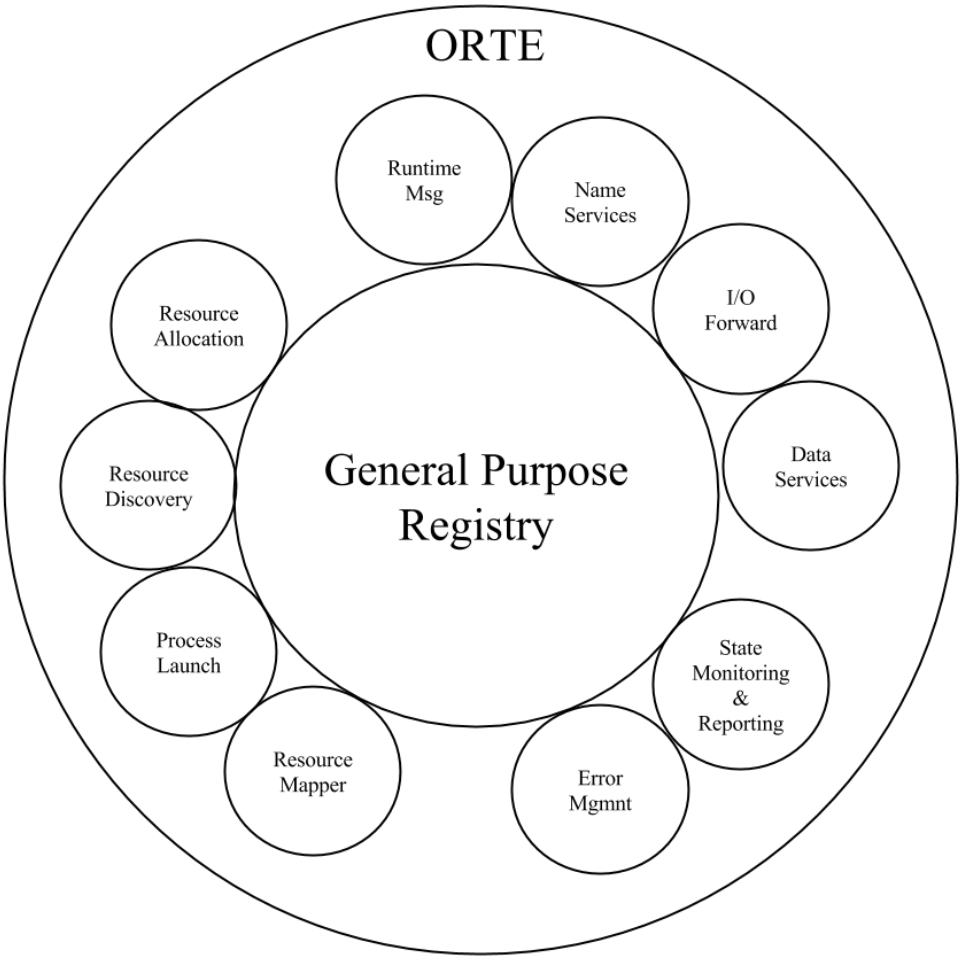

Figure 2.3: The ORTE architecture

3. **Error Management**

   The *State Monitoring and Reporting (SMR)* subsystem and the *Error Manager* subsystem are two smaller subsystems constructing the error management subsystem. The error management subsystem brings the fault tolerance capability to the ORTE.

4. **Support Services**

There are four subsystems comprising the support-services subsystem: The *Runtime Messaging Layer (RML)* is responsible for providing administrative communication services across all ORTE subsystems. The *Name Services (NS)* subsystem assigns each application, and each process within each application, a unique identifier. The *I/O Forwarding (IOF)* subsystem is responsible for transporting standard input, output, and error messages between the remote processes and the user. And finally, the *Data Services (DS)* subsystem is responsible for facilities like providing a single interface for all declared data types, packing/unpacking network communications, and support for transparent data manipulation within the ORTE.

2.3 ParalleX

ParalleX [36] is a new computation model that attempts to address the underlying sources of performance degradation [37]:

1. **Starvation**

   Starvation is the phenomenon of resourses being idle performing no useful action because the amount of concurrent work available to the resources is insufficient to utilize all of them.

2. **Latency**

   Latency is the amount of time a message takes to traverse a system. Accessing remote resources imposes a minimum delay equivalent to latency.

3. **Overhead**

Management of parallel resources requires extra work that is not necessary in the case of utilizing sequential resources.

4. **Waiting**

   In systems with shared resources, the possibility of contention exists. When there is such a contention, it needs to be resolved. Hence, there is a delay due to waiting for contention resolution.

ParalleX also tries to address the difficulties of programmer productivity like explicit locality management and scheduling, performance tuning, fragmented memory, and synchronous global barriers to dramatically enhance the broad effectiveness of parallel processing for high end computing [22]. ParalleX changes the fundamental model of parallel computation from communicating sequential processes (e.g., MPI) to an innovative combination of concepts using message-driven work-queue execution in the context of a global address space. [38, 39]

Main components of ParalleX include [36, 40, 41]:

1. **Active Global Address Space (AGAS)**

   While avoiding the overhead of cache coherence, the AGAS extends the PGAS [42] models (GASNet [43], UPC [44]) by allowing the dynamic migration of first class objects across the physical system without having to change the object's name. This facilitates load balancing by allowing work to be migrated from heavily loaded nodes to less loaded nodes.

2. **Parallel Processes**

   Unlike conventional models, ParalleX processes span over multiple nodes and

share nodes as well. A ParalleX process can define a name space shared across several localities supporting many concurrent threads and child processes.

3. **Threads**

   ParalleX threads provide local control flow and data usage within a single node utilized for specifying and performing most of the computational work to be performed by an application program. ParalleX threads can migrate to remote localities.

4. **Local Control Objects (LCOs)**

   LCOs provide different functionalities for event driven ParalleX thread creation, protection of data structures from race conditions, and scheduling of work automatically to let every single computation strand proceed as far as possible.

5. **Parcels**

   Parcels are messages that carry action and data asynchronously between different localities. "Parcels enable message passing for distributed control flow and dynamic resource management, implementing a split phase transaction based execution model." [36]

6. **Percolation**

   Percolation is a technique for using resources by moving the work to the resource while both hiding the latency of such action and eliminating the overhead of such action from the target resource.

2.4 High-Performance ParalleX (HPX)

High-Performance ParalleX (HPX) is the first open source general purpose C++ runtime system implementation for the ParalleX execution model [45, 46].

HPX, like many recent programming models is based on lightweight tasks. Task based parallel programming models can be divided into three major categories [46, 47]:

- **Libraries**

  Intel TBB [48], Qthreads [49], and StarPU [50] are some known examples for library solutions.

- **Language Extensions**

  OpenMP [51] and Intel Cilk Plus [52] are examples of language extensions.

- **Experimental Programming Languages**

  Chapel [53], X10 [54], and Intel ISPC [54] are notable examples in this category.

The majority of the previously mentioned task-based programming models focus on node-level parallelism. Providing a solution for homogeneous execution of local and remote operations is what distinguishes HPX from those models.

"HPX represents an innovative mixture of a global system-wide address space, fine grain parallelism, and lightweight synchronization combined with implicit, work queue based, message driven computation, full semantic equivalence of local and remote execution, and explicit support for hardware accelerators through percolation." [46]

2.4.1 HPX Design Principles

HPX follows a set of design principles that have been around for years. However, HPX gathers all these principles into a unified system [46].

1. **Latency Hiding instead of Latency Avoidance**

   It is impossible to have no latency in a system. However, to hide the latency, some unrelated useful work can be done during that time. This is one of the main concepts integrated into the design of HPX.

2. **Fine-grained Parallelism instead of Heavyweight Threads**

   To hide latencies for very short operations, low overhead of context swithing is a must. The overall system utilization improves by smaller overhead of a context switch and finer granularity of the threading system.

3. **Constrained Based Synchronization to Replace Global Barriers**

   Having all threads or processes wait for a particular operation to be finished is waste of resources that could otherwise be utilized. Replacing such barriers with constraint based synchronization through dataflow techniques could improve the overall performance of a system.

4. **Adaptive Locality Control instead of Static Data Distribution**

   MPI leaves the responsibility of data distribution to the programmer. The easy approach for most programmers is to distribute the data statically. PGAS systems rely on static data distribution as well. When there is an imbalance in the workload, this becomes a problem. In such cases, migrating part of the

application data to different localities (nodes) by the runtime system could increase overall utilization and make programmers job easier.

5. **Moving Work to the Data instead of Moving Data to the Work**

   It is obvious that the amount of data for a particular operation is very often much smaller than the amount of data the operation is performed on. Although this is possible using MPI, it is not deeply built into MPI model. Having a system that could provide this capability would reduce overhead of unnecessary data movement.

6. **Message Driven Computation instead of Message Passing**

   In message passing models like MPI, the receiver of a message needs to spend time waiting for the incoming messages. However, message driven computation allows sending messages without the receiver actively waiting for them. Incoming message are handled asynchronously. This allows the overlap of communication with useful work.

2.4.2 HPX Implementation Features

The implementation of HPX is feature complete. It supports all the main components of ParalleX (Figure 2.4) [55].

HPX has a modular architecture. This allows simple compile-time customization and minimizes the runtime memory footprint. Similar to the implementation of Open MPI, HPX enables dynamically loaded modules to extend the available functionality at runtime. Modules could be statically binded at link time as well.

The API exposed by HPX is aligned with the latest C++11 Standard [56] and

the C++14 Standard [57] as much as possible and utilizes Boost [58] C++ libraries to facilitate distributed operations, enable fine-grained constraint-based parallelism, and support runtime adaptive resource management[46].

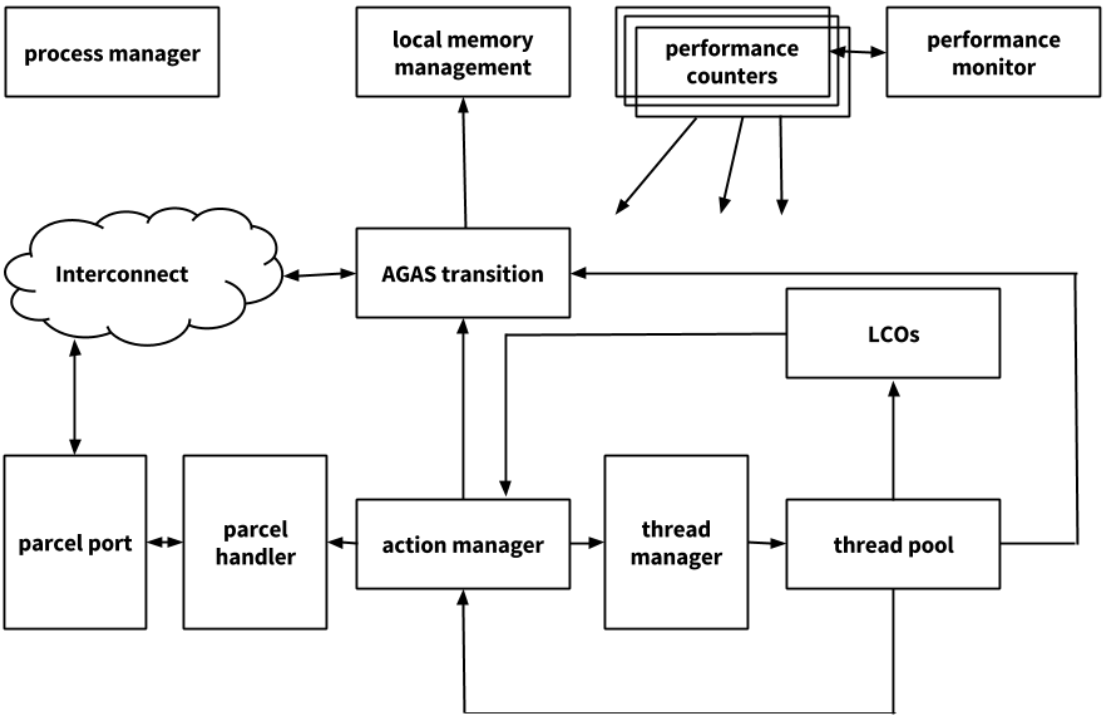

Figure 2.4: Modular Structure of HPX Implementation

Actions in HPX are special types used to describe possibly remote operations. "Applying the action" is the process of invoking a global function (or a member function of an object) with the help of the associated action. Actions can have arguments, which will be supplied while the action is applied. At least one parameter is required to apply an action: the id of the locality the associated function should be invoked on for global functions, or the id of the component instance for member

functions [59].

