# GePSeA: A General-Purpose Software Acceleration Framework for Lightweight Task Offloading

Ajeet Singh

Thesis submitted to the Faculty of the

Virginia Polytechnic Institute and State University

in partial fulfillment of the requirements for the degree of

Master of Science

in

Computer Science and Applications

Wu-chun Feng, Chair

Pavan Balaji

Eli Tilevich

July 14, 2008

Blacksburg, Virginia

Keywords: Task offloading, multi-core, many-core, accelerators

© Copyright 2009, Ajeet Singh

# GePSeA: A General-Purpose Software Acceleration Framework for Lightweight Task Offloading

Ajeet Singh

(ABSTRACT)

Hardware-acceleration techniques continue to be used to boost the performance of scientific codes. To do so, software developers identify portions of these codes that are amenable for offloading and map them to hardware accelerators. However, offloading such tasks to specialized hardware accelerators is non-trivial. Furthermore, these accelerators can add significant cost to a computing system.

Consequently, this thesis proposes a framework called GePSeA (General Purpose Software Acceleration Framework), which uses a small fraction of the computational power on multi-core architectures to offload complex application-specific tasks. Specifically, GePSeA provides a lightweight process that acts as a helper agent to the application by executing application-specific tasks asynchronously and efficiently. GePSeA is not meant to replace hardware accelerators but to extend them. GePSeA provide several utilities called core components that offload tasks on to the core or to the special-purpose hardware when available in a way that is transparent to the application. Examples of such core components include reliable communication service, distributed lock management, global memory management, dynamic load distribution and network protocol processing. We then apply the GePSeA framework to two applications, namely mpiBLAST, an open-source computational biology application and Reliable Blast UDP (RBUDP) based file transfer application. We observe significant speed-up for both applications.

***To my Grandpa who has been a source of inspiration in every walk of my life and continue to do so.***

# Acknowledgments

I thank my adviser Dr. Wu-chun Feng for his invaluable support and guidance. It has been a great learning experience working with him for past one and half years. He has taken keen interest in my research work and provided me his feedback from time to time which helped me to improve the quality of my research work.

I thank Dr. Pavan Balaji at Argonne National Laboratory with whom I closely worked on many research projects. I have benefited greatly from his experience in conducting research work and his in-depth knowledge on various subject matters which helped me to drive these projects. I thank my committee member Dr. Eli Tilevich for his support for my work and for numerous healthy discussions we had for past one year.

I thank all the members of Synergy Lab at VT for their support. The constructive suggestions and criticisms offered by them during group meetings have been very helpful in shaping up this thesis. In particular, I would like to thank Jeremy Archuleta and Heshan Lin with whom I had many informal discussions about my research which helped me to expedite my research work. I also thank Jeremy for the administrative help he rendered to set up ICE cluster for my testing.

I thank all my friends who have made my stay in Blacksburg very enjoyable. I thank my parents for their unconditional love and support.

# Contents


**1 Introduction** **1**

1.1 Motivation . . . 3

1.2 Contributions . . . 4

1.3 Outline . . . 5

**2 Background** **6**

2.1 Terminology Used . . . 6

2.2 Multi/Many-Core Systems . . . 6

2.3 Application Accelerators . . . 7

2.4 Myri-10G Network Interface Cards . . . 8

**3 GePSeA: Design and Implementation** **10**

3.1 GePSeA Infrastructure and Communication Layer . . . 11

3.2 Application Plug-ins Layer . . . 14

3.3 Core Components . . . 14

3.3.1 Data Management Core Components . . . 15

3.3.2 Memory Management Core Components . . . . . . . . . . . . . . . . 17
3.3.3 Coordination and Synchronization Core Components . . . . . . . . . . 18
3.4 Accelerator to Core Mapping . . . . . . . . . . . . . . . . . . . . . . . . . . 25
3.5 Discussion . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 26

**4 Case Study: mpiBLAST over GePSeA Framework 27**
4.1 mpiBLAST Overview . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 27
4.2 mpiBLAST over GePSeA . . . . . . . . . . . . . . . . . . . . . . . . . . . . 29
4.2.1 Asynchronous Output Consolidation Plug-in . . . . . . . . . . . . . . . 30
4.2.2 Runtime Output Compression Plug-in . . . . . . . . . . . . . . . . . . 30
4.2.3 Hot-Swap Database Fragments Plug-in . . . . . . . . . . . . . . . . . . 31

**5 Case Study: RBUDP File Transfer over GePSeA Framework 32**
5.1 RBUDP File Transfer over GePSeA . . . . . . . . . . . . . . . . . . . . . . 33
5.2 Hardware-Assisted UDP Acceleration . . . . . . . . . . . . . . . . . . . . . 33
5.2.1 Protocol Offload Engines . . . . . . . . . . . . . . . . . . . . . . . . . 34
5.2.2 High Performance Sockets . . . . . . . . . . . . . . . . . . . . . . . . 35
5.2.3 Overview of Existing TCP/IP Implementation . . . . . . . . . . . . . 36
5.2.4 Modified TCP/IP Flow . . . . . . . . . . . . . . . . . . . . . . . . . . 38

**6 Experimental Evaluation and Analysis 41**
6.1 Evaluation of mpiBLAST over GePSeA Framework . . . . . . . . . . . . . . 41
6.1.1 Experimental Setup . . . . . . . . . . . . . . . . . . . . . . . . . . . . 42

6.1.2 Accelerator on “Committed” Core . . . . . . . . . . . . . . . . . . . . . 42
6.1.3 Accelerator on “Available” Core . . . . . . . . . . . . . . . . . . . . . 44
6.1.4 Accelerator with Unequal Workers . . . . . . . . . . . . . . . . . . . . 45
6.1.5 Increase in Problem Size . . . . . . . . . . . . . . . . . . . . . . . . . 46
6.1.6 Worker Search Time . . . . . . . . . . . . . . . . . . . . . . . . . . . 47
6.1.7 Distributed Output Processing Core Component . . . . . . . . . . . . 47
6.1.8 Dynamic Load Balancing Core Component . . . . . . . . . . . . . . . 48
6.1.9 Data Compression Core Component . . . . . . . . . . . . . . . . . . . 49
6.2 Evaluation of RBUDP File Transfer over GePSeA Framework . . . . . . . . 50
6.2.1 Experimental Setup . . . . . . . . . . . . . . . . . . . . . . . . . . . . 50
6.2.2 Hardware-Assisted UDP Acceleration . . . . . . . . . . . . . . . . . . 51
6.2.3 High-Speed Reliable UDP Core Component . . . . . . . . . . . . . . 52
6.2.4 Hardware Vs Software-Based UDP Acceleration . . . . . . . . . . . . 54

**7 Related Work** **55**

**8 Conclusions and Future Work** **57**
8.1 Concluding Remarks . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 57
8.2 Discussion and Future Work . . . . . . . . . . . . . . . . . . . . . . . . . 58

**Bibliography** **61**

# List of Figures


2.1 Intel Core 2 Duo Processor . . . . . . . . . . . . . . . . . . . . . . . . . . . 7

3.1 Layered Architecture of GePSeA Framework . . . . . . . . . . . . . . . . . . . . 11

3.2 Application-Accelerator Interactions Within a Node . . . . . . . . . . . . . . . 12

3.3 Application-Accelerator Interactions for a Three-Node Cluster . . . . . . . . 13

3.4 High-Speed Reliable UDP Core Component Design . . . . . . . . . . . . . . 22

3.5 Algorithm for Receiving Data . . . . . . . . . . . . . . . . . . . . . . . . . 23

3.6 Algorithm for Sending Data . . . . . . . . . . . . . . . . . . . . . . . . . . 24

4.1 mpiBLAST over GePSeA Framework . . . . . . . . . . . . . . . . . . . . . 29

5.1 High Performance UDP/IP Sockets Architecture . . . . . . . . . . . . . . . . . 35

5.2 Simplified TCP Flow Diagram . . . . . . . . . . . . . . . . . . . . . . . . . 39

6.1 Node Configuration for Experiment 6.1.2 . . . . . . . . . . . . . . . . . . . . 43

6.2 Speed-up Obtained by Running Accelerator on Committed Core . . . . . . . 43

6.3 Node Configuration for Experiment 6.1.3 . . . . . . . . . . . . . . . . . . . . 44

6.4 Speed-up Obtained by Running Accelerator on Available Core . . . . . . . . 45

6.5 Node Configuration for Experiment 6.1.4 . . . . . . . . . . . . . . . . . . . . 45

6.6 Speed-up Obtained by Running Accelerator for Unequal Workers . . . . . . 46

6.7 Speed-up Obtained with Increase in Problem Size . . . . . . . . . . . . . . . . 47

6.8 Worker Search Time as a Percentage of Total Time . . . . . . . . . . . . . . . 48

6.9 Distributed Output Processing Feature of GePSeA . . . . . . . . . . . . . . . 48

6.10 Dynamic Load Balancing Feature of GePSeA . . . . . . . . . . . . . . . . . . 49

6.11 Data Compression Feature of GePSeA . . . . . . . . . . . . . . . . . . . . . . 50

6.12 Evaluation of UDP Offload Core Component . . . . . . . . . . . . . . . . . . . 51

# List of Tables

6.1 File Transfer using Single System Core . . . . . . . . . . . . . . . . . . . . . . 53

6.2 File Transfer using Two System Cores . . . . . . . . . . . . . . . . . . . . . . 53

6.3 File Transfer using Three System Cores . . . . . . . . . . . . . . . . . . . . . . 53

# Chapter 1

# Introduction

Hardware acceleration techniques have been used to improve specific computational kernels for many years. However, such accelerators have typically focused on accelerating compute intensive portions for application, including Fourier transformations, search modules, and other numerical methods. However, a secondary benefit of hardware accelerators is their capability to serve as dedicated asynchronous engines, as compared to general purpose CPUs that are designed to be shared between multiple processes.

As high-end computing systems and their associated applications continue to grow in scale and complexity, the need for offloading more complex tasks is becoming clear. With systems scaling to hundreds of thousands of cores today, complex tasks such as asynchronous synchronization and data management are becoming extremely important. However, such tasks are orthogonal to the primary capabilities of existing hardware acceleration units. Consequently, offloading these tasks would result in a significant increase in the complexity and overall cost of these hardware units, which is undesirable. On the other hand, multi- and many-core architectures are becoming increasingly common today, with quad- and hex-core processors already available and 16- and 80-core processors on the roadmap [21, 6]. These trends point to the fact that today, and more so in the future, each physical node will have a massive number of processing units which, for some tasks, can be viewed as low-cost alternatives to

expensive hardware accelerators.

Software-based accelerators have not received much attention from the research community in the past mainly because single-core processors are only capable of application-level parallelism through time-sharing of the single CPU core. However, as mentioned earlier, with availability of additional horsepower in terms of additional cores on multi-core systems, software-based accelerators offer an attractive alternative to accelerate applications by effectively utilizing the system's *multiple* computing resources. For example, software accelerators can asynchronously handle communication tasks while the application is performing the computation i.e. core processing.

Consequently in this thesis we propose GePSeA—a general Purpose software acceleration framework that can onload complex application-specific tasks as well. Specifically, GePSeA provides a number of utility components called core components that support different functionalities as well as a generic interface for applications to utilize these components through simple plug-ins. In this thesis, we present three categories of utility components: (i) data management components, (ii) memory management components, and (iii) synchronization and coordination components. However, the general-purpose nature of our framework allows it to be extended to other categories as well. In our design, different utility components are aggregated together into a lightweight process referred to as a *software accelerator*. This process acts as a helper agent to the application by executing lightweight application-specific tasks asynchronously and efficiently.

GePSeA does not aim to replace the dedicated and specialized hardware accelerators such as GPGPUs and Cell processors. Instead, it utilizes the existing hardware-offloaded features of such accelerators when available and extend them by onloading complex application-specific functionalities that cannot be easily offloaded to the hardware. GePSeA framework provide several utilities called core components that offload tasks on to a system core or to a special-purpose hardware when available in a way that is transparent to the application. Of particular note is the high-speed reliable UDP core component presented in this thesis.

It proposes a novel software-based technique to accelerate UDP application by utilizing the system cores effectively. GePSeA provide several other core components such as reliable communication service, distributed lock management, global memory management, dynamic load distribution, distributed data sorting that efficiently perform data management and data I/O for parallel applications running on multi-core clusters.

We evaluate GePSeA framework using two applications. We used an open-source computational biology application called mpiBLAST and a RBUDP based file transfer application as case studies to test the efficacy of the our framework. Our results show significant improvement in application performance in both the cases using GePSeA framework.

## 1.1 Motivation

There are several direct and indirect benefits of software accelerators. We list some of the important ones here that motivated us to build GePSeA framework.

Software accelerators, like hardware accelerators, boost the application performance. They execute tasks delegated to them by the applications asynchronously, efficiently and in parallel. Using software accelerator, applications can also overlap their computation and communication tasks. For example, accelerator can pre-fetch the new data set from the network for the application while the appication is working on the current data set thereby eliminating any application stalls. And since both application and the accelerator work simultaneously on the same problem possibly without contending for system resources, significant speed-up can be achieved.

Applications greatly benefit using hardware accelerators as they execute the tasks that they have been designed for in faster and efficient manner. However, developing application-specific hardware accelerators in not trivial. Huge investment in terms of time and money is required. Moreover, complexity in developing hardware accelerator increases exponentially with increase in complexity of tasks. Software accelerators on the other hand do not require

any specialized hardware and therefore can be very cost effective alternative to accelerate application .

Multi/many-cores systems are ubiquitous today. And with clock speed now stagnant, multi-core systems are only going to be more prevalent in the near future. Software advances, however, continue to lag behind. New software designs therefore need to be “core aware” in order to tap the increased horsepower of multiple cores. Software accelerators proposed in this thesis are designed to maximally utilize the processing power of system cores, thereby efficiently utilizing the system resources.

