ΕΘΝΙΚΟ ΜΕΤΣΟΒΙΟ ΠΟΛΥΤΕΧΝΕΙΟ
ΤΜΗΜΑ ΗΛΕΚΤΡΟΛΟΓΩΝ ΜΗΧΑΝΙΚΩΝ ΚΑΙ ΜΗΧΑΝΙΚΩΝ ΥΠΟΛΟΓΙΣΤΩΝ

ΤΟΜΕΑΣ ΤΕΧΝΟΛΟΓΙΑΣ ΠΛΗΡΟΦΟΡΙΚΗΣ ΚΑΙ ΥΠΟΛΟΓΙΣΤΩΝ
ΕΡΓΑΣΤΗΡΙΟ ΥΠΟΛΟΓΙΣΤΙΚΩΝ ΣΥΣΤΗΜΑΤΩΝ

Σχεδιασμός και Υλοποίηση Συστήματος Απομακρυσμένης Πρόσβασης Συνεπεξεργαστών

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

του

Κωνσταντίνος Γ. Φερτάκης

Επιβλέπων: Γεώργιος Γκούμας
Επίκουρος Καθηγητής

Αθήνα, Ιούλιος 2018

Εθνικο Μετσοβιο Πολυτεχνειο
ΤΜΗΜΑ ΗΛΕΚΤΡΟΛΟΓΩΝ ΜΗΧΑΝΙΚΩΝ ΚΑΙ ΜΗΧΑΝΙΚΩΝ ΥΠΟΛΟΓΙΣΤΩΝ
ΤΟΜΕΑΣ ΤΕΧΝΟΛΟΓΙΑΣ ΠΛΗΡΟΦΟΡΙΚΗΣ ΚΑΙ ΥΠΟΛΟΓΙΣΤΩΝ
ΕΡΓΑΣΤΗΡΙΟ ΥΠΟΛΟΓΙΣΤΙΚΩΝ ΣΥΣΤΗΜΑΤΩΝ

Σχεδιασμός και Υλοποίηση Συστήματος Απομακρυσμένης Πρόσβασης Συνεπεξεργαστών

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

του

Κωνσταντίνος Γ. Φερτάκης

Επιβλέπων: Γεώργιος Γκούμας
Επίκουρος Καθηγητής

Εγκρίθηκε από την τριμελή εξεταστική επιτροπή την $13_{η}$ Ιουλίου 2018.

.....	.....	.....
Ν. Κοζύρης	Γ. Γκούμας	Δ. Τσουμάκος
Καθηγητής Ε.Μ.Π.	Επ. Καθηγητής Ε.Μ.Π	Αν. Καθηγητής Ιόνιο Πανεπιστήμιο

Αθήνα, Ιούλιος 2018.

.....

Κωνσταντίνος Γ. Φερτάκης
Διπλωματούχος Ηλεκτρολόγος Μηχανικός και Μηχανικός Υπολογιστών Ε.Μ.Π.

Copyright© Κωνσταντίνος Φερτάκης, 2018. Εθνικό Μετσόβιο Πολυτεχνείο.
Με επιφύλαξη κάθε δικαιώματος. All rights reserved.

Απαγορεύεται η αντιγραφή, αποθήκευση και διανομή της παρούσας εργασίας, εξ'ολοκλήρου ή τμήματος αυτής, για εμπορικό σκοπό. Επιτρέπεται η ανατύπωση, αποθήκευση και διανομή για σκοπό μη κερδοσκοπικό, εκπαιδευτικής ή ερευνητικής φύσης, υπο την προϋπόθεση να αναφέρεται η πηγή προέλευσης και να διατηρείται το παρόν μήνυμα. Ερωτήματα που αφορούν τη χρήση της εργασίας για κερδοσκοπικό σκοπό πρέπει να απευθύνονται προς το συγγραφέα. Οι απόψεις και τα συμπεράσματα που περιέχονται σε αυτό το έγγραφο εκφράζουν το συγγραφέα και δεν πρέπει να ερμηνευτεί ότι αντιπροσωπεύουν τις επίσημες θέσεις του Εθνικού Μετσόβιου Πολυτεχνείου.