Figure 2.5 shows the function invocation syntax as defined by the C++ language (dark blue), the additional invocation syntax as provided through C++ Standard Library features (medium blue), and the extensions added by HPX (light blue)[46].

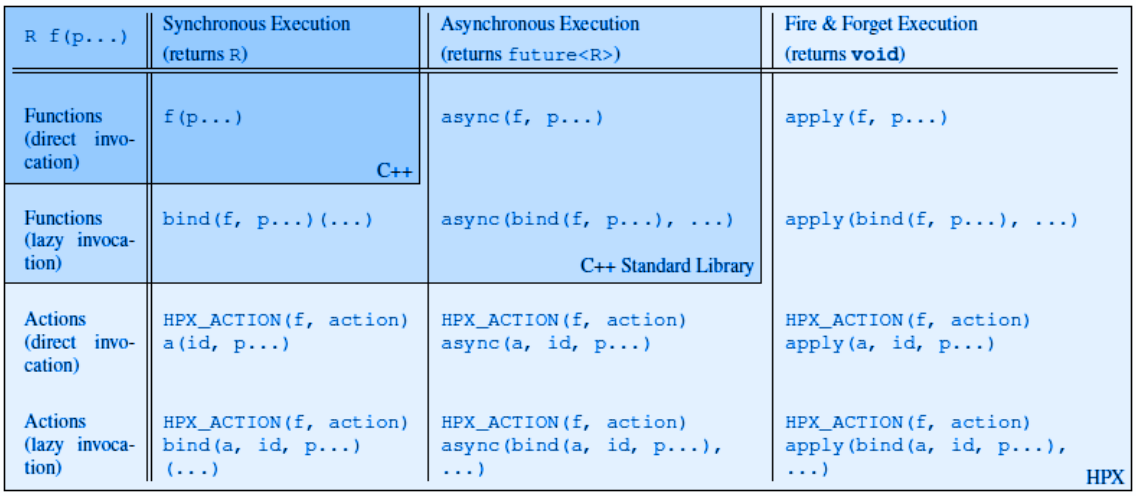

Figure 2.5: Overview of the main API exposed by HPX

HPX provides several ways to apply an action. Its API exposes three different ways of executing a function, locally on the same physical locality as the invocation site or remotely on a different locality:

- "**Synchronous function execution**

  This is the most natural way of invoking a C++ function. The caller waits for the function to return, possibly providing the result of the function execution. In HPX, synchronously executing an action suspends the current thread relinquishing the processing unit for other available work. Once the function is executed, the current thread is rescheduled.

- **Asynchronous function execution**

  Asynchronous invocation of a function means that it will be scheduled as a new HPX thread (either locally or on another locality). The call to async will return almost immediately providing a new future instance which represents the result of the function execution. Asynchronous function execution is the fundamental way of orchestrating asynchronous parallelism in HPX.

- **Fire & Forget function execution**

  This is similar to asynchronous execution except that the caller has no means of synchronizing with the result of the operation. The call to apply schedules a local (or remote) HPX thread which runs to completion at its own pace. Any result returned from that function (or any exception thrown) is being ignored. This leads to less communication by not having to notify the caller."[46]

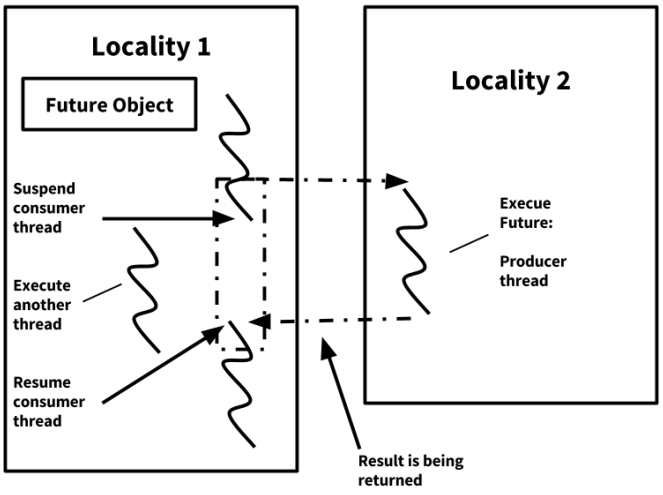

Figure 2.6: Schematic of a Future Execution

In C++, a future encapsulates a delayed computation. A future holds the result

of an asynchronous call because the computation of the result has not completed yet [60]. HPX uses futures to synchronize the access to the value in a future by suspending any HPX threads requesting the result until the value is available. When a future is created, it spawns a new HPX thread which will execute the action associated with the future. The new thread is spawned either remotely with a parcel or locally by being placed into the thread queue. Utilizing futuers allows HPX to schedule work early in a program so that when the function value is needed it will already be calculated and available (Figure 2.6) [36].

2.4.3 HPX Hello World Example

Listing 2.1 [61] is a simple program that prints out a hello world message on every OS-thread on every locality.

Listing 2.1: HPX Hello World

```
1  int main() {
2    std::vector<hpx::naming::id_type> localities =
3    hpx::find_all_localities();
4    std::vector<hpx::lcos::future<void> > futures;
5    futures.reserve(localities.size());
6    for (hpx::naming::id_type const& node : localities) {
7      typedef hello_world_foreman_action action_type;
8      futures.push_back(hpx::async<action_type>(node));
9    }
10   hpx::wait_all(futures);
11   return 0;
12 }
```

Using the `hpx::find_all_localities()` function, we get a list of all available localities and put them into a vector. Then, we reserve storage space for futures, one for each locality. Looping through all the localities, we asynchronously start a new task on each locality. The task is encapsulated in a future, which can be queried to determine if the task has completed. `hpx::wait_all()` returns when all of the futures have finished.

Listing 2.2 [61] illustrates the `hello_world_foreman()` function which is used to make `hello_world_foreman` action. `hpx::get_os_thread_count()` returns the number of worker OS-threads in use by the current locality. `hpx::find_here()` returns the global name of the current locality. Inside the for loop, we populate a set with the OS-thread numbers of all OS-threads on this locality. When the hello world is printed on a particular OS-thread, we remove it from the set. As long as there are elements in the set, we keep scheduling HPX-threads. Note that because HPX features work-stealing task schedulers, we have no way of enforcing which worker OS-thread will actually execute each HPX-thread. Inside the while loop, in each iteration we create a task for each element in the set of OS-threads that have not said "Hello world". Each of these tasks is encapsulated in a future. Eventually, we wait for all of the futures to finish. The `hpx::lcos::wait_each` function takes two arguments: a binary callback, and a vector of futures. The callback takes two arguments; the first is the index of the future in the vector, and the second is the return value of the future. `hpx::lcos::wait_each` doesn't return until all the futures in the vector have returned. The macro `HPX_PLAIN_ACTION` defines the boilerplate code necessary for the function `hello_world_foreman` to be invoked as an HPX action.

Listing 2.2: Hello World Foreman

```
1  void hello_world_foreman() {
2    std::size_t const os_threads = hpx::get_os_thread_count();
3    hpx::naming::id_type const here = hpx::find_here();
4    std::set<std::size_t> attendance;
5    for (std::size_t os_thread = 0; os_thread < os_threads; ++os_thread) {
6      attendance.insert(os_thread);
7    }
8    while (!attendance.empty()) {
9      std::vector<hpx::lcos::future<std::size_t> > futures;
10     futures.reserve(attendance.size());
11     for (std::size_t worker : attendance) {
12       typedef hello_world_worker_action action_type;
13       futures.push_back(hpx::async<action_type>(here, worker));
14     }
15     hpx::lcos::local::spinlock mtx;
16     hpx::lcos::wait_each(
17     hpx::util::unwrapped([&](std::size_t t) {
18       if (std::size_t(-1) != t) {
19         boost::lock_guard<hpx::lcos::local::spinlock> lk(mtx);
20         attendance.erase(t);
21       }
22     }),
23     futures);
24   }
25 }
26 HPX_PLAIN_ACTION(hello_world_foreman, hello_world_foreman_action);
```

Listing 2.3 shows the `hello_world_worker` function. This function is used to create `hello_world_worker_action`. The function `hpx::get_worker_thread_num()` returns the OS-thread number of the worker that is running this HPX-thread. If the HPX-thread is the one running on the desired OS-thread, we print the hello world message and flush the output stream. Otherwise, we don't do anything here and foreman action reschedules the action again until it is on the desired OS-thread.

Listing 2.3: Hello World Worker

```
1  std::size_t hello_world_worker(std::size_t desired) {
2    std::size_t current = hpx::get_worker_thread_num();
3    if (current == desired) {
4      char const* msg =
5      ``hello world from OS-thread %1% on locality %2%\n'';
6      hpx::cout << (boost::format(msg) % desired % hpx::get_locality_id())
7                << hpx::flush;
8      return desired;
9    }
10   return std::size_t(-1);
11 }
12 HPX_PLAIN_ACTION(hello_world_worker, hello_world_worker_action);
```

Chapter 3

HPX-RTE

ParalleX and HPX are both parts of the eXascale Programming Environment and System Software (XPRESS) [45, 62] project funded by the Department of Energy (DOE).

The goals of XPRESS project are [63]:

- Enable exascale performance capability for Department of Energy applications
- Develop a software stack, "OpenX", for future Department of Energy computing systems
- Provide programming models, languages, environments, and tools for expressing system and application software for exascale

This project is a collaboration between Sandia National Laboratories (SNL), Indiana University (IU), Lawrence Berkeley National Laboratory (LBNL), Louisiana State University (LSU), Oak Ridge National Laboratory (ORNL), University of

Houston (UH), University of North Carolina at Chapel Hill/RENCI (UNC/RENCI), and University of Oregon (UO).

Figure 3.1 illustrates the OpenX software architecture. Support for legacy applications, and specifically support for MPI applications is this thesis' target.