Software accelerators, unlike their hardware counterpart, are easy to develop. In fact, it is possible to re-use the code for developing different software accelerators. For example, developing accelerators for parallel programs often require generic utilities such as reliable communication, distributed lock management, global memory management, dynamic load balancing etc. Rather than forcing the developers to design these utilities from scratch, they can benefit by reusing the existing implementations if available. GePSeA framework proposed in this thesis provide such utilities for the development of software accelerator and also the infrastructure to add new utilities.

## 1.2 Contributions

Specifically, we make these contributions in this thesis:

- Proposing and designing GePSeA, a software acceleration framework for lightweight task offloading. GePSeA is a easy-to-use framework that provide infrastructure for quick development of application-specific software accelerators.
- Design of several general-purpose utilities called core components such as distributed data sorting, dynamic load balancing and data compression engine, for parallel and distributed applications. These core components are the building blocks of GePSeA

framework and are independent research components in themselves.

- Proposing high-speed reliable UDP core component to enable high-speed reliable UDP data transfer which is a "core aware" scheme that utilize the processing power of the cores on multi/many-core systems to accelerate UDP data transfers.

- Designing and implementing hardware-assisted UDP acceleration scheme to offload UDP protocol processing to latest generation network adapters such as Myricom's Myri-10G NIC that support *stateless* protocol processing for TCP. Specifically, we utilize adapter's TCP segment offload (TSO), large receive offload (LRO) and checksum offloading features to offload UDP protocol processing.

- Evaluation of the GePSeA framework using a popular computational biology application mpiBLAST as a case study. We developed software accelerator for mpiBLAST using the GePSeA framework and conducted several tests to prove the efficacy of the accelerator. We also evaluated GePSeA using RBUDP based file transfer application.

## 1.3 Outline

This thesis is organized as follows. In Chapter 2, we present the background, define terminology, and introduce concepts relevant to this thesis. Chapter 3 present the detailed design of the GePSeA framework including the design of some of the important core components. The case study of mpiBLAST application using the GePSeA framework is presented in Chapter 4. Another case study of RBUDP based file transfer application over GePSeA framework is presented in Chapter 5 that evaluates the efficacy of high-speed reliable core component provided by GePSeA. It is followed by a detailed experimental evaluation in Chapter 6. In Chapter 7, we discuss related work on accelerators and frameworks for multi/many-core architectures. Chapter 8 presents our conclusions and discussions for future work.

# Chapter 2

# Background

In this chapter, we present the background, define terminology, and introduce concepts relevant to this thesis.

## 2.1 Terminology Used

Terms *GePSeA* and *GePSeA framework* have been used interchangeably in this thesis. Both the terms refer to the software acceleration framework proposed in this thesis. Also terms, *GePSeA accelerator* and *software accelerator* used in this thesis refer to the software accelerator (helper process) built using GePSeA framework. The term *task onloading* used in this thesis refer to the process of executing the task on a general-purpose CPU (or core) which would "normally" run on a specialized hardware when available.

## 2.2 Multi/Many-Core Systems

Design obstacles due to power demands and heat concerns have limited CPU clock speeds. As the clock speed became stagnant, multi-core architectures evolved in order to increase the

processing power. In this section we discuss some the architectural details of multi-core systems.

A core refers to a processing unit. In a multi-core system, two or more cores are integrated on a single integrated chip die. Each core can execute instructions independent of other. As cores are on a single integrated circuit (and therefore physically close to each other), cache coherency is greatly improved. Figure 2.1 shows the architecture of Intel Core 2 Duo processor. Each processor has two cores integrated on a same die. Each core has its own L1 cache but they share L2 cache and system bus interface. Cache coherency protocols keep the caches coherent. One advantage of such architecture is that communication between the two cores does not have to go outside the die and therefore eliminates die pin overhead.

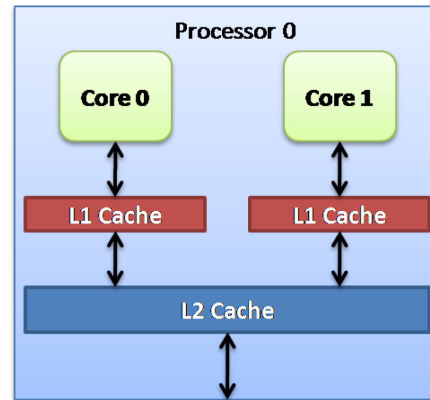


Figure 2.1: Intel Core 2 Duo Processor

One disadvantage of multi-core system is the block-waiting for shared resources. For example, system memory bandwidth is fixed. If there is too much memory contention between the two cores, then real-world advantage of having two cores drops considerably. From software developer point of view, multi-threaded applications must be carefully written to take the advantage of the cores on a multi-core system.

## 2.3 Application Accelerators

In high performance computing, efficient data management and data I/O often becomes the key to an application's performance. For example, inefficient delivery of data to the application may stall the application by forcing it to wait for the data's arrival. With ever increasing problem sizes and software complexity, offloading these complex and data intensive

tasks from the application can be critical to an application's performance.

Using specialized hardware for offloading is one popular technique. Hardware accelerators are blocks of logic that are designed to offload specific tasks from system processor, such as GPUs for offloading intensive graphics rendering. GPGPUs, for example, which provide some programming flexibility, offer higher bandwidth than the CPUs for data parallel algorithms [24]. The end result is that tasks are performed much more quickly and efficiently in the specialized hardware than in software.

Access to these features are often provided through a set of library functions to blend the hardware-software interface. Using these library functions, data streams or shared memories are used to create abstract connections between the application running on the general-purpose CPU and the hardware-accelerated subroutines running in the special-purpose hardware. Further, the application can continue execution on the CPU simultaneously with the subroutine being executed on the hardware enabling parallelism.

## 2.4 Myri-10G Network Interface Cards

In this thesis, we proposed a hardware-assisted UDP acceleration scheme that exploit offloading features available in modern network adapters to enable UDP protocol processing in hardware. Specifically, this scheme utilizes protocol offloading features provided by Myri-10G network interface card (NIC). We therefore present an overview of Myri-10G NIC in this section.

Myri-10G NICs [15] are new generation network interface cards provided by Myricom. Myri-10G NICs support link speed of 10 Gbps. These NICs, include a programmable Lanai processor and DMA engines. Firmware and associated software of Myri-10G have support for Ethernet and proprietary Myrinet Express protocol which is a layer 2 protocol developed to maximally utilize the processing capabilities embedded in Myri-10G cards. In our experimental setup, Myri-10G NICs were connected to the host through PCI-Express x8 bus.

Detailed specification about Myri-10G NICs can be found in [15].

Myri-10G NICs achieve high throughput performance and a low host-CPU utilization through offloading network protocol processing. Myri-10G does not support stateful offloads like the ones used by "TCP Offload Engines" (TOEs) but they support variety of stateless offloads. Some of the offloads supported are:

- IP and TCP Checksum Offload, Send and Receive
- Interrupt Coalescing
- TSO (TCP Segmentation Offload)
- LRO (Large Receive Offload)
- RSS (Receive-Side Scaling)
- Multicast Filtering

# Chapter 3

# GePSeA: Design and Implementation

In this chapter, we will present design and implementation details of the GePSeA framework. We will also discuss how GePSeA framework can be utilized to develop application-specific software accelerators. As discussed previously, software accelerators much like the hardware accelerators execute tasks assigned to them on a dedicated unit. In multi/many-core systems, accelerators execute these tasks asynchronously and efficiently on a subset of available system cores.

GePSeA framework provide infrastructure to build software accelerator. It provides generic interface for applications to interact with the accelerator in order to offload work to them in a simple and efficient manner while continuing their respective processing. GePSeA framework therefore has a generic layered architecture as shown in Figure 3.1. It is a two-layer architecture. The upper layer consists of application-specific plug-ins while the lower layer consists of more generic utilities called core components. These two layers reside between the application and the underlying network. GePSeA framework therefore provide GePSeA communication layer (not shown in the Figure 3.1) that provide infrastructure for accelerator-application communication. For parallel applications running on cluster, GePSeA communication layer provide infrastructure for inter-node accelerator-accelerator communication.

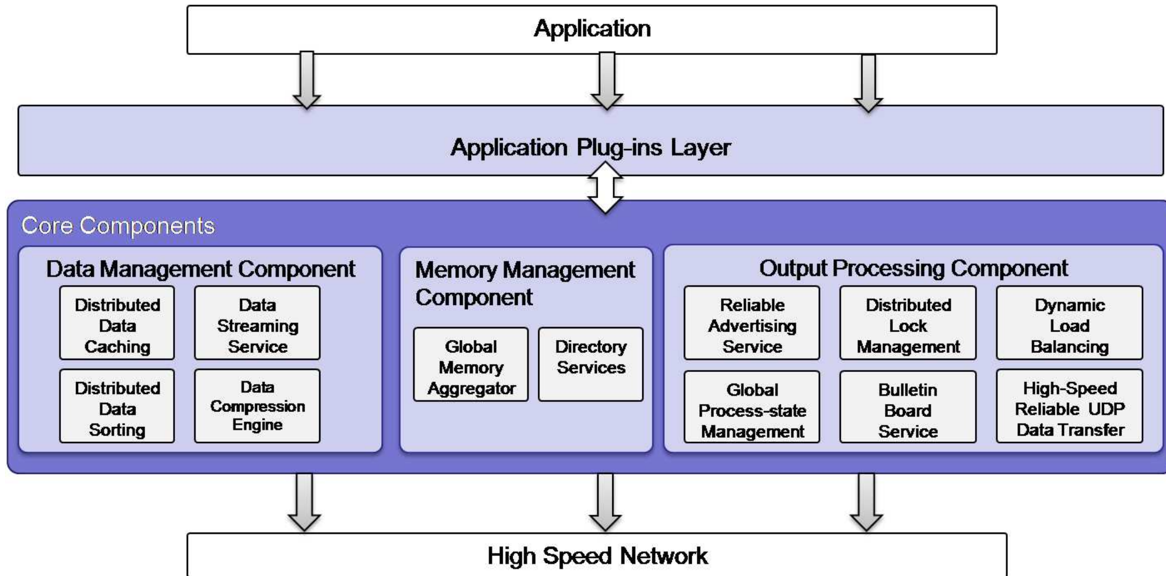


Figure 3.1: Layered Architecture of GePSeA Framework

GePSeA accelerator is a lightweight helper process to the application that executes the tasks delegated to it. Application plug-ins and the core components of the GePSeA framework are compiled together into this helper process referred to as a GePSeA accelerator. GePSeA accelerator utilizes the GePSeA communication layer to communicate to the application and to the underlying high-speed network. We now present some detailed discussion on various layers of the GePSeA framework.

## 3.1 GePSeA Infrastructure and Communication Layer

Figure 3.2 shows various components of GePSeA framework. It shows an accelerator process running on a node and servicing two application processes. Applications first need to register themselves with the accelerator running on that node before they can use it. Once accelerators receives the registration request from all the participating application processes, it sends them a registration successful message. After which, the applications can begin delegating the tasks to the accelerator. For parallel applications running on multiple nodes in a cluster, each compute node run only one accelerator process that service all the application processes

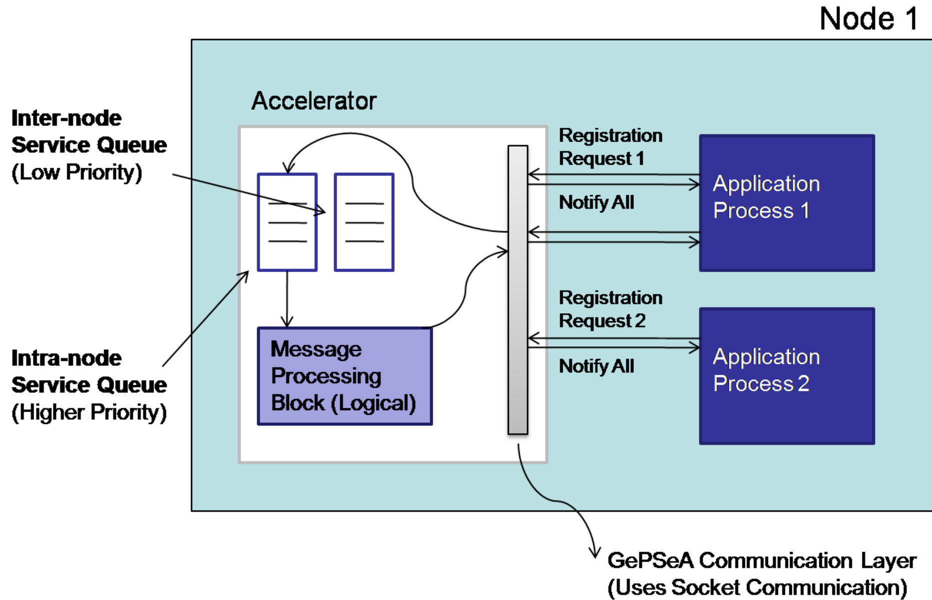


Figure 3.2: Application-Accelerator Interactions Within a Node

running on that node. Accelerator-application interactions for a three-node cluster in shown in Figure 3.3.