National Technical University of Athens
Electrical and Computer Engineering Department
Computing Systems Laboratory

Master's Thesis

Remote Accelerator Execution

Author:	Fertakis, Konstantinos
Address:	Alkaiou 4
	11528 Athens
	Greece
Matriculation Number:	03112099
Supervisor:	Georgios Goumas, Assistant Proffesor, NTUA
Date:	13. July 2018

Περίληψη

Η χρήση επιταχυντών σε υπολογιστικές εγκαταστάσεις που χρησιμοποιούν ετερογένεια για την επίτευξη υψηλότερων επιδόσεων έχουν εδραιωθεί τα τελευταία χρόνια. Οι επιταχυντές βρίσκονται στις καρδιές των σύγχρονων κέντρων δεδομένων και υπολογιστών, υποστηρίζοντας τη λειτουργία της πλειοψηφίας των δέκα ταχύτερων υπερυπολογιστών στον κόσμο. Είναι ουσιαστικής σημασίας για τις κοινότητες υπολογιστών υψηλής απόδοσης και μηχανικής μάθησης, εφαρμόζοντας προσαρμοσμένη αρχιτεκτονική προκειμένου να παρέχουν αποτελεσματική κλιμακούμενη ισχύ επεξεργασίας που στοχεύει σε ένα ευρύ φάσμα επιστημονικών τομέων.

Σε αυτήν την εργασία, αναλαμβάνουμε την πρόκληση να σχεδιάσουμε και να υλοποιήσουμε ένα σύστημα το οποίο θα επιτρέπει την εξ αποστάσεως πρόσβαση στους πόρους ενός επιταχυντή. Παρουσιάζουμε το RACEX, ένα σύστημα που επιτρέπει την αποτελεσματική απομακρυσμένη εκτέλεση εφαρμογών σε επιταχυντή. Στη υλοποίηση της ιδέας μας, έχουμε στοχεύσει το συνεπεξεργαστή Intel Xeon Phi. Η προτεινόμενη λύση επιτρέπει την πλήρη ή μερική εκφόρτωση υπολογισμών και εφαρμογών σε έναν επιταχυντή Intel Xeon PHI προκειμένου αυτές να εκτελεστούν και να αξιοποιήσουν τη δύναμη των μαζικά παράλληλων επεξεργαστών του. Το RACEX εισέρχεται στη στοίβα λογισμικού του επιταχυντή στο επίπεδο στρώματος μεταφοράς που υλοποιείται απο το πρωτόκολλο SCIF της Intel, το οποίο προορίζεται για τη μεταφορά δεδομένων μέσω του PCIe στη συσκευή επιτάχυνσης. Το σύστημα μας υπεισέρχεται στις κλήσεις προς το πρωτόκολλου SCIF και προωθεί αυτές σε κάποιο απομακρυσμο διακομηστή προκειμένου να επιτρέψει την εξ αποστάσεως εκτέλεση. Το σύστημα μας χρησιμοποιεί BSD Sockets για δικτύωση και επικοινωνία μεταξύ των διαδικτυακά κατανεμημένων κόμβων. Τα αρχικά αποτελέσματα αξιολόγησης είναι ελπιδοφόρα, καθώς οι σχετικές μετρήσεις καταδεικνύουν 10% επιβάρυνση του RACEX σε σύγκριση με τη φυσική εκτέλεση όσον αφορά την καθυστέρηση για την ανταλλαγή μεγάλων μηνυμάτων μεταξύ του host και του επιταχυντή.

Λέξεις Κλειδιά: Επιταχυντές, Intel Xeon Phi, Απομακρυσμένη Εκτέλεση, Middleware Framework, Cluster Computing, Cloud Computing, Σύστημα Διαμοιρασμού Επιταχυντών, Hardware Abstraction, Transport Layer Abstractions

Abstract

The use of accelerators in computing facilities that employ heterogeneity in order to achieve higher performance has become prominent in the past years. Accelerators lie on the hearts of modern data center and computing facilities, powering the majority of the top ten super-computers in the world. They are essential for the High Performance Computing and Machine Learning communities, implementing custom architecture in order to provide efficient scalable processing power targeting a wide range of scientific domains.