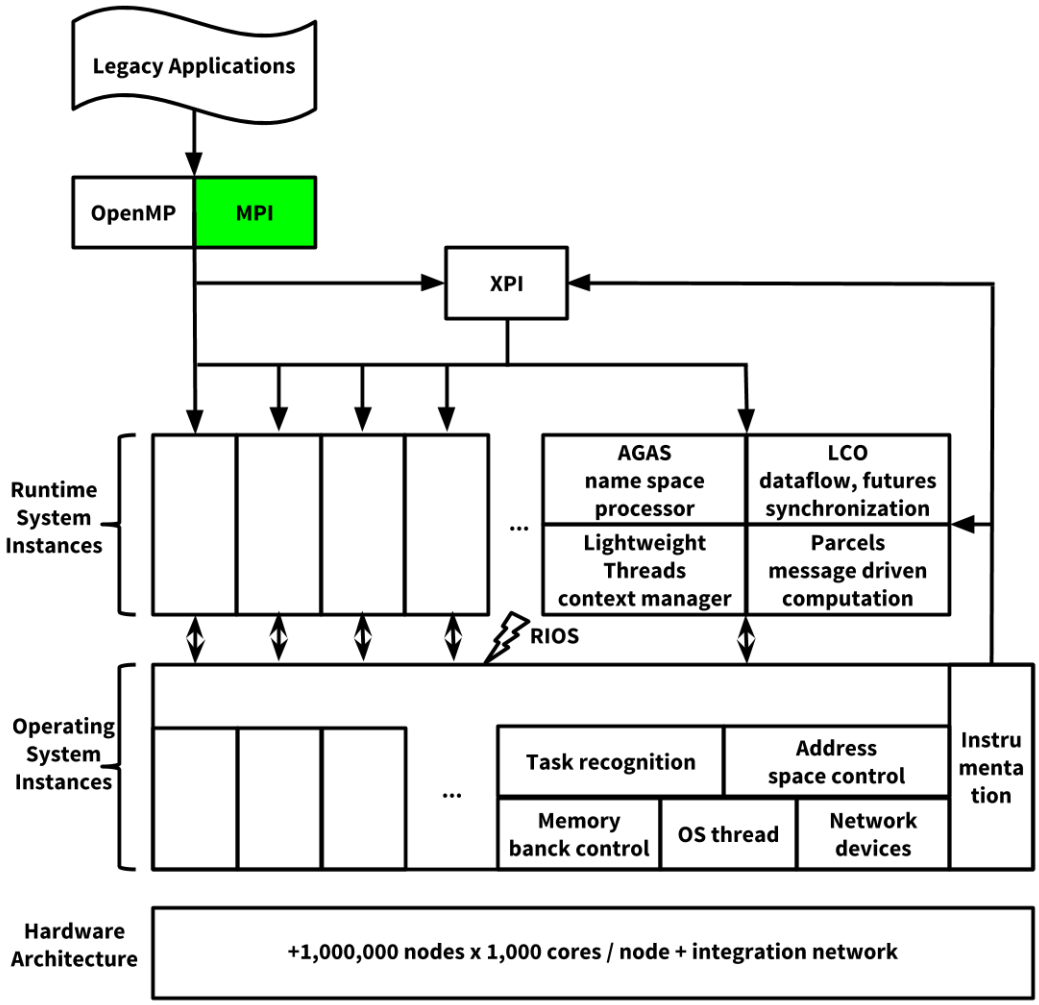

Figure 3.1: The OpenX Software Architecture

3.1 Design Principles

The main goal of this project that led to the development of HPX-RTE was to provide facilities for MPI applications to compile and run the exact same way in exascale software stack environment and be compatible with HPX runtime environment. This goal is achieved by replacing the current runtime environment (ORTE) with a new runtime environment developed from scratch to take advantage of the API provided by HPX. This choice was made by having a set of design principles in mind:

- **Modularity**

  Utilizing the modular structure of the Open MPI project, there is a framework dedicated to the runtime environment (rte) in OMPI layer. This framework is designed to provide the interfce necessary for different runtime environments (Figure 3.2). This allows different runtime environments to coexist independently inside the Open MPI project and be chosen by users based on their environment and application needs and priorities.

- **Functionality**

  The main focus of this work is to have the MPI applications work on HPX infrastructure. To do this, most of the effort is put on having the functionality in place before tackling any possible consequences such as its effect on performance, efficiency, or power consumption. Although every possible consideration has been taken into account not to introduce sources of performance degradation, such potential side effects could still happen. Those may be the topic for other studies after having the new runtime environment in place and

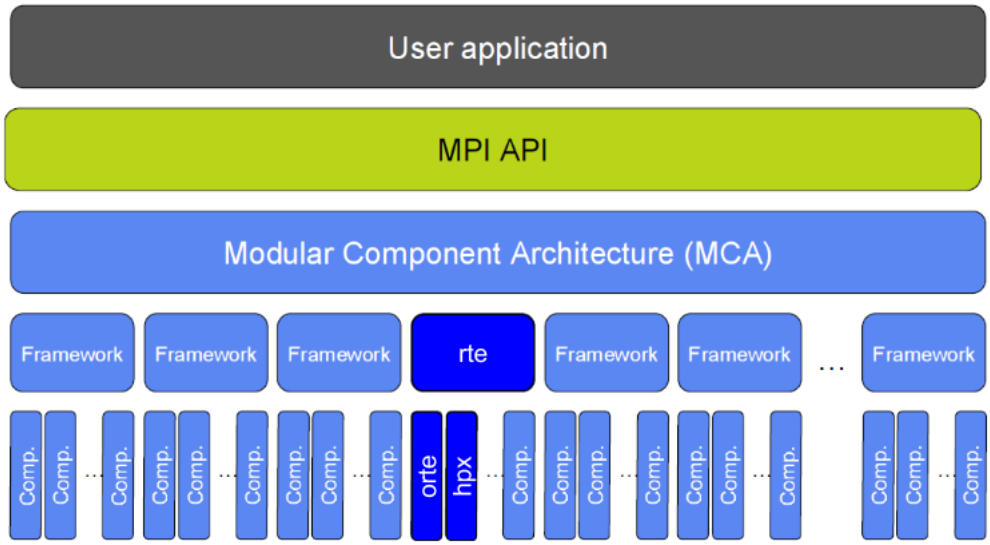

Figure 3.2: Modular Component Architecure with rte Framework

functional.

- **Simplicity**

  Simplicity is a key factor is the design of HPX-RTE. We have avoided introducing any unnecessary algorithm or functionality. Implementing the required API, we have tried to keep implementation as simple and straightforward as possible.

- **Code Reuse**

  HPX-RTE relies on a number of advanced features and concepts provided by the HPX API. Therefore, there was no need to reimplement what was already done. We have tried to take those features to a new level by integrating them tightly into the implementation of HPX-RTE.

3.2 Architecture

HPX-RTE is designed to replace the functionality of current Open MPI runtime (ORTE). Figure 3.3 illustates the logical layers of Open MPI software using HPX-RTE as its runtime environment. The modular component architecture of Open MPI facilitates the implementation of this design.

The required functinality for HPX-RTE is implemented as a component of the runtime environment (rte) framework (Figure 3.2) in OMPI layer. This design dictates a number of changes outside the rte framework. We will discuss those changes in the implementation section.

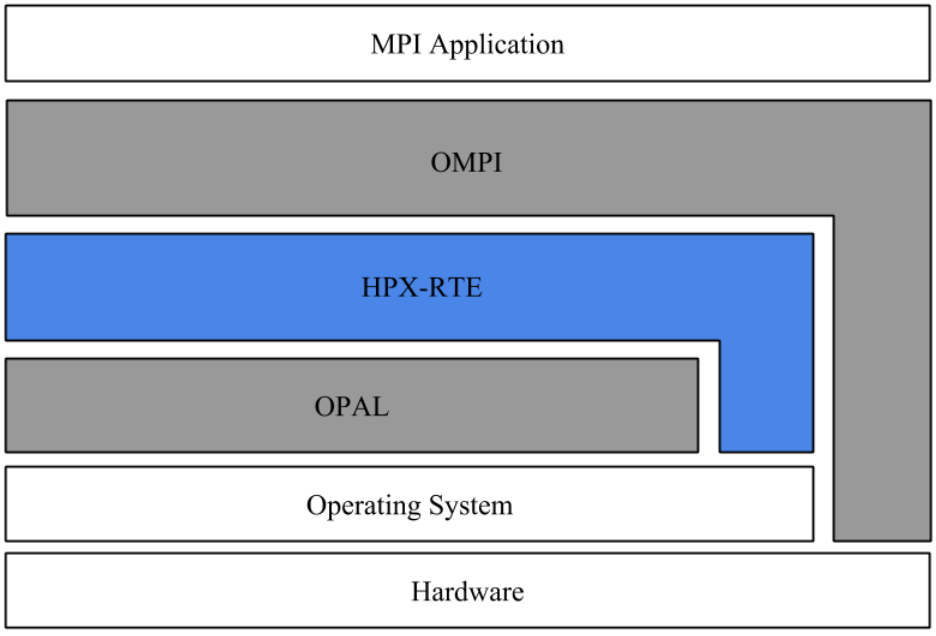

Figure 3.3: Open MPI Layers using HPX-RTE

3.3 Runtime Environement Requirements

The rte framework inside OMPI layer of Open MPI defines a required set of data structures and functions that every runtime environment component needs to provide implementation for [18]:

3.3.1 Process Name Objects and Operations

1. `ompi_jobid_t` **and** `ompi_vpid_t`

   These two need to be defined as integer types. The jobid must be unique for a given `MPI_COMM_WORLD` capable of connecting to another `OMPI_COMM_WORLD` and the vpid will be the process's rank in `MPI_COMM_WORLD`.

2. `ompi_process_name_t`

   This is a struct that must contain at least two fields of type integer:

   (a) `ompi_jobid_t` jobid

   (b) `ompi_vpid_t` vpid

3. `OMPI_NAME_PRINT`

   When given a pointer to `ompi_process_name_t`, this macro has to print a process name. The output format has to be a single string representing the name. This function should be thread-safe for multiple threads to call simultaneously.

4. `OMPI_PROC_MY_NAME`

   A pointer to a global variable containing the `ompi_process_name_t` for this process.

5. `OMPI_NAME_WILDCARD`

   A wildcard name.

6. `ompi_rte_compare_name_fields`

   A function used to compare fields in the `ompi_process_name_t` struct. The function prototype must be of the form:

```
int ompi_rte_compare_name_fields(
        ompi_rte_cmp_bitmask_t mask,
        ompi_process_name_t *name1,
        ompi_process_name_t *name2);
```

   The bitmask must be defined to indicate the fields to be used in the comparison. Fields not included in the mask must be ignored. Supported bitmask values must include:

   (a) `OMPI_RTE_CMP_JOBID`

   (b) `OMPI_RTE_CMP_VPID`

   (c) `OMPI_RTE_CMP_ALL`

7. `uint64_t ompi_rte_hash_name`**(name)**

   Return a string hash uniquely representing the `ompi_process_name` passed in.

8. `OMPI_NAME`

   An OPAL Data Packing Subsystem(DSS) constant for a handler already registered to serialize/deserialize an `ompi_process_name_t` structure.

3.3.2 Collective Objects and Operations

1. `ompi_rte_collective_t`

   An OPAL object used during the runtime environment collective operations such as modex and barrier. This must be of type `opal_list_item_t` and contain the following fields:

   (a) `int32_t` id

   (b) `bool` active: A flag that user can poll on to know when collective operation has completed. If a user callback function is provided, this needs to be set to false just prior to calling it.

2. `ompi_rte_modex`

   A function that performs an exchange of endpoint information to wire up the MPI transports. The function prototype must be of the form:

   ```
   int ompi_rte_modex(ompi_rte_collective_t *coll);
   ```

   At the completion of the modex operation, the `coll->active` flag must be set to false, and the endpoint information must be stored in the modex database. This function must have barrier semantics across the `MPI_COMM_WORLD` of the calling process.