As we can see in the Figure 3.2, all messages passes through the GePSeA communication layer. This layer uses TCP/IP socket communication to communicate with application running on that node or to another accelerator running on some other node. It keeps up-to-date information of all participating application processes and the accelerator processes. GePSeA framework provide two different service queues to handle *inter-node* and *intra-node* service requests. If GePSeA communication layer receive service request from an application process that does not require participation from other nodes, it is queued into the intra-node service queue. All other service requests are queued into inter-node service queue. Message processing block, which basically consists of application plug-ins, first check the intra-node service queue for any available request and service them. If intra-node service queue is empty it then checks for messages in the inter-node service queue. Therefore intra-node service queue have higher priority over inter-node service queue. This is just an optimizations since intra-node service request do not require inter-node synchronization and communication and

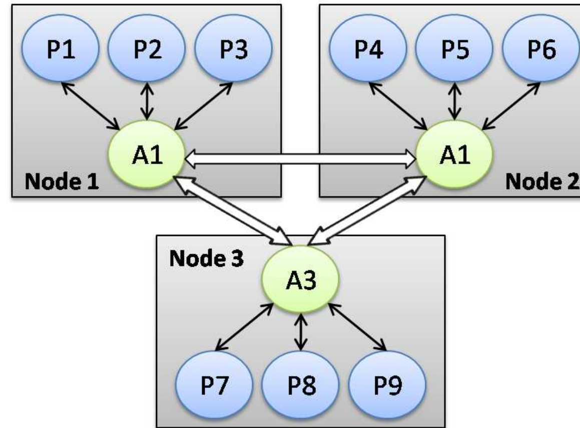


Figure 3.3: Application-Accelerator Interactions for a Three-Node Cluster

therefore can be serviced faster than inter-node service requests. However for our case study presented in Chapter 4, where we built a software accelerator for mpiBLAST application using GePSeA framework, we used a single service queue to service both intra and inter service request.

We introduced notion of two queue system later with the objective of improving the overall performance of the accelerator. However there are certain tradeoffs and issues involved with such design that needs to be addressed. Firstly, since inter-node service requests are serviced only if there are no pending service requests in intra-node service queue, this can lead to starvation for requests queued in inter-node queue. Problem of starvation can be handled by modifying the current design to incorporate the notion of *queue weights* and fetching the service requests from the two queues using weighted round-robin algorithm. Another important issue with two queue system is of deadlocks. While accelerator expects application to take care of deadlock situations, certain deadlocks might occur due to inefficiencies in the accelerator framework design. Current implementation of GePSeA does not handle such deadlock situations. Standard methods of breaking deadlocks can be employed to break the deadlocks. This work is included in our future work which is also discussed in Chapter 6 separately.

## 3.2 Application Plug-ins Layer

Application plug-ins forms the upper layer of GePSeA framework. They are built using the services core components and the GePSeA communication layer. Application plug-ins are simple and abstract application-specific utilities. Unlike the core components which are generic reusable utilities, application plug-ins are generally not reusable among different applications. In order to build application plug-in, application-specific task must be identified first that is suitable for offloading. This task can then be written as a plug-in that will execute on accelerator process using services of core components provided by the framework. Application can have any number of plug-ins.

## 3.3 Core Components

Core components are building blocks of the framework. They describe core features provided by the framework. These features, in general, are simple yet powerful tasks that can be utilized by the application plug-ins layer above it. Each core component implements a generic functionality. This generic nature of these core components allow them to be used by multiple applications. Most of the core components are independent research components in themselves. Design of some of challenging core components is described in following sections. While some of the core components such as high-speed reliable UDP incorporate new and novel concepts, the ideas of others may not be completely new. For example, there has been previous work that utilizes specialized hardware support to achieve distributed data management and locking efficiently [25, 16]. However, our this thesis work focuses on providing such useful functionality in the context of a general purpose software-based accelerator, which presents its own unique challenges.

While GePSeA is a general purpose framework that allows it to *onload* any task, we concentrate on three main categories of components, namely: (a) data management components,

(b) memory management components and (c) synchronization and coordination components as illustrated in Figure 3.1. In this section, we describe the details of each of these categories as well as some of the core components within these categories. It is to be noted that while we present software-based designs for each of these utilities, this does not preclude GePSeA from taking advantage of hardware implementations when they are available. When such hardware units are available, GePSeA would simply replace these modules with them while retaining the interface it exposes to applications. Thus, from an application's perspective, the actual implementation of these utilities does not matter.

### 3.3.1 Data Management Core Components

The data management utilities primarily deal with moving I/O data associated with the application. This includes input data, output data and transitional meta-data that is created and destroyed during application execution. We present four utilities within this category that the GePSeA framework provides: (i) distributed data caching, (ii) data streaming service, (iii) distributed data sorting and (iv) data compression engine.

#### 3.3.1.1 Distributed Data Caching Core Component

This component provides caching capability for the entire input database on the overall system memory. The input data used by many applications (terabytes in size) is typically several times larger than the amount of memory on each node (gigabytes in size). However, for large-scale systems, the total system memory is tens of terabytes in size which, on the whole, is sufficient to cache the entire input data in many cases. The distributed data caching component performs this task by trapping I/O calls, reading the entire input data into the system memory and responding to I/O requests from the distributed memory cache, instead of from the disk or file-system.

One of the primary challenges in distributed data caching is the addressability of the data.

Specifically, should the application be aware of the actual locality of the data segment or should this information be hidden and handled internally by the data caching component? Typically, for most scalable applications, I/O transactions involve moving reasonably large amounts of data (at least a few kilobytes, but most times several megabytes). Thus, the overhead added due to trapping I/O calls and automatically figuring out the location and fetching data is typically not a major portion of the overall I/O cost. Further, implicitly managing data locality makes it simpler and intuitive for applications to use this component as well. Thus, we chose to hide data locality for component. Data movement is completely handled by this component through appropriate communication that is initiated and terminated within the components and never exposed to the applications.

#### 3.3.1.2 Data Streaming Service Core Component

While distributed data caching moves data from the file-system to system memory thus reducing I/O overhead, the amount of time taken for data movement is still large, especially when the amount of computation per data unit is not much (e.g., in search algorithms). Thus, to keep the application fed with data, techniques such as pre-fetching are very useful. The data streaming service handles these aspects by swapping out unused data with fresh data that application would use. This component includes distributed coordination with other instances of the GePSeA helper agents, in order to minimize duplication of data (i.e., data is swapped between two nodes instead of replicating and utilizing more memory than needed).

Another important aspect to note in this service is that since it is completely handled by the GePSeA helper agents, such pre-fetching and swapping is done in a completely asynchronous manner without disturbing the application.

#### 3.3.1.3 Data Compression Engine Core Component

As the name suggests, this component deals with compressing/decompressing data. However, together with regular byte-stream compression, this engine also provides capabilities for application-specific compression as presented in our previous work [3]. Specifically, this engine can either view the data as a stream of bytes, or as high-level application-specific objects and converts them to meta-data that is much smaller in size. Once communicated over the network, this meta-data is converted back to the actual data.

### 3.3.2 Memory Management Core Components

Memory management utilities deal with handling system memory allowing applications with access to the entire memory in the system, rather than just the local nodes memory.

#### 3.3.2.1 Global Memory Aggregator Core Component

Operations like caching, indexing, searching, etc. can all perform better with larger memory. Since the cost of remote memory access can be and is typically much lower than the cost of disk access, these components can effectively utilize the free memory available on other nodes. The global memory aggregator efficiently allows applications to utilize the memory on the entire cluster instead of just their local memory. Specifically, this core component presents a global address space to its upper layers and maintains a mapping of the locations on the global address space with the actual node and physical address of these locations. Unlike the distributed data caching service, this component does not perform the global address translation to the actual physical address and physical node transparently; this is because memory accesses are typically much smaller in size than I/O accesses and are expected to have very little overhead. Thus, applications explicitly control and manage data placement on the system. Data movement, however, is completely handled by the global memory aggregator.

### 3.3.3 Coordination and Synchronization Core Components

For applications using a large number of processes, coordination and synchronization between the processes can be a major portion of the application execution. This category deals with utility components that improve such tasks.

#### 3.3.3.1 Dynamic Load Balancing Core Component

Load imbalance among the nodes in the cluster can result in a serious bottlenecks which can limit the scalability of parallel applications. The dynamic load balancing core component provide mechanism to balance the load on each node. We developed a simple and generic design for this core component. Not all nodes in the cluster are responsible for load balancing. A special node called *leader* is elected dynamically or chosen statically. *Leader* is responsible for work assignment to various nodes in the cluster. In order to accomplish this, it maintains a Work Allocation Table (WAT) for each type of work assignment. It also identifies a Work Unit (WU) for each type of work assignment. Work assignment type, for example, can be computational work or result merge/sort work.

Each node advertises its availability periodically to all other nodes. Up-to-date information about node's current status is maintained by global process-state core component. Dynamic load balancing core component uses this information to assign work to available nodes including itself by updating work allocation table. *Leader* also informs about the work allocation to the concerned node. Alternatively node can query *leader* about its work assignment or any other node's assignment. In order to make it more efficient, We made several optimizations to above design such as assigning more than one work unit at time to a node.

#### 3.3.3.2 Global Process State Management Core Component

Managing the data processing performed by application processes requires sharing certain information about each node, such as whether the process on that node is idle and wait-

ing for communication, what fragment of the data it currently hosts, and various others, among the different nodes in the system. The performance and scalability of applications largely depends on how efficiently such a global process-state is maintained. Thus, the global process-state management core component aims at maintaining an up-to-date and complete information about the status of the different nodes in the cluster.

#### 3.3.3.3 Bulletin Board Service Core Component

This task provides an addressable memory that can be read or written to by any other node in the cluster system. The bulletin board itself is distributed memory placed on different nodes in the system. However, from an application's perspective, this is a contiguous chunk of memory that is available to publish information. Together with efficient movement of data, this component also handles the synchronization required in order to avoid data corruption.

#### 3.3.3.4 Reliable Advertising Service Core Component

The reliable advertising service primarily deals with reliable and efficient ways of distributing information across the entire system. Where available, this task can internally utilize the unreliable multicast features provided by networks such as InfiniBand, while providing software reliability on top of it. On other architectures, such reliability is handled using reliable protocols such as TCP/IP. This task also includes various other capabilities such as protection against overwrite (i.e., two continuous messages from the same host will ensure that the first message is read by the host before the second is delivered), host transparent advertising (i.e., the remote host does not have to actively provide a buffer to receive the advertisements from other nodes), advertisement filtering (i.e., getting rid of irrelevant advertisements) and various others that need to be handled efficiently.

#### 3.3.3.5 Distributed Lock Management Core Component

A distributed lock manager allows lock-based synchronization between multiple nodes to avoid various race conditions while accessing shared resources. While it is possible to provide such locking capability using the atomic operation features provided by high-speed networks such as InfiniBand, the GePSeA helper agents can enhance these features by providing capabilities such as request queuing and group-wise shared locks, that cannot be easily provided in hardware. For environments where such networks are not available, the GePSeA agents can perform both the atomic operations as well as the request queuing and shared locks.

#### 3.3.3.6 High-Speed Reliable UDP Core Component

For long network bandwidth have been considered bottleneck for high-speed communication. But the with exponential growth in network capacity in recent years and with host processor speed not matching the growth in network speed, hosts have now become the bottleneck. And with CPU clock speed not showing sign of further improvement, the problem is only going to compound. However with the invention of multi/many-core systems, additional computational power in terms of system cores is available at the host which must be efficiently exploited. High-speed reliable UDP core component implements a novel technique to utilize available system cores to provide high-speed reliable UDP data transfer.

TCP is the *de facto* standard for Internet and wide-area computing. TCP is a *connection-oriented* transport layer protocol used for end to end reliable communication. However TCP is a “heavy-weight” protocol that support many complex features. Consequently TCP stack processing offers huge overhead consuming many host-CPU cycles making it CPU intensive task. With computing power of end hosts being limited, high stack processing overhead can significantly lower the maximum achievable throughput of data transfer thereby making the host bottleneck for high-speed network communication. Failure of TCP for the *LambdaGrids*

is also well known. LambdaGrids refers to a distributed system with large number of compute nodes, visualization systems and the storage systems connected using high bandwidth optical network and acting as one large supercomputer. Unlike wide-area networks, LambdaGrids have dedicated high-speed interconnects that are "highly" reliable.

Unlike TCP, UDP is "lightweight" since it does not support heady-duty features like packet acknowledgements, packet retransmission, congestion control etc that add overhead to the protocol processing. UDP therefore have some obvious advantages over TCP. UDP based rate control protocols and applications have gained substantial popularity in the recent years. UDP based rate controlled protocols fare better than TCP on LambdaGrids. Also UDP applications such as internet telephony, video streaming and multicast applications are widely being used today.

Reliable UDP core component this core component uses some of the underlying concepts of reliable blast UDP (RBUDP), a reliable UDP implementation that uses blast mechanism for data transfer. Therefore in order to understand the design and implementation details of this core component, we first present a short overview of RBUDP in the next section.

### Overview of Reliable Blast UDP

Reliable Blast UDP (RBUDP) [9], proposed by Eric He Jason Leigh, Oliver Yu and Thomas DeFanti, is a an implementation that provide reliable UDP communication between two end-points. It can be used as application-level protocol for reliable data transfer. However unlike TCP, a transport layer protocol, it eliminates the overhead of individual packet acknowledgements and therefore it is very effective for bulk data transfer over a "fairly" reliable high bandwidth network.

Sending and receiving hosts using RBUDP for data transfer maintains a TCP connection to send/receive control packets and a UDP socket to send/receive data packets. Sending and receiving of data is done in *rounds*. Sender encapsulates the data (payload) into UDP

datagrams and blast all of them over UDP socket to the receiver at an specified sending rate. After the last UDP datagram has been transmitted, sender sends an *end-of-round* packet, a control packet, over the TCP connection to signify the completion data-packet sending operation by the sender.

After receiving *end-of-round* packet, receiver prepares a bitmap of packets not yet received by it and send this bitmap back to the sender over TCP connection. Sender would then retransmit these erroneous packets to receiver marking the beginning of a new round. This process continues until all the packets have been received by receiver correctly.