In this work, we address the challenge of making accelerator resources remotely accessible. We present RACEX, a middleware framework that enables efficient Remote ACcelerator EXecution. For our proof-of-concept, we have targeted the Intel Xeon Phi coprocessor. Our proposed solution for the challenge allows applications to be, either completely or in part, offloaded remotely on an Intel Xeon PHI accelerator in order to be executed and harness the power of its massively parallel processors. RACEX intercepts Intel's SCIF transport layer API, intended to transfer data over the PCIe to the accelerator device, and wraps it to make it remotely available making use of BSD Sockets. Initial evaluation results are promising with RACEX framework showing 10% overhead compared to the native execution in terms of latency for big messages exchange between the host processor and the accelerator.

Keywords: Accelerator, Intel Xeon Phi, Remote Execution, Middleware Framework, Cluster Computing, Cloud Computing, Accelerator Sharing Framework, Hardware Abstraction, Transport Layer Abstractions

Ευχαριστίες

Η διπλωματική εργασία αυτή πραγματοποιηθήκε στο Εργαστήριο Υπολογιστικών Συστημάτων της Σχολής Ηλεκτρολόγων Μηχανικών και Μηχανικών Υπολογιστών του Εθνικού Μετσόβιου Πολυτεχνείου, υπο την επίβλεψη του Επίκουρου Καθηγητή Γεώργιου Γκούμα.

Καταρχήν θα ήθελα να ευχαριστήσω τον υποψήφιο διδάκτορα και μέλος του εργαστηρίου Στέφανο Γεράνγκελο για την συνεισφορά και εποπτεία του κατα τη εκπόνηση της διπλωματικής μου εργασίας, καθώς και για την ενθάρρυνσή του, την υπομονή του και το χρόνο που αφιέρωσε. Η συμβολή του ήταν καθοριστική τόσο στη σύλληψη του θέματος της εργασίας αλλά και για την ολοκλήρωση της.

Έπειτα θα ήθελα να ευχαριστήσω τους καθηγητές μου κ. Γέωργιο Γκούμα και κ. Νεκτάριο Κοζύρη για τη συμβολή τους στη διαμόρφωση μου ως μηχανικού απο τις διδασκαλίες τους. Θα ήθελα επίσης να εκφράσω τις ευχαριστίες μου σε όλα τα μέλη του εργαστηρίου για τις επικοδομητικές συζητήσεις οι οποίες συνεισφέραν στη διαμόρφωση αυτής της εργασίας.

Στην συνέχεια, επιβάλλεται να ευχαριστήσω τους φίλους και συμφοιτητές μου για την παρουσία τους και την πολύτιμη βοήθειά τους, τόσο σε προσωπικό όσο και σε επιστημονικό επίπεδο, καθ'όλη τη διάρκεια των προπτυχιακών μου σπουδών.

Τέλος, οι θερμότερες ευχαριστίες απευθύνονται στους γονείς μου, την αδερφή μου και την οικογένεια μου, για την αγάπη τους, την αμέριστη υποστήριξή τους και την εμπιστοσύνη τους σε κάθε μου επιλογή, από τα πρώτα χρόνια της ζωής μου μέχρι σήμερα, αδιάκοπα και ακούραστα.

Contents

List of Figures

Chapter 1

Introduction

1.1 Motivation

Heterogeneity has long been established in the realm of High Performance Computing as a design pattern applied in large scale computing systems and data centers. With the aggregation of data worldwide increasing in size exponentially each year alongside the complexity of applications serving scientific and commercial users, the race for ever more processing power is well underway. In this context, the employment of accelerators in modern computing facilities is a paradigm of the vigorous efforts of both academia and industry towards achieving higher computational performance for their respective workloads. Accelerators are prominent due to the massive parallelism capabilities they can provide, compared to a traditional general purpose CPU, and the custom, application specific architecture they leverage in order to provide the intended performance increase.

Furthermore, Cloud Computing has become commonplace for all manners of scientific and business endeavors, providing end-users with elastic and scalable access to computational resources [1, 2, 3]. Traditionally, cloud computing providers serve a diverse group of clients, by providing them access to distributed computational and storage resources, who in return utilize the provided infrastructure for a wide range of applications and purposes. However, with the proliferation of Machine Learning and Artificial Intelligence applications in both academia and industry, there has been an increasingly prevalent demand towards cloud computing providers for specialized hardware that can meet the heavy computational requirements of modern-day workloads [1, 4, 5]. This has led to a paradigm shift where today providers have incorporated accelerating resources in their computing facilities and data centers and offer elastic access to their massively parallel computational power, therefore giving birth to accelerating cloud computing [6, 7, 8].