3. `ompi_rte_barrier`

   A function that performs a barrier operation within the RTE. The function prototype must be of the form:

   ```
   int ompi_rte_barrier(ompi_rte_collective_t *coll);
   ```

At the completion of the barrier operation, the `coll->active` flag must be set to false.

3.3.3 Process Information Structure

1. `ompi_process_info_t` A structure containing information about the current process. The following fields are mandatory within the struct:

   (a) `app_num`

   (b) `pid`

   The current process's id. This should be the same as the output of getpid() function.

   (c) `num_procs`

   Number of processes in this job (ie, `MPI_COMM_WORLD`).

   (d) `my_node_rank`

   Relative rank on local node to other peers this runtime instance knows about. In a static job this will be `my_local_rank`.

   (e) `my_local_rank`

   Relative rank of the process on the local node with other peers in this job (ie, `MPI_COMM_WORLD`).

   (f) `num_local_peers`

   Number of local peers (peers in `MPI_COMM_WORLD` on the same node).

   (g) `my_hnp_uri`

(h) `peer_modex`

This is a collective id for the modex operation.

(i) `peer_init_barrier`

A collective id for the barrier during `MPI_Init`.

(j) `peer_fini_barrier`

A collective id for the barrier during `MPI_Finalize`.

(k) `job_session_dir`

(l) `proc_session_dir`

(m) `nodename`

A string representation for the name of the node this process is located on.

(n) `cpuset`

(o) `ompi_process_info`

A global instance of the `ompi_process_t` structure.

2. `ompi_rte_proc_is_bound`

A global boolean that will be set to true if the runtime bound the process to a particular core or set of cores. Otherwise, it will be false.

3.3.4 Error-Handling Objects and Operations

1. `void ompi_rte_abort(int err_code, char *fmt, ...)`

Abort the current process with the specified error code and message.

2. `int ompi_rte_abort_peers(ompi_process_name_t *procs, size_t nprocs)`

   Abort the specified list of peers.

3. `OMPI_ERROR_LOG(rc)`

   This is a macro that prints the error message regarding the given return code.

4. `ompi_rte_register_errhandler`

   A function to register a callback function for the runtime environment to report asynchronous errors to the caller.

3.3.5 Initializing and Finalizing Objects and Operations

1. `int ompi_rte_init(int *argc, char ***argv);`

   This function initializes the runtime environment.

2. `int ompi_rte_finalize(void);`

   This function finalizes the runtime environment.

3. `void ompi_rte_wait_for_debugger(void);`

   This function is called during `MPI_Init`. It is used to wait for debuggers to do their pre-MPI attachment. This function will not block if no debugger is attached.

3.3.6 Database Operations

1. ```
   int ompi_rte_db_store(const ompi_process_name_t *proc,
                         const char *key,
                         const void *data,
                         opal_data_type_t type);
   ```

   This function is used to store modex and other data in a local database. It is primarily used for storing modex data. The implementation of this function must store a copy of the data provided. The data is not guaranteed to be valid after return from the call.

2. ```
   int ompi_rte_db_fetch(const struct ompi_proc_t *proc,
                         const char *key,
                         void **data,
                         opal_data_type_t type);
   ```

   This function is used to fetch modex and other data from the database. Fetch accepts an `ompi_proc_t`.

3. ```
   int ompi_rte_db_fetch_pointer(
                       const struct ompi_proc_t *proc,
                       const char *key,
                       void **data,
                       opal_data_type_t type);
   ```

4. Pre-defined database keys (with associated values after `rte_init`)
   ```

(a) `OMPI_DB_HOSTNAME`

(b) `OMPI_DB_LOCALITY`

3.4 Implementation

We have explained in previous sections that Open MPI has an abstraction layer for runtime environments. HPX-RTE provides an implementation of that abstraction layer using HPX features and facilities.

3.4.1 HPX-RTE Requirements

Since both Open MPI and HPX project are under active development, we decided to fix our implementation target to specific versions of these two software projects during the development cycle. HPX-RTE was built on a fork of Open MPI version 1.8 branch. We used the released HPX version 0.9.10. HPX library is dependent on the Boost libraries. Boost libraries version 1.55.0 were installed on the development machines. To summerize, users will need to have Open MPI 1.8, HPX 0.9.10, and Boost 1.55.0 as requirements to take advantage of HPX-RTE.

3.4.2 Features

Some of the main implementation features of HPX-RTE include:

1. **Distributed Database**

   For the runtime environment, we need to store a set of key-value pairs in a database accessible to all the localities participating in a job. Utilizing HPX

actions, HPX-RTE stores the data on local node at the time a store operation is issued. We have used singly linked lists to store the data. However, to retrieve the data the processes take advantage of HPX actions (remote function calls) to fetch the stored data from the database on the locality it is stored on. In theory, this database scheme eliminates the bottleneck of accessing a centralized database, or storing copies of all the database entries in every single locality. Therefore, it distributes the database, uses less space, and makes accessing the stored data faster. Listing 3.1 and 3.2 demononstrate two funtions (`hpx_rte_cpp_put` and `hpx_rte_cpp_get`) that store and fetch operations are built based on.

Listing 3.1: Put Functionality

```
int rte_hpx_cpp_put(char* key, int keysize,
                    char* val, int valsize)
{
  std::vector<char> vectorval (val, val + valsize);
  rte_hpx_put( std::string (key), vectorval );
  return 0;
}
```

2. **Barrier Semantics**

HPX strives not to have barriers that would stall all processes or threads for a particular operation to be completed. Instead, there are fine-grained mechanisms available for synchronization. However, a barrier (global synchronization point) is a requirement for the runtime component in Open MPI. We have developed an algorithm that implements barrier semantics by making effective

use of HPX actions and remote asynchronous function calls (Listing 3.3). We create a vector of futures with its size equal to the number of localities. Then, we asynchronously call an action to add one to a globally shared atomic counter on each locality. When this counter reaches the total number of localities, the synchronization is accomplished.

Listing 3.2: Get Functionality using HPX Asynchronous Actions

```
int rte_hpx_cpp_get( int vpid, char* key, char** value )
{
  hpx::naming::id_type const& node =
    rte_hpx_cpp_get_locality_from_vpid (vpid);
  std::string stringkey = key;
  int counter;
  *value = NULL;
  std::vector<char> temp_vector =
    hpx::async<rte_hpx_get_action>(node, stringkey).get();
  if (temp_vector.size() != 0) {
    char *temp_value = (char *) malloc (temp_vector.size());
    memcpy(temp_value, temp_vector.data(), temp_vector.size());
    *value = temp_value;
  }
  return (int) temp_vector.size();
}
```

3. **Mapping Localities to Ranks**

HPX uses the notion of locality. "A locality represents a set of hardware resources with bounded, finite latencies." [46] MPI uses ranks. We have provided

a translation table from HPX localities to mpi ranks.

4. **Populating Internal Data Structures**

   A number of internal data structures need to be initialized within the runtime environment before starting the MPI initialization. This is done in the runtime initialization function.

Listing 3.3: Barrier Implementation

```
void rte_hpx_barrier()
{
  std::vector<hpx::naming::id_type> localities =
  hpx::find_all_localities();
  std::vector<hpx::lcos::future<void> > futures;
  futures.reserve(localities.size());
  BOOST_FOREACH(hpx::naming::id_type const& node, localities) {
    futures.push_back(hpx::async<rte_hpx_one_action>(node));
  }
  hpx::wait_all(futures);
  while(sum != (int) localities.size()) {
  }
  sum = 0;
  return;
}
```

5. **Populate Key-Value Pairs in the Database**

   Since the defined abstraction layer is not implemented perfectly, there was a number of key-value pairs that were needed to be populated into the database. In theory, this shouldn't be the case. But this is a matter of implementation

practices. For instance, some of these key-value pairs are set in ORTE layer (which is supposedly an independent layer) and used in OMPI layer. Identifying and extracting those pairs was a challenging task.

6. **Intercepting C Standard Output**

   In order to see the output from C part of the code, we needed to intercept the printf function in C and send its output to C++ output stream. Listing 3.4 shows our implementation of print function. This technique can be used for any other standard output function in C.

Listing 3.4: Intercepting printf

```
int printf(const char *restrict format, ...)
{
  char      out[1024];
  va_list   args;
  va_start ( args, format );
  vsnprintf ( out, 1024, format, args );
  va_end    ( args );
  rte_hpx_cpp_printf(out);
  return (strlen(out));
}
```

7. **Automatic Boost and HPX Detection**

   Since Boost and HPX are required to be installed before the installation of Open MPI with HPX-RTE, we modified the configure logic of Open MPI to detect the installation paths of these libraries and accordingly modify the generated make files with appropriate compilation and linkage flags.

8. **C and C++ Compatibility**

The majority of the Open MPI source code is written in C language. HPX is completely written in C++ with extensive use of recent C++14 features. This definitely causes compatibility issues which needed to be handled in the code. For instance, Listing 3.5 illustrates an example on how different headers are made visible to either C or C++ compiler.

Listing 3.5: Exposing Different Headers to C and C++ Compilers

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

#if defined(__cplusplus)
#include <hpx/hpx_main.hpp>
#include <hpx/hpx_finalize.hpp>
#include <hpx/include/iostreams.hpp>
#endif
```

The same technique is applied for functions that are written in C or C++. We also utilized **extern "C"** to make function names in C++ have C linkage (compiler does not mangle the name) so that client C code can link to the functions using a C compatible header file that contains just the declaration of those functions. Listing 3.6 illustrates the technique. Note that this listing does not contain all the function prototypes in the source code.

We also had to use C++ to compile a number of components that were initially compiled by the C compiler in Open MPI.

Listing 3.6: Exposing Different Functions to C and C++ Compilers

```
#if defined(__cplusplus)
extern ''C'' {
  struct Node{
    struct Node *next;
    std::string key;
    std::vector<char> value;
    int originator;
  };
  typedef struct Node Node;
  int rte_hpx_vpid(void);
  int rte_hpx_num_localities(void);
  void rte_hpx_local_map(void);
  void rte_hpx_barrier(void);
}
#endif
```

9. **Communication Protocols**

   Current implementation of HPX-RTE supports Transmission Control Protocol(TCP) [64] and Infiniband [65] communication protocol.

Chapter 4

Evaluation

4.1 Test Environment

For the purpose of evaluation, the crill cluster at the Research and Computing Center of the University of Houston was used.

4.1.1 Crill Cluster Hardware Specification

- **16 NLE Systems nodes (crill-001 - crill-016)**

  Four 2.2 GHz 12-core AMD Opteron processors (48 cores total)

  64 GB main memory

  Two dual-port 4xDDR InfiniBand HCAs

- **2 Appro 1326G4 nodes (crill-101 - crill-102)**

  Two 2.2 GHz 12-core AMD Opteron processors (24 cores total)

  32 GB main memory

Four NVIDIA Tesla M2050 GPUs (448 cores each)

4xDDR InfiniBand HCA

- **4 HP DL 160 Gen 8 nodes (crill-200 - crill-203)**

  Two 2.4 GHz quad-core Intel Xeon E5-2665 processors (24 cores total)

  8 GB main memory

  4xDDR InfiniBand HCA

- **Network Interconnect**

  144 port 4xInfiniBand DDR Voltaire Grid Director ISR 2012 switch (shared with whale cluster)

  24 port 4xInfiniBand SDR switch (I/O switch to the SSD storage)

  48 port Netgear GE switch

- **Storage**

  2 TB RamSan 620 SSD storage (/pvfs2-ssd)

  20 TB Sun StorageTek 6140 array (/home shared with shark cluster)

  8 TB distributed PVFS2 storage (/pvfs2)

4.1.2 Crill Cluster Nodes Software Specification

- **Operating System**

  Linux kernel version 3.11.10-21-desktop

  Distribution: openSUSE 13.1 (x86_64)

- **Open MPI** version 1.8

- **HPX** version 0.9.10 release

- **Boost** version 1.55.0
- **GCC and G++** version 4.8.1

4.2 Configuration, Compilation, and Execution

4.2.1 Open MPI Configuration Parameteres

For running different test cases, our evaluation is mostly focused on comparison between Open MPI using ORTE vs. Open MPI using HPX-RTE as runtime. Listing 4.1 shows the configure line for the Open MPI installation. The only parameter that needs to be changed is `--with-orte`. When this parameter is set to "yes", Open MPI uses ORTE as runtime. If we set this parameter to "no", Open MPI will use HPX-RTE.

Listing 4.1: Configure Line of Open MPI with HPX-RTE