### Design and Implementation Details

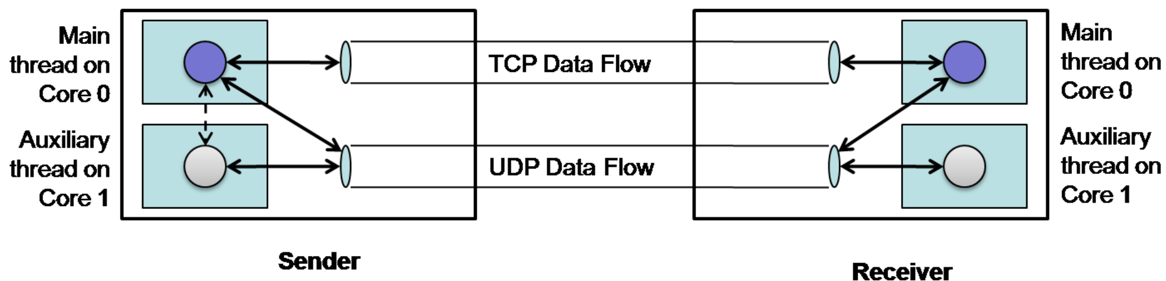


Figure 3.4: High-Speed Reliable UDP Core Component Design

High-speed reliable UDP core component is a "core aware" version of reliable blast UDP implementation. It uses multi-threading mechanism to tap the computation power of system cores to accelerate UDP based data transfers. Like RBUDP it uses a TCP connection to send/receive control packets and a UDP socket to send/receive data packets. However unlike RBUDP, this implementation use multiple user threads to send/receive data packet from UDP sockets as shown in Figure 3.4. Main thread (thread 0), spawns auxiliary threads that run on different cores. While all the threads including main thread can read/write data to UDP socket, only main thread can receive/transmit control packets using TCP connection.

```
Set number of packets to be received = Data size / packet size;
Reset receive_complete_flag;
Create threads 0 to p-1 each bound to a different system core;

For thread 0:
while receive_complete_flag not set do
    Wait for the data on both UDP and TCP socket;
    if Packet received on the UDP socket then
        Extract the packet sequence number ;
        Calculate the receive buffer offset and copy the packet to the buffer;
        Acquire the lock on the bitmap ;
        Update the bitmap by setting a bit corresponding to the packet received;
        Release the lock ;
    else
        if end_of_round packet received on the TCP socket then
            Acquire the lock on the bitmap ;
            Check the bitmap and count the number of packets not received ;
            Release the lock ;
            if Count is zero then
                Set receive_complete flag and free bitmap memory;
                Wait for all the other threads from 1 to p-1 to exit ;
                Exit this thread ;
            else
                Send the packet bitmap to the sender over TCP connection;
            end
        end
    end
end

For threads 1 to p-1:
while receive_complete_flag not set do
    Wait for the data packet on UDP socket;
    if Packet received on the UDP socket then
        Calculate the receive buffer offset and copy the packet to the buffer;
        Acquire the lock on the bitmap ;
        Update the bitmap by setting a bit corresponding to the packet received;
        Release the lock ;
    end
end
```

Figure 3.5: Algorithm for Receiving Data

```
Reset send_complete_flag; Set packet_left = Data size / packet size;
Create hash table containing sequence number of packet to be sent;
Create threads 0 to p-1 each bound to a different system core;
Create status_array[] of size p for each thread to synchronize at the end of each round;

For thread 0:
while send_complete_flag not set do
    local_packet_count = packet_left / p ;
    foreach index i from 0 to (local_packet_count-1) do
        Get the packet sequence number at index i in hash table ;
        Calculate packet payload offset in the send buffer ;
        Construct the packet and send it on UDP socket ;
    end
    Wait till all threads finish sending packets (status_array[] is reset) ;
    Send end_of_round packet to the receiver on the TCP socket;
    Wait on the TCP socket for error bitmap from receiver;
    Count the number of packets not received by their receiver from the bitmap ;
    if Count is zero then
        Set receive_complete flag and free bitmap memory;
        Wait for all the other threads from 1 to p-1 to exit ;
        Exit this thread ;
    else
        packet_left = Packets not received by receiver ;
        Update the hash table for packets not received by receiver ;
        Set each element in status_array[] to 1;
    end
end

For threads 1 to p-1:
while send_complete_flag not set do
    if status_array[thread_id] set then
        local_packet_count = packet_left / p ;
        foreach index i from thread_id* local_packet_count to thread_id *
        local_packet_count-1 do
            Get the packet sequence number at index i in hash table ;
            Calculate packet payload offset in the send buffer ;
            Construct the packet and send it on UDP socket ;
        end
        Reset element at index thread_id to 0 ;
    end
end
```

Figure 3.6: Algorithm for Sending Data

Acceleration comes from the fact that multiple threads running on different cores can read-/write UDP socket simultaneously. Due to this processing of UDP packets is done in parallel. Since a read system call on UDP socket reads exactly one UDP packet of specified size at a time, problems such as same packet being read multiple times or a packet being read partially by multiple threads do not arise. However synchronization among the threads is needed to correctly record data packets sent over the UDP socket. For example, at the receiver, the error bitmap must be locked before thread updating it (each bit in bitmap represents a packet and therefore bit corresponding to the packet received must be updated in the error bitmap only by the thread that received the packet) in order to correctly count the packets received. Similarly at the sender, each thread must synchronize with the main thread after each round in order to avoid unnecessary retransmission of data packets. Details of algorithm is given in Figure 3.5 and Figure 3.6.

To evaluate the efficacy of this core component, we present a case study in Chapter 5. We compare performance of RBUDP based file transfer application using "core aware" version provided by this core component with the hardware-assisted UDP acceleration technique discussed in Chapter 5. Experimental analysis presented in Chapter 6 show that application performance using this core component is comparable to the hardware-assisted technique.

## 3.4 Accelerator to Core Mapping

As discussed above, software accelerators can be very useful when run on subset of available cores. However mapping accelerator or even application process to an appropriate system core for optimal performance can be a challenging problem. This is because all cores in the system are not "equal" and that the load on core changes dynamically. For example, when load on the cores changes at runtime (say due to scheduling of a new CPU-intensive process by operating system on the core on which accelerator is already running), mapping accelerator to a new core dynamically is not easy. However such mapping may be critical

to performance of accelerator. Dynamic accelerator to core mapping is out of scope of this thesis. But we intend to pursue this problem in future. Existing work on dynamic process-to-core mapping such as SyMMer framework proposed in [20] can be utilized by GePSeA.

In this thesis, we performed experiments by statically binding application process and accelerators to core using `physcpubind` utility available on NUMA machines. In chapter 6 we show various combination of process to core mapping and we observe subtle difference in performance of mpiBLAST application in each case.

## 3.5 Discussion

We developed these core components with the view that these represent generic utilities that are frequently used by parallel applications. It is therefore up to the application to make use of these core components as per its need. Not all core components may be used by the application. Similarly some applications may require core components not yet part of GePSeA framework. However such core components can easily be added using the GePSeA infrastructure.

# Chapter 4

# Case Study: mpiBLAST over GePSeA Framework

To demonstrate the capabilities and efficacy of GePSeA framework, in this chapter, we describe a case study using an open source computational biology genetic sequence-search application called mpiBLAST. We present a brief overview of mpiBLAST in Section 4.1 followed by the description of the integration of mpiBLAST with GePSeA in Section 4.2.

## 4.1 mpiBLAST Overview

mpiBLAST [7] is one the most popular genetic sequence-search applications used in computational biology. It internally uses the NCBI BLAST toolkit [1], the de facto "gold standard" for sequential pairwise sequence-search that is ubiquitously used in biomedical research. The BLAST tool searches one or multiple input query sequences against a database of known nucleotide (DNA) or amino acid sequences. A similarity score is calculated for each close match based on a statistical model. The similarity of the comparison is measured by the match with the highest score. As a result, the sequences in the database that are most

similar to the query sequence are reported, along with their matches scored beyond a certain threshold. Therefore, the BLAST process is essentially a top-$k$ search, where $k$ can be specified by the user, with a default value of 500.

The core of the mpiBLAST algorithm is based on *database segmentation*. Before the search, the raw sequence database is formatted, partitioned into fragments, and stored in a shared storage space. mpiBLAST then organizes parallel processes into one master and many workers. The master breaks down the search job in a Cartesian-product manner and maintains a list of unsearched tasks, each represented as a pair of a query sequence and a database fragment. Whenever a worker becomes idle, it asks the master for a unsearched task and copies the needed fragment to its local disk (if the fragment has not been cached locally) and performs a BLAST search on its assignment. Upon finishing a task, a worker reports its local results to the master for centralized result merging. Once the results of searching a query sequence against all database fragments have been collected, the master calls the standard NCBI BLAST output function to format and print out results of this query to the output file. By default, those results contain the top 500 database sequences with highest similarity to the query sequence along with their matches. The above process repeats until all tasks have been searched. With database segmentation, mpiBLAST can deliver superlinear speed-up when searching sequence databases larger than the memory of a single node. In recent developments, mpiBLAST has evolved to use a more scalable parallel approach that allows different queries to be concurrently searched as well [10].

mpiBLAST like most parallel sequence search algorithms follow scatter-search-gather model. The *scatter* stage consists of query and/or database segmentation. In the *search* stage, each worker searches the query against the assigned database portion. Finally, the *gather* stage consists of merging of output results from individual workers.

## 4.2 mpiBLAST over GePSeA

We developed a software accelerator for mpiBLAST application using the GePSeA framework. Figure 4.1 shows various components of GePSeA framework used by accelerator of the mpiBLAST application. As described in the design section, to develop software accelerator, application plug-ins are be designed first using the core components and the GePSeA communication layer. Application plug-ins are designed for tasks that can be offloaded by application to the accelerator.

We developed three important plug-ins for mpiBLAST application using GePSeA infrastructure. These three plug-ins namely, asynchronous output consolidation plug-in, runtime output compression plug-in and hot-swap database fragments plug-in forms the upper layer of GePSeA as shown in Figure 4.1. Using these plug-ins mpiBLAST can offload considerable amount of work to accelerator. Though we developed these plug-ins for mpiBLAST, we believe they may be usable by other sequence search applications that follow scatter-search-gather model as described in the previous section.

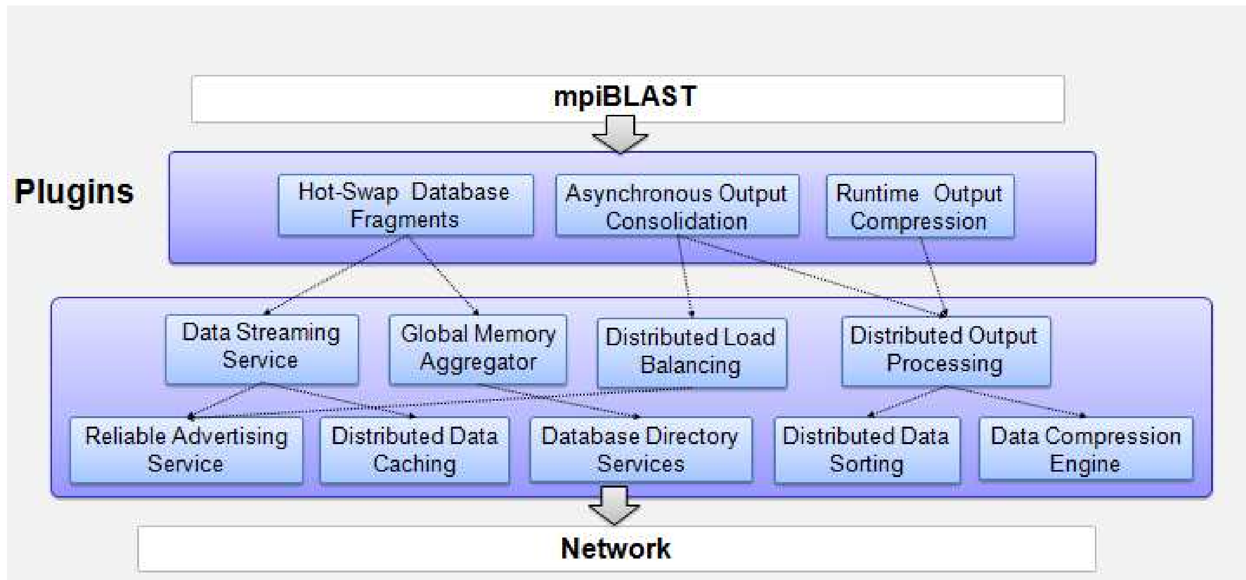


Figure 4.1: mpiBLAST over GePSeA Framework

### 4.2.1 Asynchronous Output Consolidation Plug-in

This plug-in utilizes the processing capability of the accelerator to merge/sort the data that is distributed across all multiple nodes in parallel. This is an important capability since result merging can be done asynchronously by the accelerators while the applications can continue their respective application processing. For example, if the first node has gotten its results earlier than the other nodes, it does not have to wait for till the others are done to perform the merge. Instead, it can hand over this data to the accelerator; accelerator can wait for the other nodes and sort the data incrementally as the other nodes finish their task.

To remove the bottleneck due to single writer, each accelerator has the capability to write the output results directly to the output file on a shared storage. Accelerator writes the results into a separate file for each query assigned to it. After all the queries have been processed, the result files are sorted and merged into a single output file.

As we can see from Figure 4.1 that this plug-in utilizes the services of many core components namely distributed data sorting, distributed load balancing, reliable advertising service and distributed output processing. Dependencies among various core components is also shown in the figure. As we noted earlier, we only do the load balancing for result merging and sorting tasks. Dynamic load balancing core component has the capability to balance computational load (i.e. sequence searching) also. However since mpiBLAST already uses a special node called *scheduler* to do assignment of computational loads and therefore this task is not handled by the accelerator.

### 4.2.2 Runtime Output Compression Plug-in

The data compression engine provides capabilities to compress the actual data. We conducted tests on the compressibility of the BLAST output and found that when the output was in the standard pairwise alignment text format that the output could be compresses to less than 10 percent of its original size using gzip. This is mainly due to the redundancy

found in the BLAST output. The runtime output compression can be used to compress the output before transfer thereby significantly reducing the transfer time.

As we can see from Figure 4.1 that this plug-in utilizes the services of distributed output processing and data compression core components.

### 4.2.3 Hot-Swap Database Fragments Plug-in

Currently, in sequence search applications, the database is pre-partitioned into a number of fragments that are distributed over different nodes. Normally worker node searches the database it holds. However balancing the work load on each worker may require a node to hot-swap the portion of the database it is currently processing with the other portions on-demand. This feature swaps the database fragments asynchronously across the nodes while hosts can continue their respective application processing.