The High Performance Computing community has already adopted computing services provided by cloud providers as a viable solution for executing compute-intensive HPC applications responsive on-demand [9]. Initially, cloud services offered by providers were not suited for traditional HPC applications, mainly due to the lack of high performance interconnects. However, recently major cloud computing providers have included high performing networking capabilities in their offered infrastructure, making them a compelling alternative to traditional HPC clusters which were expensive to maintain and operate. Moreover, in the IoT realm, thin, low-power devices capture an immense load of data that need to be processed in order to provide the intended

data analysis and intelligence. However, despite the technological advancements in the underlying hardware, IoT devices are by nature bound by processing power and energy consumption constraints. Thus, Cloud Computing for IoT has emerged by providing a compelling solution where the connected devices on the edge network offload the processing of computationally intensive tasks to the computing infrastructure and the specialized accelerating hardware that the Cloud can provide [10, 11].

In the light of the above, the importance of accelerators for the Cloud Computing community is paramount and although many current computing facilities may assimilate heterogeneity in their designs and employ the benefits that hardware accelerators have to offer, their usage is not ideal. Modern Cloud providers offering cloud accelerating services, grant access to accelerators on a per device basis. The integration of accelerators in the cloud infrastructure as part of collective computational resources can be troublesome, as their architectures are designed to be used monolithically [12]. Hence, providers offer accelerators as a fixed resource in an Infrastructure-as-a-Service model. In this context, providers are forced to acquire additional hardware in order to meet increasing demands, without having their initial hardware operating at maximum capacity and thus leading to over-provisioning. Low overall utilization of the underlying hardware and over-provisioning, in turn, means higher operational and maintenance cost, higher energy consumption for the computing facilities and larger space requirements in order to host the additional servers and hardware [13, 14].

In order to address the concerns that arise regarding the usage of accelerators resources in modern data centers, we argue that an accelerator sharing framework for computer clusters in cloud environments can be employed. By making accelerator resources available not only to the host device that they are attached to, but also to the rest of the cluster nodes in a data center, higher levels of utilization can be achieved per hardware accelerator. Hence, we can abate the total amount of hardware components needed by a data center for the same performance capabilities and subsequently curtail the total space required for the hardware to occupy. This not only increases usage of the underlying hardware but also mitigates the operational costs of the entire data processing facility. There has been a plethora of interesting research activities that have been carried out recently outlining the benefits of making accelerators available through sharing frameworks and the performance that can be gained [15, 16, 17, 18].

Taking into consideration recent developments in the field of sharing accelerator resources, we move forward and explore the idea of adding an additional level of hardware abstraction on the stack of computer clusters, one that will enable the accelerators present in the cluster to be managed collectively and their resources to be shared among all nodes of the cluster. In comparison to other remote accelerator sharing frameworks that have been explored [18, 19, 20], we content that by abstracting away the Transport Layer communication between the host machine and the accelerator, we can decouple the accelerating hardware from the sole usage from the host machine and make it remotely accessible. Current practices in cloud computing dictate that for a cloud computing user to gain access to accelerating resources, a virtual machine would have to be spawned on a host machine that the accelerator is directly attached to and then give to that virtual machine direct exclusive access to the accelerator. By abstracting away the Transport Layer, present on the accelerator's software stack and responsible for the bilateral communication between host and accelerator, we can grant shared access to accelerating resources to multiple virtual machines running on a single accelerated server node or even to remote VMs, scattered across the computing cluster. We can essentially virtualize the accelerators present in a data

center, traditionally used exclusively by the servers which they were attached to, and share their resources collectively by the cluster as part of a pool of accelerating hardware that a central scheduler can spin and assign tasks.