```
$ ./configure CFLAGS=``-g -O0'' CXXFLAGS=``-g -O0''
  --prefix=/home/hadi/opt/openmpi
  --with-hpx=/opt/hpx-0.9.10-release
  --with-boost=/opt/boost/1-55-0
  --disable-vt --enable-mca-no-build=coll-ml,vprotocol
  --enable-oshmem=no --enable-mpi-profile=no
  --enable-static --with-orte=no
```

4.2.2 Compiling MPI Applications

Listing 4.2 illustrates the compile line for our Hello World example. This line could be integrated into Open MPI software in future iterations of the development. Therefore, users will not have to manually insert all the compile and linkage flags needed. The same command can be used when ORTE is the installed runtime.

Listing 4.2: Compile Line for Hello World

```
$ mpicxx -o mpi_hello mpi_hello.c -rdynamic -fPIC -std=c++11
-Wall -Wno-unused-local-typedefs -Wno-strict-aliasing
-Wsign-promo -Wno-cast-align -Werror=vla -Werror=return-type
-fdiagnostics-show-option -Werror=uninitialized -pthread
-DHPX_DEBUG -D_GNU_SOURCE -I${HPX_DIR}/include/hpx/external
-I${HPX_DIR}/include -I/usr/include/google
-DHPX_APPLICATION_EXPORTS -DHPX_ENABLE_ASSERT_HANDLER
-DHPX_DEBUG -finline-functions -I/opt/boost/1-55-0/include
-L/opt/boost/1-55-0/lib -Wl,-rpath,:${HPX_DIR}/lib/hpx
-L${HPX_DIR}/lib/hpx -L${HPX_DIR}/lib -lhpx -lhpx_init
-lhpx_serialization -lboost_date_time -lboost_filesystem
-lboost_program_options -lboost_regex -lboost_serialization
-lboost_system -lboost_thread -lboost_atomic -lboost_chrono
-lprofiler -liostreams -L/lib64 -lrt -ldl -lutil -g -O0
```

4.2.3 Running MPI Applications

For running applications using ORTE, we will use “mpirun” command, which is a soft link to “orterun” command. This is shown in listing 4.3. ORTE also supports a direct launch mechanism which avoids creating the ORTE daemons on individual compute

nodes but uses the native resource manager instead for the management services, e.g. an application can be started directly using the srun command in a SLURM enviornment. This version has lower startup costs, but reduces the functionality of the runtime environment.

Listing 4.3: Running MPI Applications Using ORTE

```
$ mpirun -np 1 -pernode ./mpi_hello
```

HPX provides support for Slurm (Simple linux utility for resource management) [66]. To run applications using HPX-RTE, we use "srun" command. Listing 4.4 demonstrates an example.

Listing 4.4: Running MPI Applications Using HPX-RTE

```
$ srun -N 1  -n 1 --ntasks-per-node=1 ./mpi_hello --hpx:run-hpx-main
  --hpx:threads=2
```

4.3 Evaluation

In our evaluations, there are two ways to measure the time in the case of each application: The total time measured by the "time" command in Linux, and measuring the time within the application. The time reported by the application is reported on a per process basis in some test cases. Numbers reported in this section are averge numbers reported by the application when the application reported times are reported. Every test was run at lesat three times. The average of all three times is reported.

4.3.1 Parallel Computational Fluid Dynamics

We use an MPI implementation [67] of a computational fluid dynamics (CFD) test case defined by Ecer, Satofuka, Periaux, and Taylor, 2006 [68]. The objective is to solve the following partial differential equation (PDE) on a parallel computer:

$$\frac{df}{f} = \Delta f$$

This test case uses an explicit scheme with forward time and centered space. The equation is solved over a cube of unit dimensions. The initial conditions everywhere inside the cube are: f = 0. The boundary conditions are: f = x on all edges. The implementation calculates the steady-state solution.

Figure 4.1 shows the time spent running the application using ORTE and HPX-RTE. Figure 4.2 illustrates the times reported by the application using ORTE and HPX-RTE. We ran the application with different number of processes (1, 2, 4, 8, and 16). The grid dimensions are 100x100x100.

Both Figure 4.1 and Figure 4.2 demonstrate that despite slight better time performance using HPX-RTE in most cases, the overall times are very close. We believe this is because the overall time is dominated by communication and computation over the time spent by the runtime. To verify this, we decreased the problem size and used a grid of size 80x80x80. The results shown in Figure 4.3 and Figure 4.4 support this hypothesis. As we decrease the problem size, the time difference becomes more distinct.

To better understand the effect of the runtime, we can study an example with minimal communication and computation.

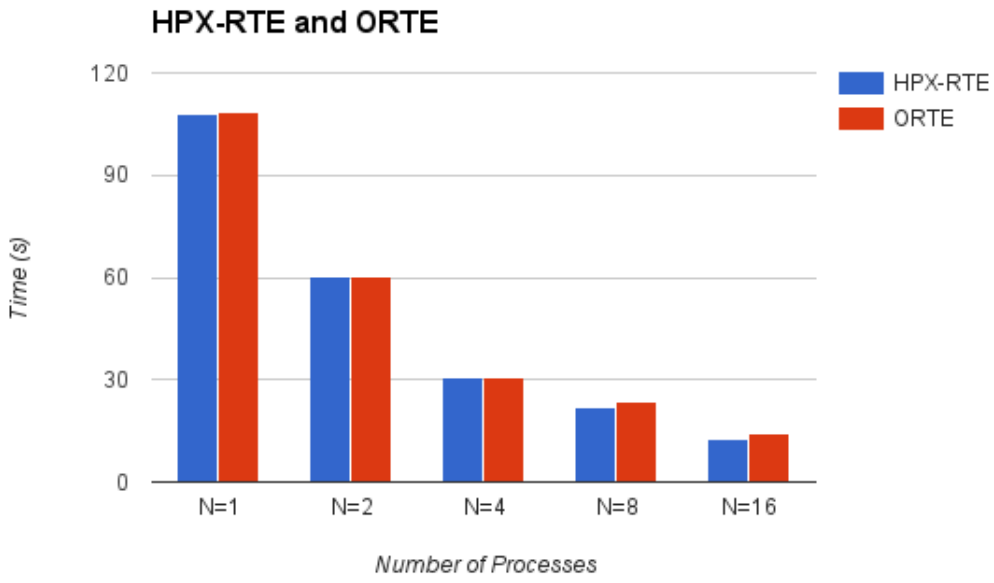

Figure 4.1: CFD Running Time - 100x100x100

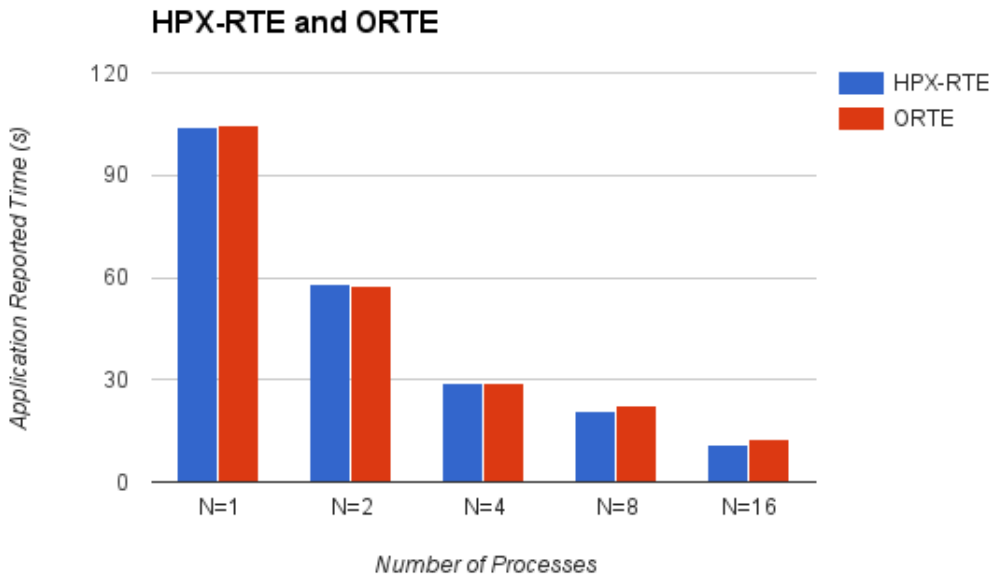

Figure 4.2: CFD Application Reported Time - 100x100x100

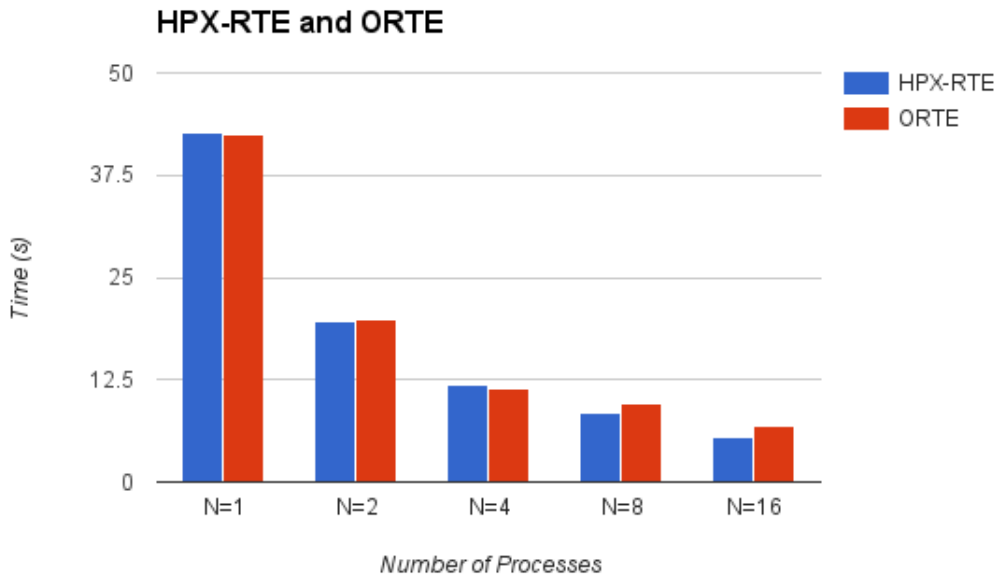

Figure 4.3: CFD Running Time - 80x80x80

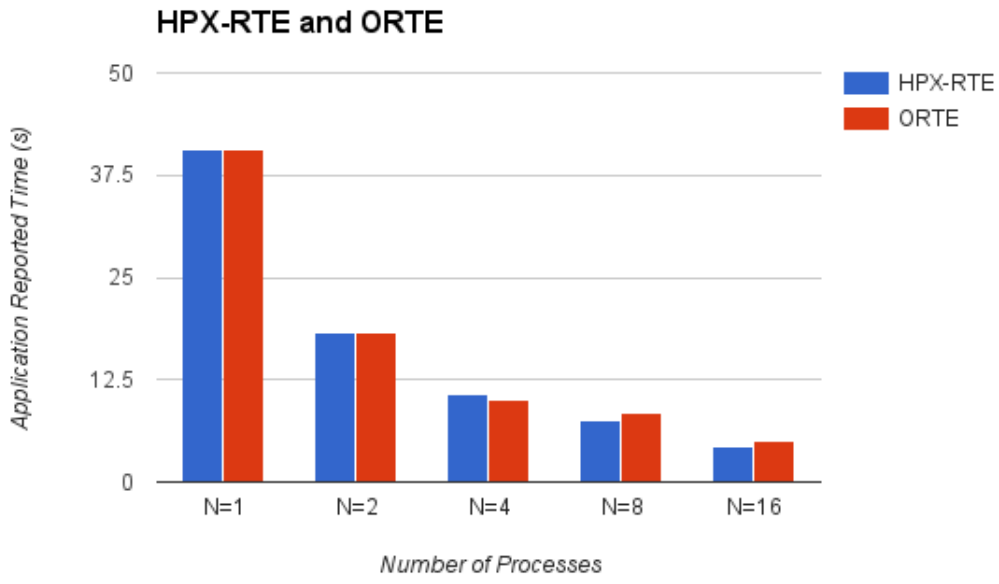

Figure 4.4: CFD Application Reported Time - 80x80x80

4.3.2 MPI Hello World

A very-good example to study the performance of the runtime would be a simple "Hello World" example (Listing 4.5). Since this simple application includes minimum communication and computation time, it could give us a better picture of the time spent in the runtime. Figure 4.5 illustrates the running time measured by the "time" command. Figure 4.6 shows the time measured starting before `MPI_Init()` until after `MPI_Finalize()` funtion. There is an improvement of up to **53%** using HPX-RTE over ORTE.

Listing 4.5: MPI Hello World