As we can see from Figure 4.1 that this plug-in utilizes services of data streaming, global memory aggregator, distributed data caching, directory services and reliable advertising service core components. Dependencies among these core components is also shown in the Figure 4.1.

# Chapter 5

# Case Study: RBUDP File Transfer over GePSeA Framework

In this chapter, we present another case study to exclusively evaluate the efficacy of high-speed reliable UDP core component provided by the GePSeA discussed in Section 3.3.3.6. Since mpiBLAST application do not use any UDP based network communication, we need another application to evaluate this core component. We use reliable blast UDP (RBUDP) based file transfer application that uses blast mechanism to reliably transmit and receive UDP data. We present an overview of this application in next section.

In order to make a "fair" comparison of UDP acceleration achieved by the high-speed reliable UDP core component and the hardware-assisted UDP acceleration, we present a novel technique to exploit protocol offloading features available in latest generation networks adapters to offload UDP protocol processing. Technique presented in Section 5.2 utilize the hardware for UDP acceleration.

## 5.1 RBUDP File Transfer over GePSeA

RBUDP based file transfer application uses RBUDP application-level protocol discussed in Section 3.3.3.6 to send/receive bulk UDP data between two endpoints. Here *file transfer* refer to the process of transmitting the data buffered in the main memory (RAM) of the sender and storing the data received by the receiver into its main memory. RBUDP based file transfer application guarantees reliable delivery of data from sender to the receiver like other file transfer applications such as application using File Transfer Protocol (FTP).

RBUDP protocol performs well on dedicated networks for bulk data transfer. Therefore we used this application to send/receive large data files (up to 1 Gigabytes) on dedicated 10 Gbps link. We evaluated the performance of this application using the high-speed reliable UDP core component. Unlike mpiBLAST case study, we did not build application-specific plug-ins in order to utilize high-speed reliable UDP core component as this core component is a "core aware" extension of RBUDP and implements its core functionalities. Therefore we do not need application plug-in to take advantage this core component. It can be directly used by the applications. Also unlike mpiBLAST, in this case study we utilize only one core component provided by the GePSeA framework.

## 5.2 Hardware-Assisted UDP Acceleration

In this section, we present the scheme to exploit the offloading features available in latest generation network adapters, such as Myricom's Myri-10G NIC, to offload UDP protocol processing thereby accelerating UDP based data transfer. We describe the design and implementation details of this novel technique along with important concepts used by it.

### 5.2.1 Protocol Offload Engines

Continued focus on reducing the overhead of TCP/IP protocol processing ultimately led to the idea of having a specialized hardware called "TCP Offload Engine" or TOE. TOE implements TCP/IP stack in hardware. Benefits of having such specialized hardware are two fold. First, host operating system can be bypassed completely thereby reducing the host-CPU utilization considerably. With availability of "extra" CPU cycles, application can perform more computation and thus improve its performance. Secondly, hardware implementations are optimized for protocol stack processing and handle multiple network messages simultaneously. This reduces number of system interrupts considerably.

TCP offload engines support *stateful* protocol offloading. They have a dedicated processor that can efficiently perform stack processing keeping track of state of each TCP connection. Many latest generation network adapters like Myri-10G however ***do not*** support stateful protocol offloading like TOE. There are many reasons for these network adapters not to support stateful protocol offloading which "essentially break's the protocol stack". Detailed discussion on this can be found at [17]. However these modern network adapters *do* support *stateless* protocol offloading such as send/receive checksum offloading, TCP segmentation offloading (TSO), large receive offloading (LRO), interrupt coalescing. Such hardware features must therefore must be utilized when available.

However as discussed in Section 3.3.3.6 , UDP may be preferred network protocol over TCP by many applications. But unlike TCP, UDP offload engines (UOE) does not exist. Most new generation network adapters do not support UDP offloading. UDP protocol processing mainly include tasks like datagram fragmentation/reassembly and checksum calculations for UDP datagrams. However network adapters that support stateless protocol offloading for TCP can be tricked to support UDP offloading. In the next section, we present an important technique to exploit hardware for offloading UDP protocol processing.

### 5.2.2 High Performance Sockets

High performance sockets [26] were first proposed by V. Vishwanath, P. Balaji, W. Feng, J. Leigh and D.K. Panda. We have contributed by adding support for many socket system calls such as `select()`, `close()` that are not supported by existing implementation of high performance sockets. We also extended the idea to offload UDP protocol processing to Myri-10G network adapters. Unlike Chelsio T110, used by them to implement UDP offload engine, Myri-10G NIC is not a TOE but support stateless protocol offloading.

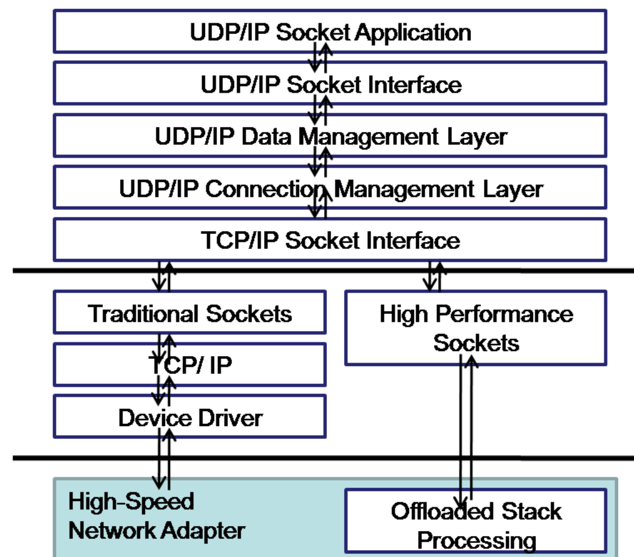


Figure 5.1: High Performance UDP/IP Sockets Architecture

The following description of high performance sockets architecture and the Figure 5.1 have been adapted from [26]. Under this technique, a pseudo UDP layer (UDP/IP Connection Management Layer (CML) and UDP/IP Data Management Layer (DML)) is developed between the application and traditional TCP/IP socket interface as shown in Figure 5.1. This pseudo UDP layer enables UDP/IP applications to transparently use hardware features supported for TCP/IP applications by modern network adapters. Specifically, these layers creates and manages TCP/IP connections for each UDP/IP endpoints (endpoints communicating over UDP) in a way that is transparent to the application. Therefore when application calls `sendto()/recvfrom()` on a UDP socket, the request is handed over to

CML which converts it to corresponding `send()/recv()` on a TCP socket. If no such TCP connection already exist with the other endpoint, CML attempts to create it first. The send/receive data is temporarily buffered and processed only after CML has established a TCP connection between the endpoints. DML layer that sits on top of CML, handles the error cases returned by CML.

Using above technique, all the protocol offloading features available in hardware for TCP can now be exploited by UDP applications. UDP applications can particularly benefit from TSO/LRO and checksum offloading feature implemented in hardware. We will evaluate the efficacy of high performance sockets in Chapter 6.

### 5.2.3 Overview of Existing TCP/IP Implementation

Previous section described how UDP applications can benefit from hardware support for TCP protocol processing. However by converting "UDP connections" to TCP connections, we also inherit various TCP drawbacks discussed in Section 3.3.3.6. Therefore we made several changes in the existing Linux TCP/IP stack with the objective of reducing the overhead associated with TCP/IP stack processing and *mimic* it like UDP stack while taking advantage of hardware offloading at the same time. In order to understand the modifications we made to TCP/IP stack, we first briefly look at the existing TCP/IP stack in this section. Existing TCP/IP flow is show in blue in the Figure 5.2.

When an application does application-level read for socket of type `SOCK_STREAM`, kernel makes system call `sys_read()`, which passes the call to `do_sock_read()`, then to `sock_rcvmesg()` and finally to `tcp_recvmsg()`. Thus `tcp_recvmsg()` is the TCP function called on an application-level read on a TCP socket. On a packet arrival at a receiver's interface, the following events occur.

- Network interface card put the received packet in the main memory through a DMA operation and interrupt the processor.

- Device driver code creates `sk_buff` structure for the incoming packet. The function `net_rx action()` then de-multiplexes the packet. For IP packets, the `ip_rcv()` function is called.

- For TCP packets `tcp_v4_rcv()` function is registered to the IP layer to receive TCP packets. For an existing TCP connection `tcp_v4_rcv()` function does a lookup for `sock` structure (hereafter referred to as `sk`).

- If it is not locked, the `tcp_v4_rcv()` function calls the `tcp_prequeue()` function to see if the packet can be added to the prequeue.

- The packet is either added to the prequeue or the `tcp_v4_do_rcv()` function is called to handle the packet. The `tcp_v4_do_rcv()` function calls various receive-handler functions, depending on the state of TCP connection. It calls `tcp_rcv_established()` if the state is `TCP_ESTABLISHED` otherwise `tcp_rcv_state process()` is called.

- The `tcp_v4_do_rcv()` puts the in-order received packet into the receive queue (`sk->sk-_receive_queue`) for FAST PATH processing. Out-of-order packets are placed in (`tp->out_of_order_queue`), for SLOW PATH processing. It also calls `tcp_ofo_queue()` to see if packets from the out-of-order queue can be moved to the receive queue.

- When a packet is queued into prequeue or the receive queue, the receiving user process is woken up by calling `wake_up_interruptible()`.

Similarly on an application-level write of the socket file descriptor, kernel makes the system call `sys_write`. For socket of type `SOCK_STREAM`, the `sys_write` finally ends up calling `tcp_sendmsg` function. Following operations occur at the sender when transmitting the packet.

- Inside `tcp_sendmsg()` function, tail of the socket write queue `sk->sk_write_queue` is checked first. If there is no socket buffer in the write queue or if there is no space

available in the tail socket buffer to append new data, a new socket buffer (`sk_buff`)is allocated using `sk->sk_stream_alloc_skb()` function. Separate pages each of size 4KB are allocated to hold the payload.

- Socket buffer `sk_buff` is then queued to socket write queue to be transmitted later. Actual transmission of the packet of the queued packet is governed by the sliding window mechanism.

- TCP uses kernel timers to trigger various packet transmission events. Packet is transmitted by calling `tcp_transmit_skb()` TCP function. In this function, TCP header for the packet is created. Packet is cloned using `skb_clone()` and cloned packet is given to the IP layer (by calling `sk->tp->af_specific_queue_xmit` which points to `ip_queue_xmit()`) for further processing.

- Retransmission timer is set to trigger packet re-transmission event (`tcp_retransmit_timer()`) if the packet acknowledgement is not received before the timeout value. Once the packet acknowledgement is received from the sender, `tcp_timer()` is called to dequeue the socket buffer from the socket write queue and free the buffer.

- After the packet processing is completed at the IP layer, it is given to the device for the final transmission by a call to `dev_queue_xmit()`. At a later stage, link layer header is prepended to the queued buffer and device-specific hard transmit routine is called to transmit the frame.

### 5.2.4 Modified TCP/IP Flow

As mentioned before, We have simplified the TCP flow to remove packet acknowledgements, congestion control, sliding window mechanism and disabling Nagle algorithm. Simply put, we have modified TCP to “behave” like UDP. Modified flow is shown in red in Figure 5.2. However we encapsulate a valid TCP header (containing a valid sequence number) to the segments processed by this modified implementation. Advantage of doing this is to fake the

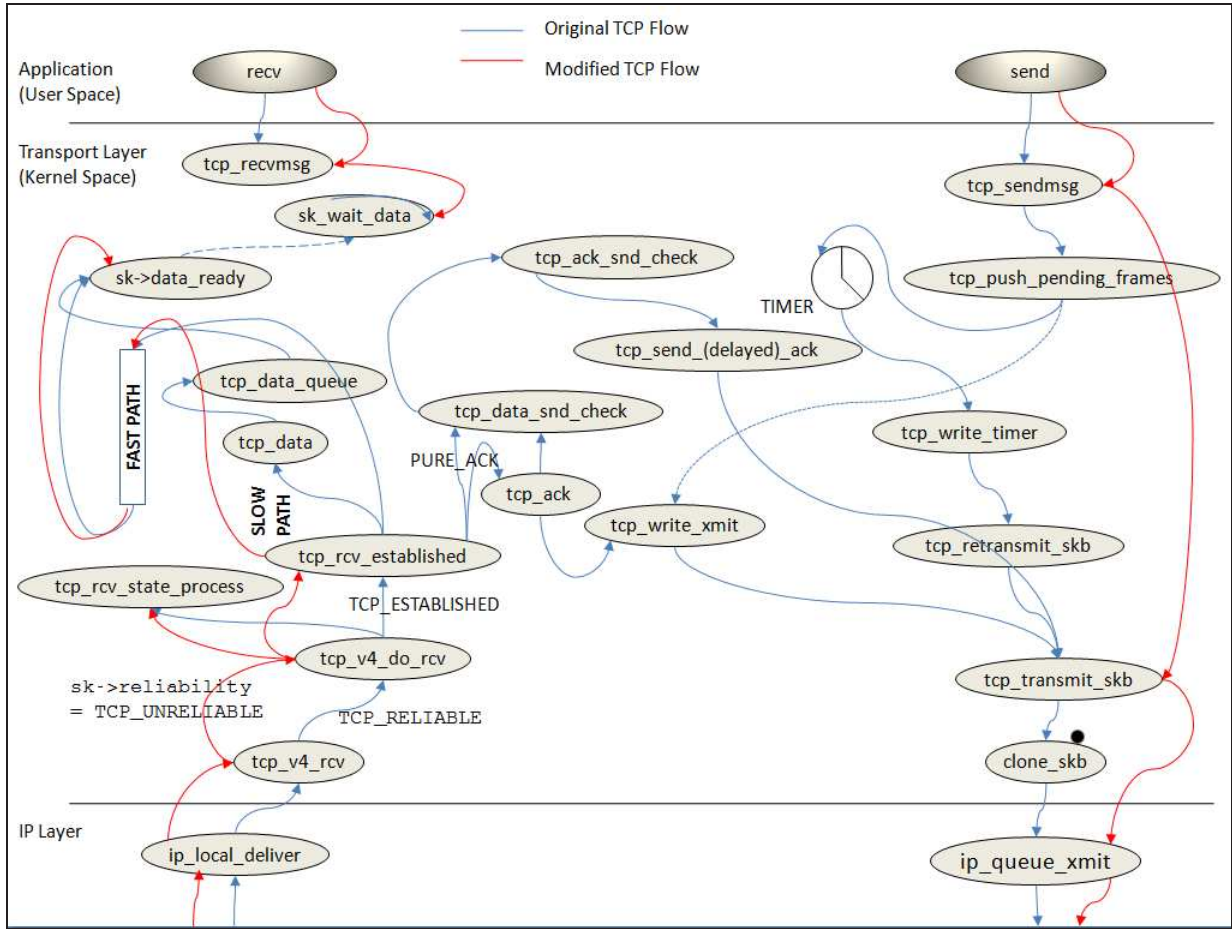


Figure 5.2: Simplified TCP Flow Diagram

network adapter in assuming that the packet belongs to a valid active TCP connection and therefore benefit from the packet processing done in the hardware for TCP. In the modified flow, at the sender end, packet is queued directly for transmission (without cloning it) after `tcp_sendmsg` is called. Similarly at the receiver end, we only follow FAST PATH processing with out-of-order packets being dropped.