1.2 Contribution

In this work, we explore the feasibility, performance and benefits of the aforementioned argument by implementing a proof-of-concept that targets the Intel Xeon Phi accelerator. We present RACEX, a Remote ACcelerator EXecution framework that enables the remote concurrent access to accelerator resources for multiple clients transparently. RACEX is a middleware framework with a distributed Server-Client architectural design. In our design, RACEX is interposed into the Intel Xeon Phi's operating software stack directly above the transport layer, intercepting, wrapping and forwarding Intel's SCIF API (Symmetric Communication Interface) calls. SCIF constitutes the transport layer operating over the PCIe bus enabling communication between the host processor the accelerator coprocessor device. RACEX exposes the same Application Programming Interface as that of SCIF and it's binary compatible with precompiled applications. Also, we support traditional parallel computing frameworks and tools making our framework application transparent. In our current work, we support both *native* and *offload* execution modes and with minor adjustments we can extend our work to support also the *symmetric* execution model.

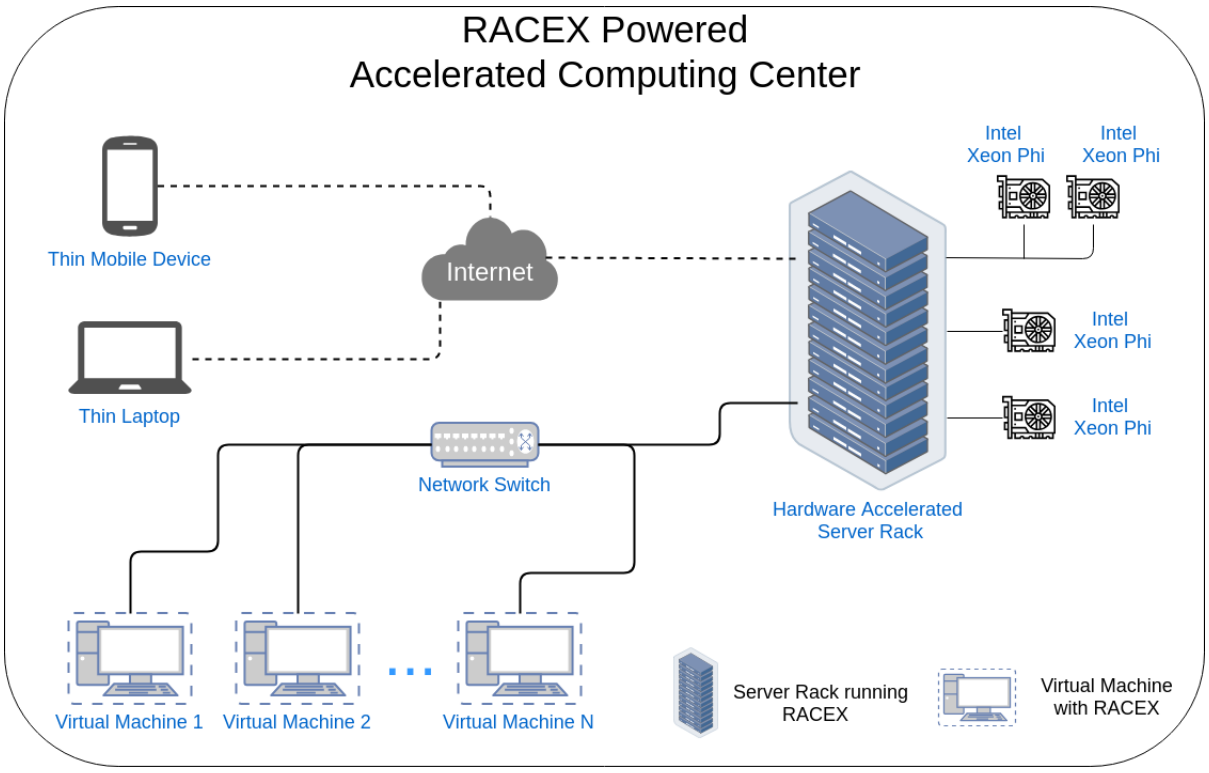


Figure 1.1: Accelerated Computing Center with RACEX Framework

An overview of a potential RACEX powered Heterogeneous Computing Center is depicted in Figure 3.1. In this illustration, thin remote devices but also in-house computing nodes and virtual machine instantiations offload computationally intensive tasks to available accelerators in a server rack equipped with multiple accelerators. With this work, we pave the ground for a potential future computing center, where smart scheduler running RACEX will receive incoming request for access to accelerating resources and will be able to efficiently direct access to available accelerating resources inside the computing center, across all computing nodes equipped with an accelerator.