```
#include <mpi.h>
#include <hpx/hpx_main.hpp>
int main (int argc, char** argv)
{
  int rank, size;
  double t1, t2;
  t1 = MPI_Wtime();
  MPI_Init (&argc, &argv);
  MPI_Comm_rank (MPI_COMM_WORLD, &rank);
  MPI_Comm_size (MPI_COMM_WORLD, &size);
  printf( ``Hello world from process %d of %d\n'', rank, size );
  MPI_Finalize();
  t2 = MPI_Wtime();
  printf(``%1.4f\n'', t2-t1);
  return 0;
}
```

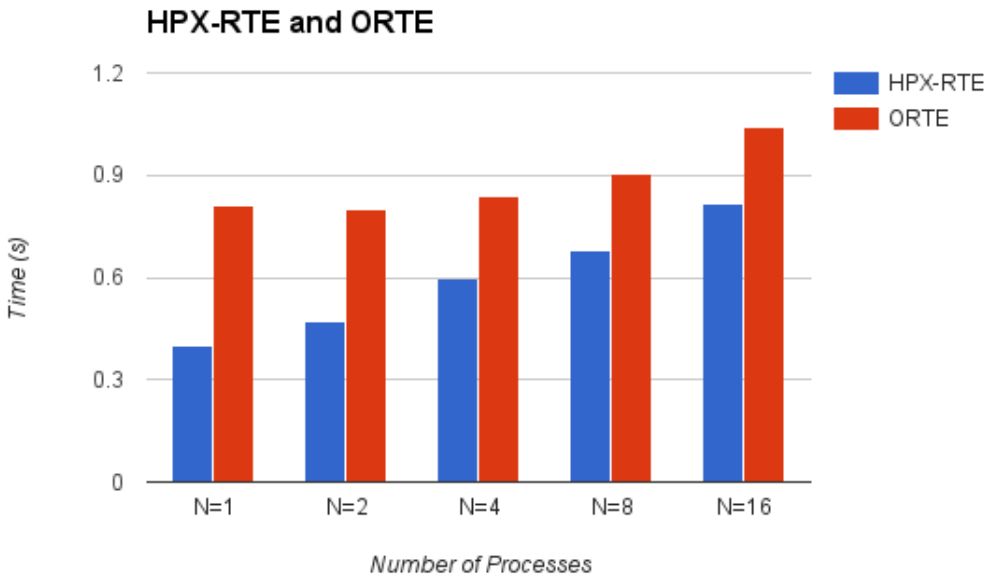

Figure 4.5: Time - Hello World

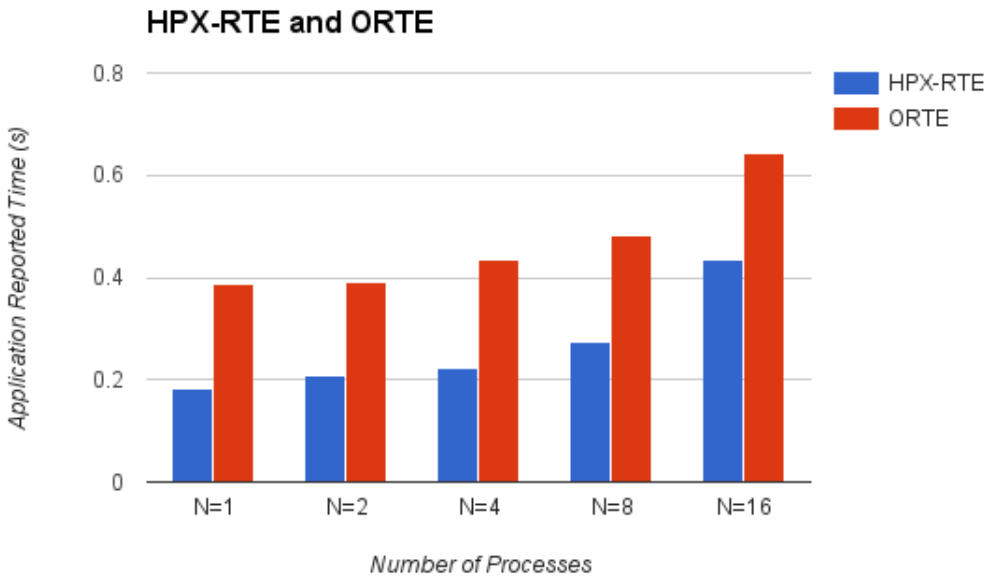

Figure 4.6: Application Reported Time - Hello World

4.3.3 Parallel Smoothing

This application performs smoothing on an input image on a pixel-by-pixel basis. The algorithm changes the class that a pixel has been assigned to if majority of neighboring elements have a different class. For each pixel, we consider an area of 5x5 pixels (two pixels in each direction). Input file contains the integer value of the class that each pixel belongs to. The algorithm is performed iteratively 10 times, and the result is written to an output file.

Figure 4.7 and 4.8 show the running time and application reported time accordingly. The input image size is 1024x1024 pixels.

We can see a slight performance benefit using HPX-RTE in both cases. However, this is more obvious in the overal reported times. In the case of application reported time in this algorithm, the application is reporting only the smoothing part (communication and computation). The majority of the time spent in runtime (start up and shut down), reading from the input file, and writing to the output file are not inside the section of the code the application is reporting the time for. This shows that the computation and communication times are very close, and the actual difference comes from the choice of runtime. We ran the same test on an input image of size 2048x2048 pixels. The results are shown in Figure 4.9 and Figure 4.10.

The comparison between two different input sizes shows that by increasing the problem size, the different between two runtimes becomes smaller and smaller. This is in accordance with what we suggested previously that the HPX-RTE runtime by itself is faster than ORTE, but the difference shows itself when the application is not dominated by communicational and computational work load.

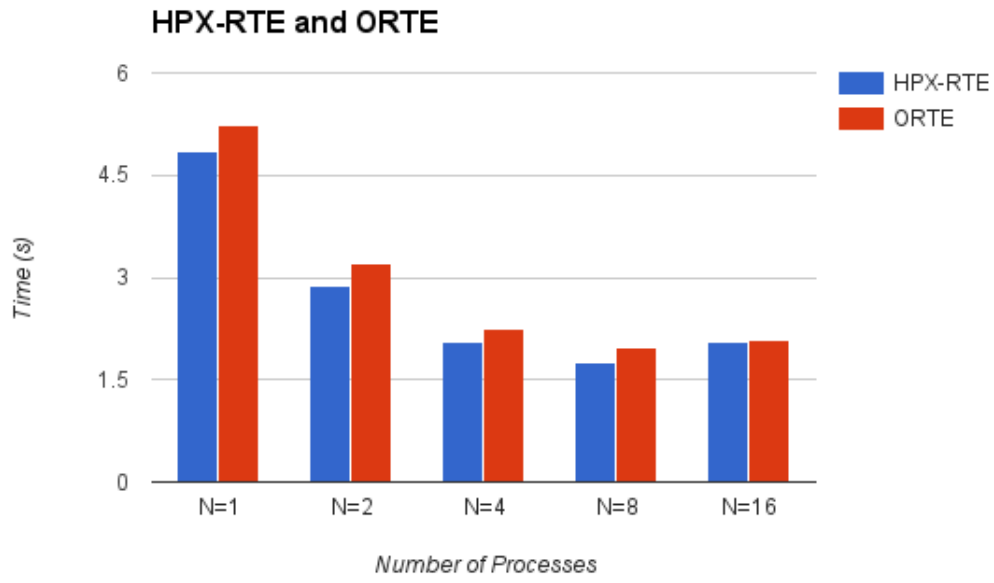

Figure 4.7: Time - Parallel Smoothing - 1024x1024

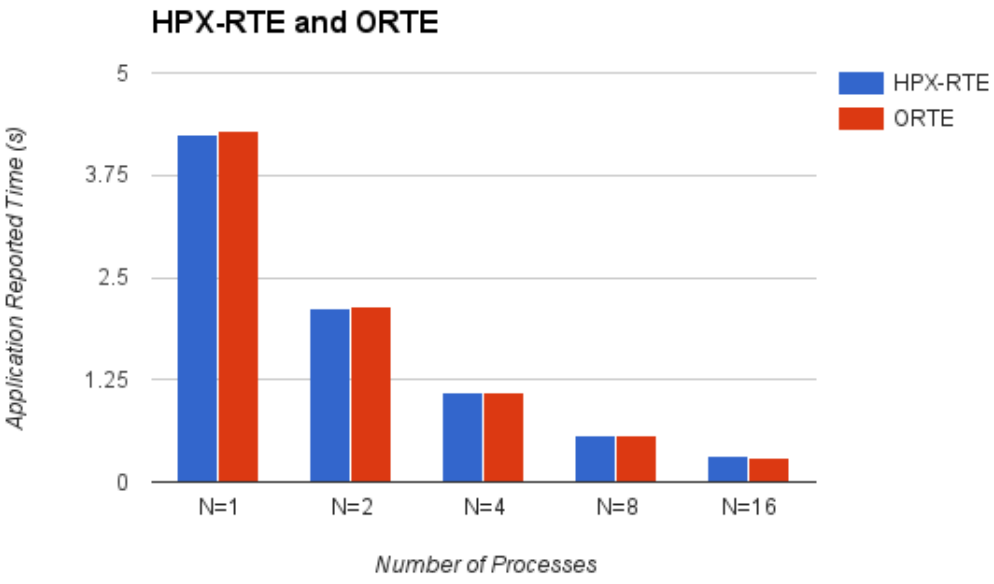

Figure 4.8: Application Reported Time - Parallel Smoothing - 1024x1024

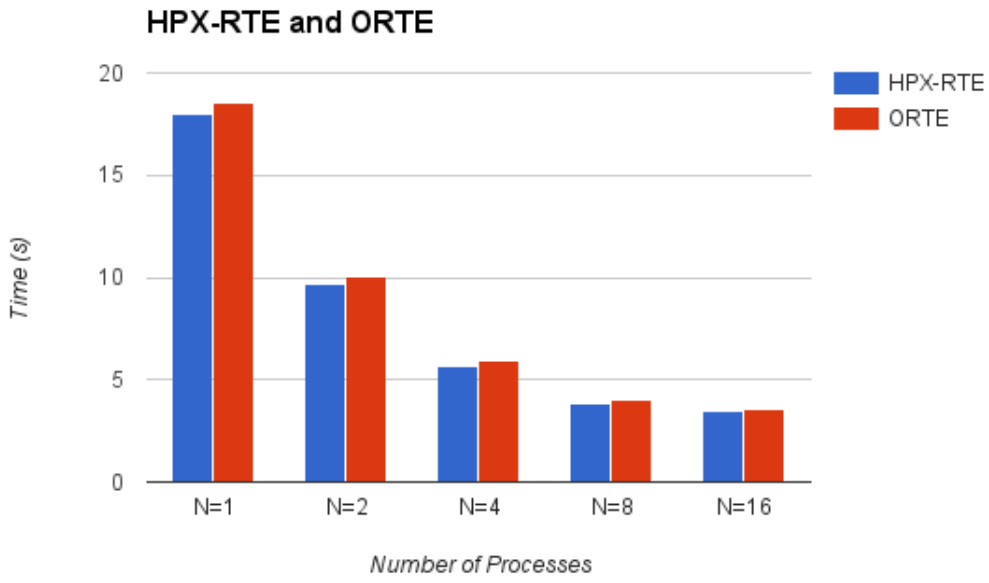

Figure 4.9: Time - Parallel Smoothing - 2048x2048

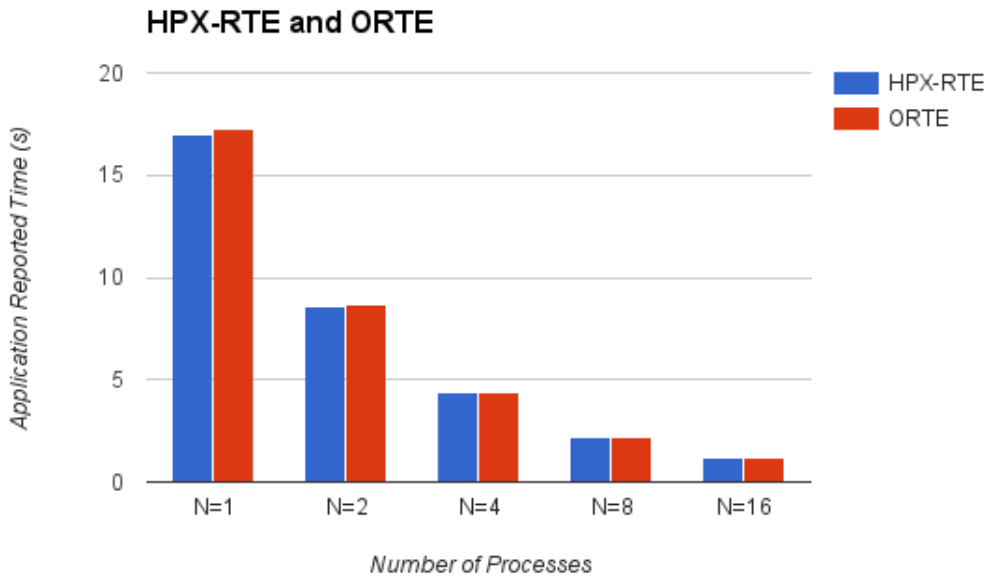

Figure 4.10: Application Reported Time - Parallel Smoothing - 2048x2048

4.3.4 Code Size

Figure 4.11 shows a comparison between HPX-RTE and ORTE code size (number of lines of code).

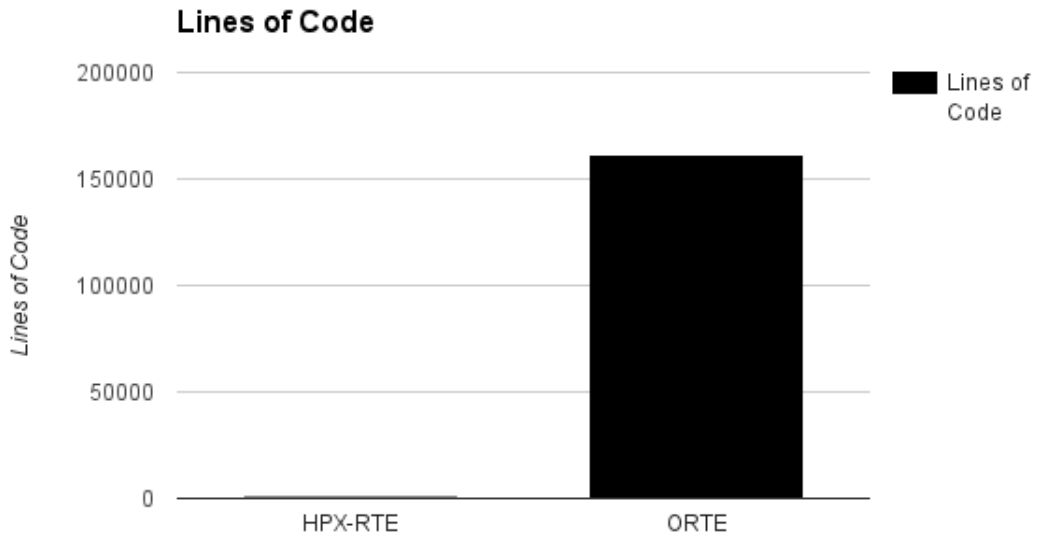

Figure 4.11: Code Size Comparison - HPX-RTE and ORTE

As it is shown in Figure 4.11, the source code size for HPX-RTE is less than **0.64%** of the code base for ORTE. This is partly because HPX-RTE is taking advantage of the HPX library and HPX-RTE component does not currently provide support for a number of features such as failure handling and dynamic process management that are supported by ORTE. Our evaluations in previous sections showed significant runtime performance improvement and very close overall performance using HPX-RTE over ORTE. Achieveing this level of performance with reducing the code size of the runtime by more than **99.36%** is a significant improvement.

Chapter 5

Conclusion

High-performance computing systems are growing toward hundreds-of-thousands to million-node machines, utilizing the computing power of billions of cores. Running parallel applications on such large machines efficiently will require optimized runtime environments that are scalable and resilient. Multi and many core chip architectures in large scale supercomputers pose several new challenges to designers of operating systems and runtime environments.

HPX-RTE is a new, light weight, and open source runtime system specifically designed for the emerging exascale computing environment. The system is designed relying on HPX project advanced features such as asynchronous remote function calls (actions) and C++ futures to allow for easy extension and transparent scalability. HPX-RTE provides full compatibility for current MPI applications to run on HPX runtime system.

5.1 Performance

Even though funcionality was the main priority in the design of HPX-RTE, our evaluations of HPX-RTE show better or equivalent performance compared to ORTE. We demonstrated the results of three different applications with different problem sizes: Parallel computational fluid dynamics, hello world, and parallel smoothing.

The source code size for HPX-RTE is more than **99.36%** smaller than ORTE's source code. This is partly because of utlizing HPX external library and also not supporting all the features provided by ORTE. The smaller code base makes the source code much simpler and easier to understand. Moreover, based on our evaluations the significant smaller size of runtime has not sacrificed the performance.

ORTE also supports a direct launch mechanism which avoids creating the ORTE daemons on individual compute nodes but uses the native resource manager instead for the management services, e.g., an application can be started directly using the srun command in a SLURM enviornment. This version has lower startup costs, but reduces the functionality of the runtime environment.

5.2 Future Extensions

HPX-RTE and the features it utilizes can be further integrated into Open MPI project in future. Some of the ways this could be accomplished include:

- **Further Evaluation and Performance Optimization**

  Since performance was not the main focus of the implementation of HPX-RTE, further evaluations can provide better insight into different performance aspects

of HPX-RTE that can be improved. Future developments of HPX-RTE can specifically target performance and make further improvements to the source code.

- **More Than One Locality Per Node**

  Current version of HPX-RTE is limited to one locality per node. Adding support for more than one locality (process) per node could be an extension in future versions.

- **Hybrid Programming Models**

  With the tight integration of HPX into Open MPI runtime provided by HPX-RTE, the possibility of hybrid programming models such as HPX-MPI is not far from reach. HPX-RTE also makes the transition from MPI to HPX easy. Application developers can replace parts of their code with HPX based implementation while their entire application does not need to be replaced and application functionality is maintained.