It should clear from above discussion that this modified TCP flow does not guarantee reliable transmission of packets since packet acknowledgement is disabled. Therefore hereafter we will call this new modified TCP as *unreliableTCP*. Therefore applications using *unreliableTCP* should provide reliability feature at the application level if they need it. For example, RBUDP based file transfer application that we used as a case study provide reliability at

the application level for UDP applications.

While some network applications running on the host may use *unreliableTCP*, others may still be using original TCP implementation. Therefore one challenge is to uniquely identify the *unreliableTCP* connections and process them using modified TCP flow without disrupting the connections using original TCP implementation. In order to identify *unreliableTCP* connections, we added an extra field `sk_reliability` to `sock` structure defined in file `net/sock.h`.

We also introduced two macros `TCP_RELIABLE` and `TCP_UNRELIABLE` in file `net/tcp.h`. When a STREAM socket is created and initialized `sk_reliability` is set to `TCP_RELIABLE` by default (in `tcp_v4_init_sock` defined in `net/ipv4/tcp_ipv4.c`). We added a new TCP socket option (number 15) to set the value of `sk_reliability` from application. Therefore applications using *unreliableTCP* should set `sk_reliability` value to `TCP_UNRELIABLE` through `setsockopt` library function as shown below.

```
int value = TCP_UNRELIABLE;
setsockopt(fd, SOL_TCP, 15, (void*) &value, sizeof(value);
```

Chapter 6 presents experimental evaluation of this hardware-assisted technique to accelerate UDP application. We also analyze the benefits of using high performance sockets and simplified TCP flow.

# Chapter 6

# Experimental Evaluation and Analysis

In this chapter, we present experimental evaluation and results analysis of the GePSeA framework for mpiBLAST application and RBUDP based file transfer application as described in Chapters 4 and 5 respectively. While mpiBLAST application used many GePSeA core components we developed, RBUDP based file transfer application uses only one GePSeA core component namely high-speed reliable UDP core component. However, not all GePSeA core components that we developed can be evaluated using these two applications. Experimental evaluation of mpiBLAST over GePSeA and RBUDP file transfer over GePSeA is presented in Section 6.1 and 6.2 respectively.

## 6.1 Evaluation of mpiBLAST over GePSeA Framework

In this section, we evaluate the accelerator for mpiBLAST application and do a detailed performance analysis.

### 6.1.1 Experimental Setup

For all experiments in Section 6.1, we used the GenBank `nr` database, a protein type repository frequently searched against by bioinformatics researchers. The size of the raw `nr` database is nearly 1 GB, consisting of 1,986,684 peptide sequences. We used ICE cluster residing at Synergy Lab at Virginia Tech for our experiments. ICE cluster has 9 compute nodes each of which is equipped with two Dual-Core AMD Opteron Processor 2218. Therefore each ICE node has total of 4 cores. Memory and cache size on each node is 4 GB and 1024 KB respectively. ICE cluster uses 1 Gbps Ethernet interconnect.

For our experiments we pre-partitioned the `nr` database into 8 fragments using `mpiformatdb` utility. For most experiments the input query sets containing different number of sequences were chosen randomly from `nr` database. For some experiments pseudo-random query sets were chosen in order to have a better control over the output size and to better analyze the efficacy of specific GePSeA core components. Our experimental methodology include running mpiBLAST application first the with accelerator and then without it on a given set of ICE nodes for same input query set.

### 6.1.2 Accelerator on "Committed" Core

By "committed" core we mean a core that has already been committed to some application. Figure 6.1.2 shows per-node configuration for this experiment. As we can see in Figure 6.1.2, we run one worker process on each of the 4 cores of the node with each worker process explicitly bound to a core using `physcpubind` utility available on NUMA machines. Since each worker process runs on a separate core, each ICE node can run maximum of 4 worker processes. Host operating system runs accelerator (one per node) on any one of 4 cores of the node as per its scheduling policy. We run this experiment for different number of worker processes; 8, 16, 24 and 36 respectively. For this experiment 300 input query sequences were randomly chosen from the `nr` database. As mentioned earlier, we first run mpiBLAST

application without accelerator and later with accelerator and note the application execution time in both cases.

Figure 6.2 shows speed-up obtained by running mpiBLAST with accelerator compared to not using the accelerator. Our results show significant performance improvement when accelerator is used. With 36 workers we see as much as 205% (2.05x) speed-up. It increases with increase in worker processes. Our analysis show that speed-up is obtained primarily due to the overlapping of worker computation and communication tasks. While worker nodes perform computation (searching query against the database), accelerator takes cares of transmitting the results of the previous query to destined node thereby overlapping communication and computation. Offloading tasks such as merging results and writing it to the output file remove the bottlenecks and drawbacks of having single writer. Therefore by using accelerator, both query search by workers and result merging by accelerators can be done in parallel thereby reducing the overall execution time.

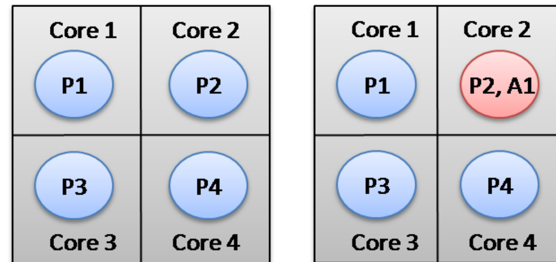


Figure 6.1: Node Configuration for Experiment 6.1.2

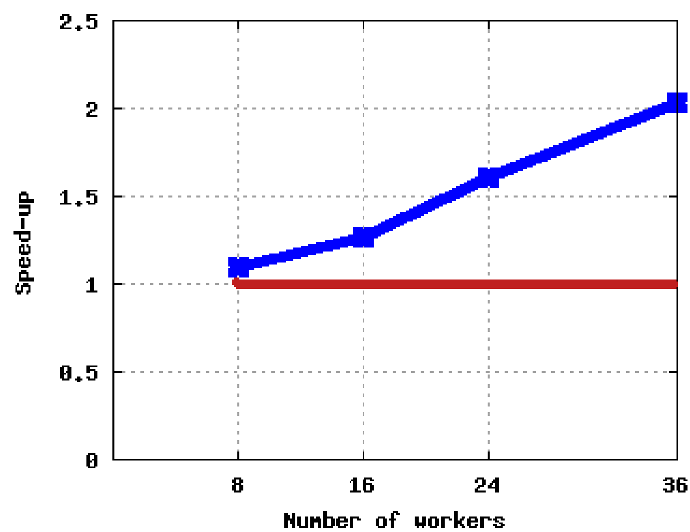


Figure 6.2: Speed-up Obtained by Running Accelerator on Committed Core

We note that worker process typically use nearly 100% of CPU time on which it is running. Core on which both worker and accelerator run, CPU cycles are shared between the two. In this context, results shown in Figure 6.2 are particularly interesting since we are running accelerator on an oversubscribed core (against running it exclusively on an available core thereby committing it additional system resources) and still observe significant performance improvement.

### 6.1.3 Accelerator on "Available" Core

By "available" core we mean core that is not running any worker process and exclusively available for accelerator to run. In this experiment, we run 3 worker processes on three different cores of an ICE node while the accelerator runs on the fourth "available" core. Therefore for 9 ICE nodes, we will have 27 worker process running on different cores with one accelerator running on each node on a separate core.

This case is similar to the precious case except that in this case accelerator run on a core that is exclusively available to it and therefore does not share CPU cycles with any other process. As we can see from the results shown in Figure 6.4 that significant speed-up is obtained in this case also. For 27 workers running on 9 ICE nodes, we observed close to 170% (or 1.7x) speed-up. We noticed that while the CPU utilization of worker process is nearly 100%, CPU utilization of accelerator is only between 2% to 5%. Therefore running accelerator exclusively on a core result in under-utilization of system resources.

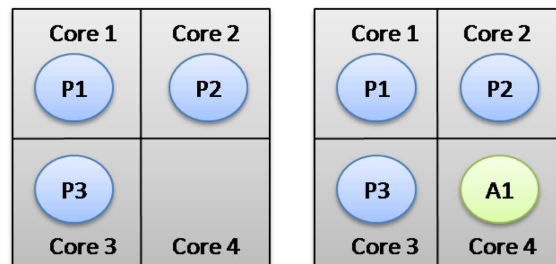


Figure 6.3: Node Configuration for Experiment 6.1.3

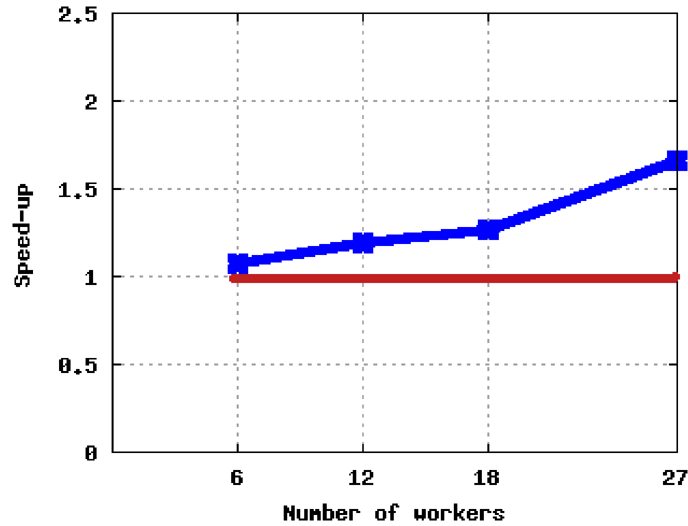


Figure 6.4: Speed-up Obtained by Running Accelerator on Available Core

### 6.1.4 Accelerator with Unequal Workers

We use the data previous two experiments to further analyze the effect of running accelerator with unequal workers. Node configuration for this case is shown in Figure 6.1.4. In first case, mpiBLAST is run without accelerator with 4 worker processes running on 4 different cores on a node. In second case, mpiBLAST is run with accelerator running on an "available" core like in Section 6.1.3, 3 worker processes run on 3 different cores of an ICE node while the accelerator runs on the fourth "available" core.

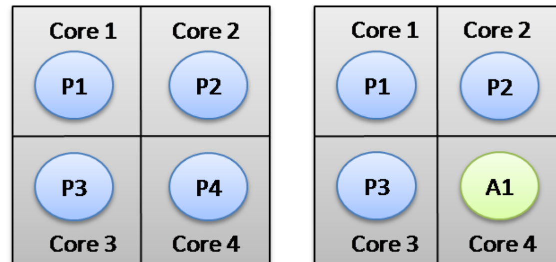


Figure 6.5: Node Configuration for Experiment 6.1.4

This case is similar to the previous other two cases except that we run one less worker per node in this case. Running this experiment on 9 nodes will have 36 worker processes in first case and only 27 worker process in the second case. As we can see in Figure 6.6 that even with one worker less per node, we still see speed-up as much as 140% (1.4x) when run on all 9 ICE nodes. Refer to Figure 6.6. However as stated earlier, running accelerator

exclusively on a core results in under-utilization of computation power of the core as it remains idle for most of the time. Therefore running accelerator on an "committed core" is better than running it on an "available core".

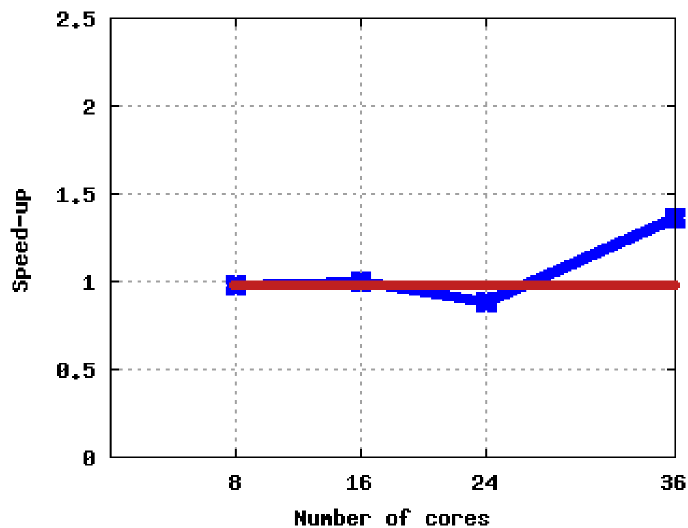


Figure 6.6: Speed-up Obtained by Running Accelerator for Unequal Workers

### 6.1.5 Increase in Problem Size

For this experiment and all the other experiments hereafter in this section, we use node configuration shown in Figure 6.1.2 which is the node configuration for running accelerator on a "committed node". In this experiment, we observe steady reduction in application running time using accelerator with the increase in the problem size. These results are along the expected lines since with the increase in the problem size, result merging and writing to the output file becomes the bottleneck in mpiBLAST which is eliminated using the accelerator. Refer Figure 6.7 for the results.