Chapter 2

Background

2.1 Historical Review

2.1.1 The Path to Multicore Systems

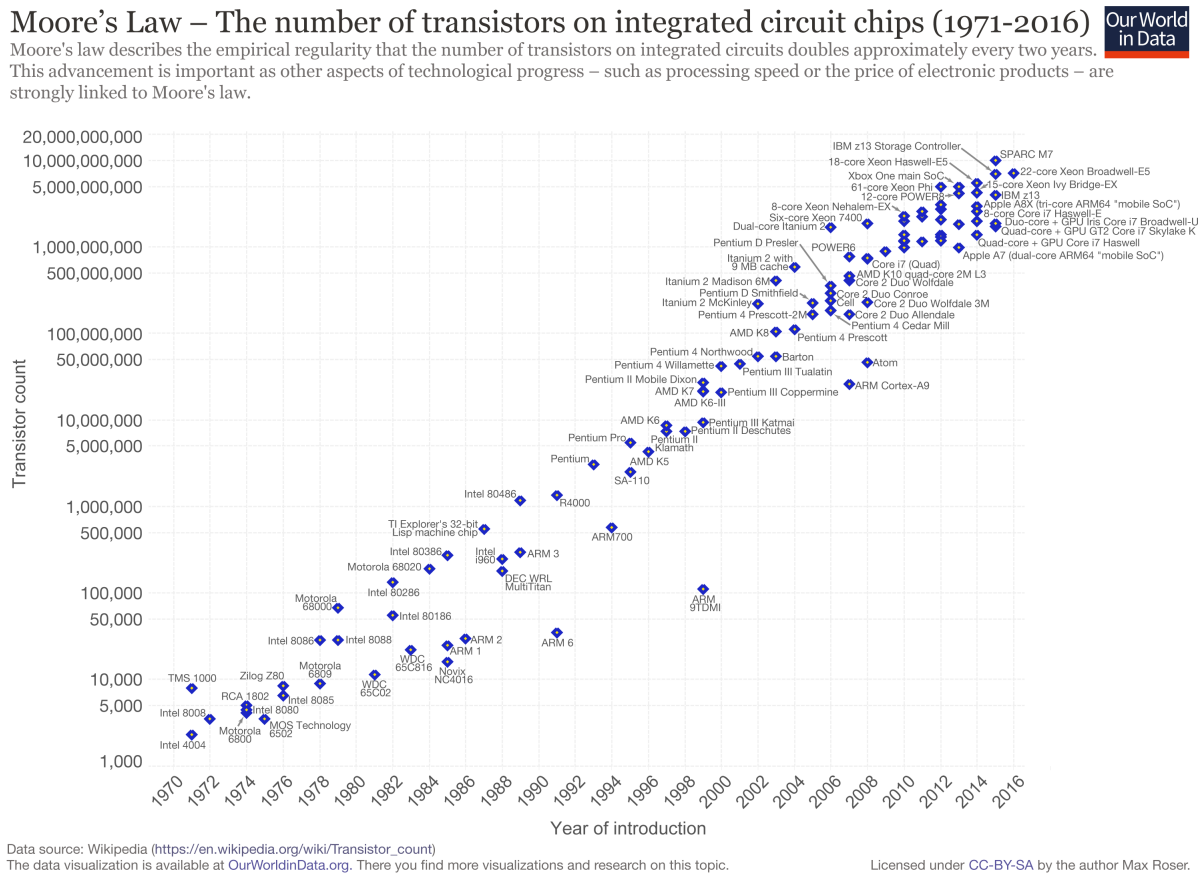
During the past century the world has bear witness to an unprecedented technological evolution, the so called Digital Revolution[21] which marked the shift from mechanical and analogue electronic technology to digital electronics. This period is considered as the birthplace of Computers but also the dawn of a new era, the beginning of the Information Age.

Originally, the first electronic general-purpose computers where enormous machines, occupying entire rooms, that had to be rewired and reconfigured in order to execute different functions while at the same time had significant operational costs, consuming immense power loads. Thus their usage was limited to mainly Government bodies and other organizations. The invention of the first transistor in 1947 in Bell Labs is considered by many one of the greatest inventions of the 20th century. By being the key component in all modern electronics, the invention of the transistor pathed the way towards the first Home Computers, allowing for the first time nontechnical users to operate a computer and harness it's processing power in an affordable cost.