- **Incorporating HPX Features into Open MPI Frameworks**

  Taking advantage of the compatibility provided by HPX-RTE, Open MPI developers could incorporate features from HPX project into the implementaion of components in other frameworks within the Open MPI project with similar design principles we had in mind for HPX-RTE. This could potentially improve performance, decrease the source code size, and provide simpler code.

Bibliography

[1] Blaise Barney et al. Introduction to parallel computing. *Lawrence Livermore National Laboratory*, 6(13):10, 2010.

[2] G. S. Almasi and A. Gottlieb. *Highly Parallel Computing*. Benjamin-Cummings Publishing Co., Inc., Redwood City, CA, USA, 1989.

[3] M. Flynn. Some computer organizations and their effectiveness. *Computers, IEEE Transactions on*, C-21(9):948–960, Sept 1972.

[4] R. Duncan. A survey of parallel computer architectures. *Computer*, 23(2):5–16, Feb 1990.

[5] David A Patterson and John L Hennessy. *Computer organization and design: the hardware/software interface*. Newnes, 2013.

[6] Ajay D Kshemkalyani and Mukesh Singhal. *Distributed computing: principles, algorithms, and systems*. Cambridge University Press, 2008.

[7] Herb Sutter. The free lunch is over: A fundamental turn toward concurrency in software. *Dr. Dobbs journal*, 30(3):202–210, 2005.

[8] Parallel programming models. `https://en.wikipedia.org/wiki/Parallel_programming_model`. Accessed: 2015-09-14.

[9] Michael J. Quinn. *Parallel Programming in C with MPI and OpenMP*. McGraw-Hill Education Group, 2003.

[10] Message Passing Interface Forum. Mpi 3.0 standard. `http://www.mpi-forum.org/docs/mpi-3.0/mpi30-report.pdf`. Accessed: 2015-10-14.

[11] William Gropp, Ewing Lusk, Nathan Doss, and Anthony Skjellum. A high-performance, portable implementation of the mpi message passing interface standard. *Parallel computing*, 22(6):789–828, 1996.

[12] Brandon Barker. Message passing interface (mpi). In *Workshop: High Performance Computing on Stampede*, 2015.

[13] Marc Snir. *MPI–the Complete Reference: The MPI core*, volume 1. MIT press, 1998.

[14] Al Geist, William Gropp, Steve Huss-Lederman, Andrew Lumsdaine, Ewing Lusk, William Saphir, Tony Skjellum, and Marc Snir. Mpi-2: Extending the message-passing interface. In *Euro-Par'96 Parallel Processing*, pages 128–135. Springer, 1996.

[15] David W Walker and Jack J Dongarra. Mpi: a standard message passing interface. *Supercomputer*, 12:56–68, 1996.

[16] William Gropp, Ewing Lusk, and Rajeev Thakur. *Using MPI-2: Advanced features of the message-passing interface*. MIT press, 1999.

[17] William Gropp. Mpich2: A new start for mpi implementations. In *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, pages 7–7. Springer, 2002.

[18] Edgar Gabriel, Graham E. Fagg, George Bosilca, Thara Angskun, Jack J. Dongarra, Jeffrey M. Squyres, Vishal Sahay, Prabhanjan Kambadur, Brian Barrett, Andrew Lumsdaine, Ralph H. Castain, David J. Daniel, Richard L. Graham, and Timothy S. Woodall. Open MPI: Goals, concept, and design of a next generation MPI implementation. In *Proceedings, 11th European PVM/MPI Users' Group Meeting*, pages 97–104, Budapest, Hungary, September 2004.

[19] George Bosilca, Thomas Herault, Ala Rezmerita, and Jack Dongarra. On scalability for mpi runtime systems. In *Cluster Computing (CLUSTER), 2011 IEEE International Conference on*, pages 187–195. IEEE, 2011.

[20] Torsten Hoefler and Kamil Iskra. Operating systems and runtime environments on supercomputers. *Int. J. High Perform. Comput. Appl.*, 26(2):93–94, May 2012.

[21] R.H. Castain, T.S. Woodall, D.J. Daniel, J.M. Squyres, B. Barrett, and G.E. Fagg. The open run-time environment (openrte): A transparent multicluster environment for high-performance computing. *Future Generation Computer Systems*, 24(2):153 – 157, 2008.

[22] G.R. Gao, T. Sterling, R. Stevens, M. Hereld, and Weirong Zhu. Parallex: A study of a new parallel computation model. In *Parallel and Distributed Processing Symposium, 2007. IPDPS 2007. IEEE International*, pages 1–6, March 2007.

[23] Hartmut Kaiser, Thomas Heller, Bryce Adelstein-Lelbach, Adrian Serio, and Dietmar Fey. Hpx: A task based programming model in a global address space. In *Proceedings of the 8th International Conference on Partitioned Global Address Space Programming Models*, PGAS '14, pages 6:1–6:11, New York, NY, USA, 2014. ACM.

[24] Open mpi: Open source high performance computing. `http://www.open-mpi.org`. Accessed: 2015-09-14.

[25] Brian Barrett, Jeffrey M Squyres, Andrew Lumsdaine, Richard L Graham, and George Bosilca. Analysis of the component architecture overhead in open mpi. In *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, pages 175–182. Springer, 2005.

[26] Richard L Graham, Timothy S Woodall, and Jeffrey M Squyres. Open mpi: A flexible high performance mpi. In *Parallel Processing and Applied Mathematics*, pages 228–239. Springer, 2006.

[27] Jeffrey M Squyres and Andrew Lumsdaine. A component architecture for lam/mpi. In *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, pages 379–387. Springer, 2003.

[28] David E Bernholdt, Benjamin A Allan, Robert Armstrong, Felipe Bertrand, Kenneth Chiu, Tamara L Dahlgren, Kostadin Damevski, Wael R Elwasif, Thomas GW Epperly, Madhusudhan Govindaraju, et al. A component architecture for high-performance scientific computing. *International Journal of High Performance Computing Applications*, 20(2):163–202, 2006.

[29] Eric P Kronstadt. Peta-scale computing. In *Parallel and Distributed Processing Symposium, 2005. Proceedings. 19th IEEE International*, pages 1–pp. IEEE, 2005.

[30] Jeffrey M Squyres. *A Component Architecture for the Message Passing Interface (MPI): The Systems Services Interface (SSI) of LAM/MPI.* PhD thesis, University of Notre Dame, 2004.

[31] Rob T Aulwes, David J Daniel, Nehal N Desai, Richard L Graham, L Dean Risinger, Mark A Taylor, Timothy S Woodall, and Mitchel W Sukalski. Architecture of la-mpi, a network-fault-tolerant mpi. In *Parallel and Distributed Processing Symposium, 2004. Proceedings. 18th International*, page 15. IEEE, 2004.

[32] Graham E Fagg and Jack J Dongarra. Harness fault tolerant mpi design, usage and performance issues. *Future Generation Computer Systems*, 18(8):1127–1142, 2002.