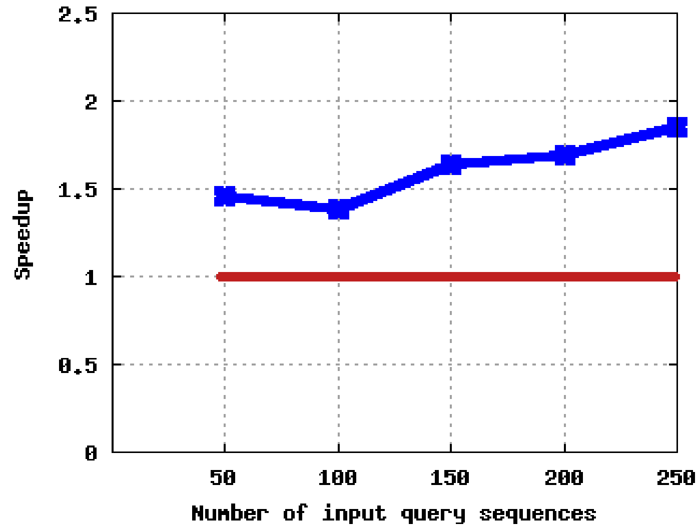


Figure 6.7: Speed-up Obtained with Increase in Problem Size

### 6.1.6 Worker Search Time

We analyzed the variation of *search time* and the *non-search time* of each worker with increase in number of worker processes. For mpiBLAST, search time refers to the computation time i.e. time a node spends searching a query against a database fragment while non-search time refers to the non-computation time. We observed that for a large input query set, percentage of *search time* of a worker to total worker time decreases rapidly from 92.2% to nearly 71% as shown in Figure 6.8. However with accelerator, worker spent more than 99% of time performing the computation.

### 6.1.7 Distributed Output Processing Core Component

In this experiment we specifically analyzed the efficacy of distributed output processing core component of GePSeA used by the accelerator. In this experiment we first run mpiBLAST application with result consolidation being done only one accelerator assigned statically. Then we run mpiBLAST application with result consolidation being done being divided among the accelerator equally again assigned statically. As we can see from the results

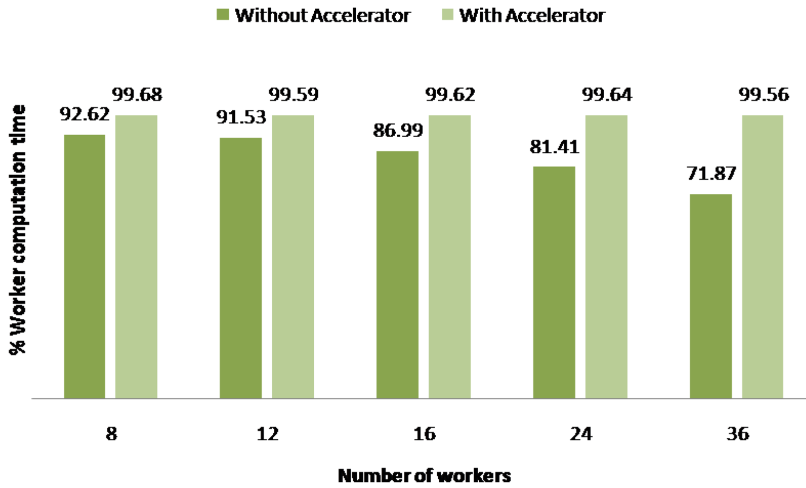


Figure 6.8: Worker Search Time as a Percentage of Total Time

shown in Figure 6.9, we observe significant reduction in the later case since accelerators are more effectively utilized in the later case.

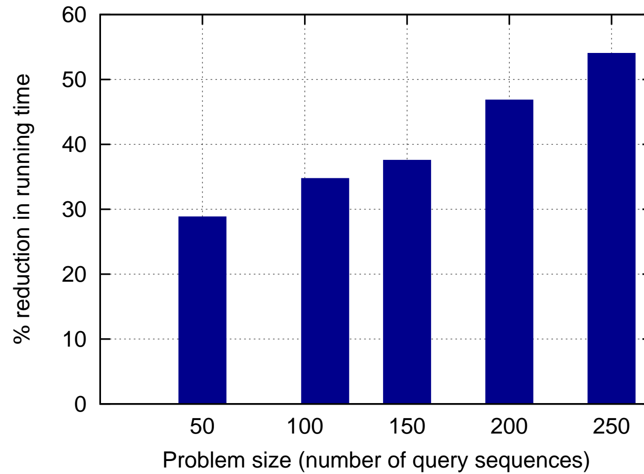


Figure 6.9: Distributed Output Processing Feature of GePSeA

### 6.1.8 Dynamic Load Balancing Core Component

We tested dynamic load balancing feature of GePSeA used by the accelerator as described in Chapter 3. We compared dynamic allocation of result merging and writing assignments

to the accelerators against the static allocation of result merging and writing assignment. In static allocation, each accelerator is assigned equal number of *work units* statically while in dynamic allocation number of *work units* assigned to accelerators vary depending on the time needed to service a particular *work unit* which is known only at run time. We see an average of 14% of improvement as shown in the Figure 6.10. With highly "uneven" queries this difference could be very high.

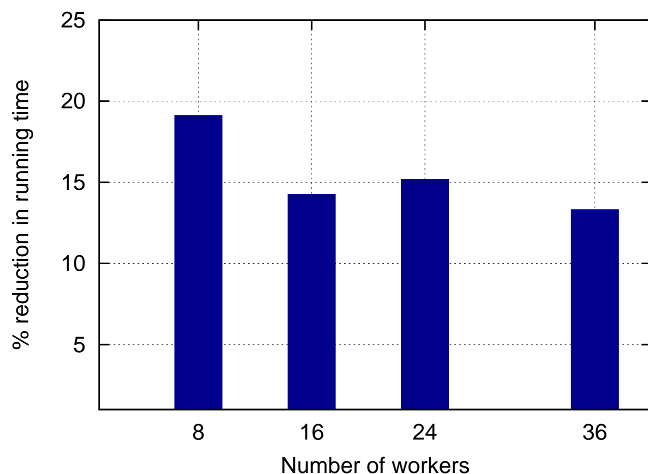


Figure 6.10: Dynamic Load Balancing Feature of GePSeA

### 6.1.9 Data Compression Core Component

We tested compression feature provided by GePSeA and used by the accelerator. Negative values in Figure 6.11 shows an increase in the running time of mpiBLAST application using this core component. This is contrary to our expectations. However, for compression at run time to be effective, network latency must exceed the time required to compress and uncompress the data. We believe that size of result data set generated by our experiments is not large enough to test for compression core component to make a positive impact on the overall running time. However we do observe a positive trend of decrease in application running time with increase in the worker processes.

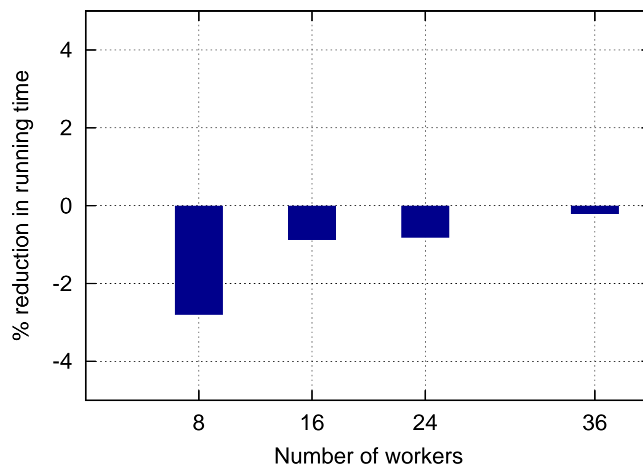


Figure 6.11: Data Compression Feature of GePSeA

## 6.2 Evaluation of RBUDP File Transfer over GePSeA Framework

In this section, we evaluate high-speed reliable UDP core component and the hardware-assisted UDP acceleration scheme discussed in Chapter 5 for a RBUDP based bulk file transfer application.

### 6.2.1 Experimental Setup

Experiments were conducted on two directly connected hosts. Each host is equipped with Myri-10G network interface cards that support link speed of 10 Gbps. Each host has two dual-core AMD Opteron 2118 processors with 4 GB of memory and 1024 KB of cache. We used MTU size of 9000 bytes supported by Myri-10G NIC for our experiments.

Experimental methodology include transmitting the data buffer stored in main memory (calling it file here) of the sending host and storing the received data into the main memory of receiving host. RBUDP protocol ensures reliable delivery of data. We fixed UDP datagram size at 64 KB (or equal to the data size if the data to be transmitted is less than 64 KB).

64 KB is the largest datagram size allowed by the Linux operating system. This is done to reduce the number of system interrupts (through system calls).

### 6.2.2 Hardware-Assisted UDP Acceleration

This section presents performance to the hardware-assisted UDP acceleration scheme described in Section 5.2 for RBUDP based file transfer application. Figure 6.12, shows maximum network throughput observed against different file transfer sizes for three different cases.

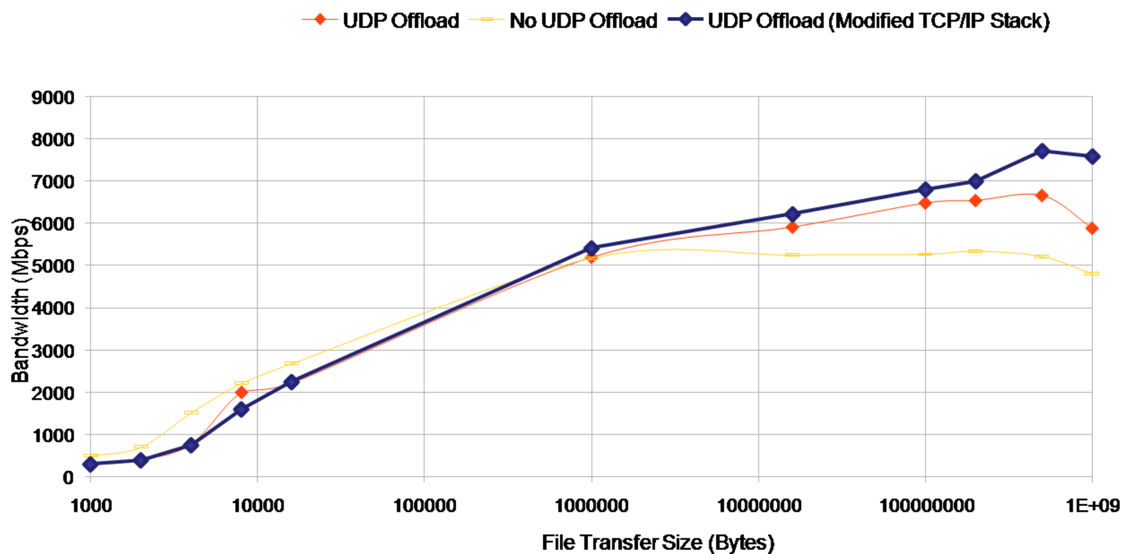


Figure 6.12: Evaluation of UDP Offload Core Component

"No UDP Offload" case refers to file transfer from sending host to the receiving host without using hardware-assisted UDP acceleration technique. It does not use high performance socket for network communication. Results shown in Figure 6.12 indicates the raw performance of RBUDP based file transfer application.

"UDP Offload" case refers to file transfer using high performance sockets, a technique discussed in Section 5.2.2. It uses abstract UDP/IP layer that essentially converts UDP/IP

communication to TCP/IP based communication. However like previous case it also uses existing Linux TCP/IP stack. From the Figure 6.12, we see that for large file transfers sizes, throughput obtained using high performance socket is considerably higher compared "No UDP Offload" case. Maximum throughput we observed for this case was close to 6800 Mbps. Unlike previous case where packet fragmentation/reassembly and checksum calculation is done by operating system (consuming important CPU cycles), application benefits from hardware support for these features using high performance sockets. Therefore this technique effectively utilizes protocol offloading features supported by Myri-10G NICs (described in Section 2.4) to obtain higher throughput.

Finally "UDP Offload (Modified TCP/IP Stack)" refers to file transfer using high performance sockets and simplified Linux TCP/IP stack discussed in Section 5.2.4. Like in "UDP Offload" case, we use hardware to offload packet fragmentation/ reassembly and checksum calculations in this case. Also we benefit by the simplified TCP/IP stack which now "behaves" like UDP/IP stack thereby getting rid of packet retransmission and congestion control features of TCP/IP among others that offers substantial overhead in packet processing. Though we observed throughput higher than 7.7 Gbps during some of our test runs, graph shown in Figure 6.12 is the most consistent reading we obtained.

### 6.2.3 High-Speed Reliable UDP Core Component

This section evaluates the application performance using high-speed reliable UDP core component to accelerate UDP data transfers. We present detailed analysis of effect of scheduling of various user threads on different system cores on the application performance.

Table 6.1 shows throughput achieved using high-speed reliable UDP core component for transfering 1 gigabyte file using single core. It can be seen from the table that maximum throughput of 5.4 Gbps can be achieved using single core by scheduling the main thread either on core 1, 2 or 3. However with core 0, lower throughput is observed. This is because core 0 besides running the receiver process (thread) also handles system-wide interrupt requests

Table 6.1: File Transfer using Single System Core

| Core 0 | Core 1 | Core 2 | Core 3 | Throughput (Mbps) | Sending Rate (Mbps) |
|---|---|---|---|---|---|
| X | | | | 3532.02 | 9467.76 |
| | X | | | 5326.21 | 9467.76 |
| | | X | | 5318.07 | 9467.76 |
| | | | X | 5313.34 | 9467.76 |

and therefore spends a percentage of its CPU cycles in servicing these interrupt requests. However same does not hold true for cores 1, 2 and 3 which run receiver process for 100% of their CPU time.