[33] Ralph H Castain, Timothy S Woodall, David J Daniel, Jeffrey M Squyres, Brian Barrett, and Graham E Fagg. The open run-time environment (openrte): A transparent multi-cluster environment for high-performance computing. In *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, pages 225–232. Springer, 2005.

[34] An open source run-time for distributed hpc. `http://www.hpcwire.com/2006/03/31/an_open_source_run-time_for_distributed_hpc-1/`. Accessed: 2015-09-29.

[35] Ralph H Castain, Timothy S Woodall, David J Daniel, Jeffrey M Squyres, Brian Barrett, and Graham E Fagg. The open run-time environment (openrte): A transparent multicluster environment for high-performance computing. *Future Generation Computer Systems*, 24(2):153–157, 2008.

[36] Hartmut Kaiser, Maciej Brodowicz, and Thomas Sterling. Parallex an advanced parallel execution model for scaling-impaired applications. In *Parallel Processing Workshops, 2009. ICPPW'09. International Conference on*, pages 394–401. IEEE, 2009.

[37] T Sterling and C Dekate. Enabling exascale through parallex paradigm. In *Parallel Computational Fluid Dynamics*, pages 3–18, 2010.

[38] Alexandre Tabbal, Matthew Anderson, Maciej Brodowicz, Hartmut Kaiser, and Thomas Sterling. Preliminary design examination of the parallex system from a software and hardware perspective. *ACM SIGMETRICS Performance Evaluation Review*, 38(4):81–87, 2011.

[39] Thomas Sterling, Matthew Anderson, P Kevin Bohan, Maciej Brodowicz, Abhishek Kulkarni, and Bo Zhang. Towards exascale co-design in a runtime system. In *Solving Software Challenges for Exascale*, pages 85–99. Springer, 2014.

[40] Guang R Gao, Thomas Sterling, Rick Stevens, Mark Hereld, and Weirong Zhu. Parallex: A study of a new parallel computation model. In *Parallel and Distributed Processing Symposium, 2007. IPDPS 2007. IEEE International*, pages 1–6. IEEE, 2007.

[41] Chirag Dekate, Matthew Anderson, Maciej Brodowicz, Hartmut Kaiser, Bryce Adelstein-Lelbach, and Thomas Sterling. Improving the scalability of parallel n-body applications with an event-driven constraint-based execution model. *International Journal of High Performance Computing Applications*, 26(3):319–332, 2012.

[42] Tim Stitt. *An introduction to the Partitioned Global Address Space (PGAS) programming model*. Connexions, Rice University, 2009.

[43] Dan Bonachea. Gasnet specification, v1. 1. 2002.

[44] UPC Consortium et al. Upc language specifications v1. 2. *Lawrence Berkeley National Laboratory*, 2005.

[45] Kevin Huck, Sameer Shende, Allen Malony, Hartmut Kaiser, Allan Porterfield, Rob Fowler, and Ron Brightwell. An early prototype of an autonomic performance environment for exascale. In *Proceedings of the 3rd International Workshop on Runtime and Operating Systems for Supercomputers*, page 8. ACM, 2013.

[46] Hartmut Kaiser, Thomas Heller, Bryce Adelstein-Lelbach, Adrian Serio, and Dietmar Fey. Hpx: A task based programming model in a global address space. In *Proceedings of the 8th International Conference on Partitioned Global Address Space Programming Models*, page 6. ACM, 2014.

[47] Artur Podobas, Mats Brorsson, and Karl-Filip Faxén. A comparison of some recent task-based parallel programming models. 2010.

[48] Chuck Pheatt. Intel® threading building blocks. *Journal of Computing Sciences in Colleges*, 23(4):298–298, 2008.

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

[50] Cédric Augonnet, Samuel Thibault, Raymond Namyst, and Pierre-André Wacrenier. Starpu: a unified platform for task scheduling on heterogeneous

multicore architectures. *Concurrency and Computation: Practice and Experience*, 23(2):187–198, 2011.

[51] Leonardo Dagum and Rameshm Enon. Openmp: an industry standard api for shared-memory programming. *Computational Science & Engineering, IEEE*, 5(1):46–55, 1998.

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

[53] Bradford L Chamberlain, David Callahan, and Hans P Zima. Parallel programmability and the chapel language. *International Journal of High Performance Computing Applications*, 21(3):291–312, 2007.

[54] Matt Pharr and William R Mark. ispc: A spmd compiler for high-performance cpu programming. In *Innovative Parallel Computing (InPar), 2012*, pages 1–13. IEEE, 2012.

[55] Matthew Anderson, Maciej Brodowicz, Thomas Sterling, Hartmut Kaiser, and Bryce Adelstein-Lelbach. Tabulated equations of state with a many-tasking execution model. In *Parallel and Distributed Processing Symposium Workshops & PhD Forum (IPDPSW), 2013 IEEE 27th International*, pages 1691–1699. IEEE, 2013.

[56] The C++ Standards Committee. iso/iec 14882:2011, standard for programming language c++. `http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2012/n3376.pdf`. Accessed: 2015-10-05.

[57] The C++ Standards Committee. N3797: Working draft, standard for programming language c++. `http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2014/n4296.pdf`. Accessed: 2015-10-05.

[58] Beman Dawes, David Abrahams, and Rene Rivera. Boost c++ libraries. *URL http://www.boost.org*, 35:36, 2009.

[59] Bibek Ghimire. *Data Distribution in HPX*. PhD thesis, Faculty of the Louisiana State University and Agricultural and Mechanical College in partial fulfillment of the requirements for the degree of Master of (degree) in System Science by Bibek Ghimire BS, Louisiana State University, 2014.

[60] Bjarne Stroustrup. *The C++ programming language*. Pearson Education, 2013.

[61] LSU Stellar Group. Hpx 0.9.11. http://stellar-group.github.io/hpx/docs/html/hpx.html#hpx.tutorial.examples.hello_world. Accessed: 2015-10-29.

[62] Ronald Brian Brightwell. Xpress project overview. Technical report, Sandia National Laboratories (SNL-NM), Albuquerque, NM (United States), 2013.

[63] Ronald Brian Brightwell. Xpress - extreme scale software stack - x-stack. https://xstackwiki.modelado.org/XPRESS. Accessed: 2015-10-07.

[64] V. Cerf and R.E. Kahn. A protocol for packet network intercommunication. *Communications, IEEE Transactions on*, 22(5):637–648, May 1974.

[65] InfiniBand Trade Association et al. *InfiniBand Architecture Specification: Release 1.0*. InfiniBand Trade Association, 2000.

[66] Andy B Yoo, Morris A Jette, and Mark Grondona. Slurm: Simple linux utility for resource management. In *Job Scheduling Strategies for Parallel Processing*, pages 44–60. Springer, 2003.

[67] Michael Resch, Björn Sander, and Isabel Loebich. A comparison of openmp and mpi for the parallel cfd test case. In *Proc. of the First European Workshop on OpenMP*, pages 71–75, 1999.

[68] Akin Ecer, N Satofuka, Jacques Periaux, and S Taylor. *Parallel Computational Fluid Dynamics' 95: Implementations and Results Using Parallel Computers*. Elsevier, 1996.