Table 6.2: File Transfer using Two System Cores

| Core 0 | Core 1 | Core 2 | Core 3 | Throughput (Mbps) | Sending Rate (Mbps) |
|---|---|---|---|---|---|
| X | X | | | 7398.85 | 9467.76 |
| X | | X | | 7891.98 | 9467.76 |
| | X | X | | 8927.79 | 9467.76 |
| | | X | X | 8599.98 | 9467.76 |

Table 6.2 shows throughput achieved using high-speed reliable UDP core component for 1 GB file transfer using two cores. In this case, main thread spawns an auxiliary thread on a different core. For the same reasons explained before, core combination involving core 0 will have lower throughput. However we see a significant throughput improvement in all the cases compared to a single core case shown in Table 6.1.

Table 6.3: File Transfer using Three System Cores

| Core 0 | Core 1 | Core 2 | Core 3 | Throughput (Mbps) | Sending Rate (Mbps) |
|---|---|---|---|---|---|
| X | X | X | | 9075.77 | 9297.96 |
| | | X | X | 9580.31 | 9585.91 |

Table 6.3 shows throughput achieved using high-speed reliable UDP core component for 1 GB file transfer using three cores. In this case, main thread spawns two auxiliary threads on

different cores. Throughput increment shown in the Table 6.3 is on the expected lines. We can observe that using high-speed reliable UDP core component, it is possible to do data transfer at line rate (which is 10Gbps here).

### 6.2.4 Hardware Vs Software-Based UDP Acceleration

In the previous sections, we analyzed the throughput achieved by RBUDP based file transfer application using hardware-assisted UDP acceleration scheme and using high-speed reliable UDP core component which is a software-based UDP acceleration scheme. As we can see from Figure 6.12, that the maximum throughput we observed for the hardware-assisted UDP acceleration is about 7.7 Gbps. However, using high-speed reliable UDP core component provided by GePSeA, we can actually achieve the throughput close to the line rate which is 10 Gbps here as can be seen in Table 6.3. Therefore, in this case software acceleration and hardware acceleration schemes are comparable to each other with former performing better than the later when "sufficient" number of system cores are available for computation to the core component. These results points to the efficacy of GePSeA framework. As we can see, GePSeA provide certain core components that perform comparable to their corresponding hardware implementations if not better as can be seen in this case.

# Chapter 7

# Related Work

In this chapter we discuss some of the previous and ongoing work on libraries and frameworks for multi/many-core systems. While there have been continued research efforts to develop specialized hardware accelerators [24, 19, 14, 22], very little exists with respect to software accelerators. However, interest in the research community to develop multi-core-aware applications [23] has increased in the recent past.

Protocol onload engine or the ProOnE [12] is closely related to our work and bear certain similarities with it. Like GePSeA, ProOnE efficiently utilizes a system resources in multi-core systems and ' certain communication- related tasks. In particular ProOnE, onloads *Rendezvous* protocol in MPI. However our work is distinct. Our framework with the both network and non-network task offloading as we have seen in this thesis. Also our work differ greatly in implementation, i.e. the way we perform task offloading.

Our work is also distinct from other frameworks and libraries [2, 4, 8] available for the development of parallel applications. Frameworks [2, 4] and libraries such as MPICH2 and OpenMP are collections of library functions that are intended for the quick development of parallel programs. In contrast, our framework is not intended for the development of parallel applications; rather, it is for the development of software accelerators that assist

parallel applications.

In [18, 11], the authors study several techniques that can be used for overlapping I/O, computation and communication in parallel scientific applications written using MPI and MPI-IO. Using GePSeA, applications can offload any of the aforementioned tasks, including disk I/O and communication.

In [23], the authors present a communication engine to exploit the cores in multi-core systems using various multi-threading techniques. The authors plan to integrate their engine with MPICH2 in the future. However, GePSeA differs in that it provides an infrastructure to quickly build application-specific software accelerators for any application (whether or not using MPICH2) as well as to efficiently schedule onloaded tasks to maximize the use of a compute node's system resources.

In summary, our work differs from the existing literature with respect to its capabilities and underlying architecture, but at the same time, forms a complementary contribution to other existing literature that can be simultaneously utilized.

# Chapter 8

# Conclusions and Future Work

In this chapter we present concluding remarks about the work presented in this thesis. In Section 8.2 we will discuss about the future work.

## 8.1 Concluding Remarks

In this thesis, we presented general purpose software acceleration framework called GePSeA that provide infrastructure to quickly build application-specific software accelerators. We presented a detailed design of GePSeA in Chapter 3 including the design of some of the core components. We proposed a efficient "core aware" reliable UDP core component for high-speed UDP data transfer. We also presented a novel hardware-assisted UDP acceleration scheme to exploit offloading features of network adapters to perform UDP protocol processing in hardware in order to evaluate reliable UDP core component.

We also demonstrated in this thesis that software accelerators are very effective in accelerating the applications on multi/many-core systems. We presented two case studies in order to evaluate GePSeA framework. First, we developed a software accelerator for a popular computational biology genome sequence search application called mpiBLAST and shown a

205% (or 2.05x) speed-up over its existing parallel implementation.

We then presented another case study of RBUDP based file transfer application over GePSeA framework to analyze the efficacy of high-speed reliable UDP core component provided by GePSeA. Our results show significant increase in UDP data transfer throughput using this core component. We also demonstrated that application performance using software-based scheme is better than the hardware-based UDP acceleration scheme presented in this thesis.

Finally, we also note that software accelerators are not meant to replace specialized hardware accelerators such as GPGPU or Cell Broadband Engine as they also offer certain benefits to the applications. Consequently, GePSeA use such hardware implementations when available and can also extend them by "onloading" some of the complex functionalities that cannot be easily offloaded to the hardware. We conclude by stating that software accelerators can be very cost-effective alternative to boost application performance on multi/many-core systems.

## 8.2 Discussion and Future Work

In this section, we highlight and discuss some issues related to the work presented in this thesis. We would address these issues in our future work.

- While we have implemented the basic infrastructure of GePSeA along with many useful core components, there are few areas where current design of the framework can be improved and extended for better accelerator performance. For example, as we have noted in Chapter 3 that we created two different service queues to service "intra-node" service requests and "inter-node" service request separately. However having two different service queues create issues such as starvation and deadlocks. While we have addressed few design issues like these in this thesis, more work needs to be done to extensively analyze such design considerations and better the existing design wherever possible.

- Many of our core components for parallel applications use *centralized server* design where a compute node (or an accelerator) on the cluster is elected as *leader*. *Leader* is the central authority that is key to the accomplishment of task performed by the core component. However in such designs, failure of *leader* is always a serious concern. Therefore more fault tolerant designs must be explored. Miguel Castro and Barbara Liskov [5] developed a efficient Byzantine fault tolerance protocol. Future work would involve studying the fault tolerant schemes proposed in [5] and incorporating them into the GePSeA framework.

- We conducted our experiments on ICE cluster where each compute node has only four cores with accelerator running on each node. While the CPU utilization for application process was close to 100%, it was only 2% to 5% for accelerator. Therefore for our mpiBLAST case study, one accelerator is sufficient to serve all application processes running on the node. However, on systems with very large number of cores like Intel's 80-core Terascale Chip [6], mapping correct number of accelerators and application processes onto the cores is challenging task that needs to be explored. Due to large number of available cores, large number of application processes can run on single node which can easily overrun a single accelerator. Optimum number of accelerators would depend on the CPU utilization of each accelerator and availability of system cores among other factors. Future work would involve exploring such accelerators to core mappings.

- mpiBLAST-1.5-pio is newer version of mpiBLAST that uses parallel I/O techniques [13] to improve the output result consolidation. However, we used mpiBLAST-1.4 for our case study since it has more structured code and therefore easy to identify portion of code amenable to offloading as compared to mpiBLAST-1.5-pio which is complex and unstructured. However both mpiBLAST-1.5-pio and mpiBLAST-1.4 using accelerator aims to eliminate the bottleneck caused due to single node result consolidation albeit using different techniques. Therefore it would be useful to compare the results from

the two in order to appreciate the impact of accelerator on performance of mpiBLAST application and the development efforts required to achieve that performance.

- We have evaluated our framework only for mpiBLAST and RBUDP based file transfer applications. Future work would involve conducting extensive performance and usability studies across many other diverse parallel applications and systems in order to better evaluate GePSeA.

# Bibliography

[1] S. Altschul, W. Gish, W. Miller, E. Myers, and D. Lipman. Basic Local Alignment Search Tool. *Journal of Molecular Biology*, 215(3), 1990.

[2] O. Aumage, G. Mercier, and R. Namyst. MPICH/Madeleine: A True Multi-Protocol MPI for High-Performance Networks. In *15th International Parallel and Distributed Processing Symposium*, 2001.

[3] P. Balaji, W. Feng, J. Archuleta, and H. Lin. ParaMEDIC: Parallel Metadata Environment for Distributed I/O and Computing . In *IEEE/ACM International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, 2007.

[4] F. Bertrand and R. Bramley. DCA: A Distributed CCA Framework Based on MPI. In *High-Level Parallel Programming Models and Supportive Environments*, 2004.

[5] M. Castro and B. Liskov. Practical Byzantine Fault Tolerance. In *Operating Systems Design and Implementation (OSDI), New Orleans, Louisiana*, 1999.

[6] Intel Corporation. Tera-Scale Computing. `http://www.intel.com/go/terascale`.

[7] A. Darling, L. Carey, and W. Feng. The Design, Implementation, and Evaluation of mpiBLAST. In *International Conference on Linux Clusters: The HPC Revolution 2003*, 2003.

[8] J. Dean and S. Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. In *Symposium on Operating System Design and Implementation*, 2004.

[9] H. Eric, J. Leigh, O. Yu, and T. DeFanti. Reliable Blast UDP: Predictable High Performance Bulk Data Transfer. In *IEEE Cluster Computing, Chicago, Illinois*, 2002.

[10] M. Gardner, W. Feng, J. Archuleta, H. Lin, and X. Ma. Parallel Genomic Sequence-Searching on an Ad-Hoc Grid: Experiences, Lessons Learned, and Implications. In *IEEE/ACM SC2006: The International Conference on High-Performance Computing, Networking, and Storage*, 2006.

[11] M. Jiayin, S. Bo, W. Yongwei, and Y. Guangwen. Overlapping Communication and Computation in MPI by Multithreading. In *International Conference on Parallel and Distributed Processing Techniques and Applications*, 2006.

[12] P. Lai, P. Balaji, R. Thakur, and D. K. Panda. ProOnE: A General Purpose Protocol Onload Engine for Multi- and Many-Core Architectures. Technical report, Argonne National Laboratory, 2009.

[13] H. Lin, X. Ma, P. Chandramohan, A. Geist, and N. Samatova. Efficient Data Access for Parallel BLAST. In *19th IEEE International Parallel and Distributed Processing Symposium (IPDPS'05)*, 2005.

[14] R. Luthy and C. Hoover. Hardware and Software Systems for Accelerating Common Bioinformatics Sequence Analysis Algorithms . In *Biosilico, 2(1)*, 2004.

[15] Myri-10G 10-Gigabit Ethernet Solutions. `http://www.myri.com/Myri1-10G/10gbe_solutions.html`.

[16] S. Narravula, A. Mamidala, A. Vishnu, K. Vaidyanathan, and D. K. Panda. High Performance Distributed Lock Management Services using Network-based Remote Atomic Operations. In *Int'l Symposium on Cluster Computing and the Grid (CCGrid)*, 2007.

[17] Net: TOE. `http://www.linuxfoundation.org/en/Net:TOE`.

[18] C. Patrick, S. Son, and M. Kandemir. Enhancing the Performance of MPI-IO Applications by Overlapping I/O, Computation and Communication. In *Symposium on Principles and Practice of Parallel Programming*, 2008.

[19] IBM Research. The Cell Project at IBM Research. `http://www.research.ibm.com/cell/`.

[20] T. Scogland, P. Balaji, W. Feng, and G. Narayanaswamy. Asymmetric Interactions in Symmetric Multi-core Systems: Analysis, Enhancements and Evaluation. In

*ACM/IEEE SC08: The International Conference on High-Performance Computing, Networking, Storage and Analysis, Austin*, 2008.

[21] L. Seiler, D. Carmean, E. Sprangle, T. Forsyth, M. Abrash, P. Dubey, S. Junkins, A. Lake, J. Sugerman, R. Cavin, R. Espasa, E. Grochowski, T. Juan, and P. Hanrahan. Larrabee: A Many-Core x86 Architecture for Visual Computing. In *SIGGRAPH '08: ACM SIGGRAPH 2008 papers*, pages 1–15, New York, NY, USA, 2008. ACM.

[22] R. Singh, W. Dettloff, V. Chi, D. Hoffman, S. Tell, C. White, S. Altschul, and B. Erickson. BioSCAN: A Dynamically Reconfigurable Systolic Array for Biosequence Analysis. In *CERCS96, National Science Foundation, Arlington*, 1996.

[23] F. Trahay, E. Brunet, A. Denis, and R. Namyst. A Multithreaded Communication Engine for Multicore Architectures. In *IEEE International Symposium on Parallel and Distributed Processing*, 2008.

[24] General-Purpose Computation using Graphics Hardware. `http://www.gpgpu.org`.

[25] K. Vaidyanathan, S. Narravula, P. Lai, and D. K. Panda. Optimized Distributed Data Sharing Substrate in Multi-Core Commodity Clusters: A Comprehensive Study with Applications. In *Int'l Symposium on Cluster Computing and the Grid (CCGrid)*, 2008.

[26] V. Vishwanath, P. Balaji, W. Feng, J. Leigh, and D.K. Panda. A Case for UDP Offload Engines in LambdaGrids. In *Protocols for FAST Long-Distance Networks*, 2006.