From those microprocessors powering the first commercially available computers, the field of processing units witnessed advancements in an exponential rate. The ability to construct exceedingly small transistors on an integrated circuit increased the complexity and the number of transistors on a single processing unit many fold. That prompted a race between the academia and the industry to create CPUs with ever more processing power, in smaller size and cost. It was Gordon E. Moore that in 1965, in the light of recent developments in the fields of transistors and integrated circuits, stated what would latter be known as the "Moore's Law"[22]

> The complexity for minimum component costs has increased at a rate of roughly a factor of two per year. Certainly over the short term this rate can be expected to continue, if not to increase. Over the longer term, the rate of increase is a bit more uncertain, although there is no reason to believe it will not remain nearly constant for at least 10 years. That means by 1975, the number of components per integrated circuit for minimum cost will be 65,000. I believe that such a large circuit can be built on a single wafer.

Moore's observation came to be known as a prediction that the number of transistors in a dense integrated circuit doubles approximately every two years. The same empirical law came to be widely accepted as a goal for the industry, and it was cited by competitive semiconductor manufacturers as they strove to increase processing power.



Alongside Moore's empirical observation, another law regarding the performance scaling of processing units came to be during the race for higher performing CPUs, the "MOSFET Scaling" law or as it is more widely known "Dennard Scaling", named after the co-author of the original 1974 paper, which stated that as transistors get smaller their power density stays constant. The development of processing units during the race for ever more and more performance power, reaching higher and higher CPU clock frequencies seemed to abide by the scaling law but around 2005 Dennard scaling appeared to have broken down. Although transistor counts in integrated circuits were still growing, the resulting improvements in performance were more gradual than the speed-ups resulting from significant frequency increases. The primary reason cited for the breakdown is that at small sizes, current leakage poses greater challenges, and also causes the chip to heat up, which creates a threat of thermal runaway and therefore further increases energy costs. These created a "Power Wall" that has limited practical single core processor frequency to around 4 GHz since 2006.

As transistors could not be relied upon to increase serial performance in a single core architecture processor, an alternative had to be found in order to meet the ever growing demand for

higher processing power. The solution came with the shift to multicore architecture for processing units. The motivation for multi-core processors came from greatly diminished gains in processors performance from increasing the operating frequency which mainly can be traced back to three primary factors:

1. The memory wall; the increasing gap between processor and memory speeds. This, in effect, pushes for cache sizes to be larger in order to mask the latency of memory. This helps only to the extent that memory bandwidth is not the bottleneck in performance.

2. The Instruction-Level Parallelism (ILP) wall; the increasing difficulty of finding enough parallelism in a single instruction stream to keep a high-performance single-core processor busy.

3. The power wall; the trend of consuming exponentially increasing power (and thus also generating exponentially increasing heat) with each factorial increase of operating frequency. This increase can be mitigated by "shrinking" the processor by using smaller traces for the same logic. The power wall poses manufacturing, system design and deployment problems that have not been justified in the face of the diminished gains in performance due to the memory wall and ILP wall.

The failure of Dennard scaling led both academia and industry to focus their efforts in increasing the number of cores on chips as a viable and compelling alternative for performance increase. Multicore systems enabled performance improvements through parallel processing, solving a problem by dividing it in multiple tasks which can execute simultaneously on multiple processors. The increase in processing power through multicore systems is more challenging to benefit from, since programmers need to develop parallel software that exploits the multicore architecture.

2.1.2 Amdahl's Law and the Challenge of Parallel Programming

As we entered the multicore era, the computing landscape moved away from single-core systems design and focused on multicore processors with the notion that the parallization that multiple cores on a single chip could attain would suffice towards maintaining the performance scaling that the market demanded and was previously experienced during the golden age of Moore's Law. However, the availability of multiple cores on the same chip does not guarantee that the performance of the executed applications will scale effortlessly. In many cases, the programmer has to explicitly design the software to take advantage of the available cores. Writing software to take advantage of multiple cores in order to speed-up an application compared to its performance on a uni-processor system is inherently difficult.

In a parallel program, the computational load has to be broken into equal-sized pieces and then each piece assigned to a core. In case that the workloads of the assigned tasks are not equally sized, some cores would remain idle while waiting for the other ones with the larger pieces to finish. Moreover, different processors have to communicate with each other during execution in order to successfully process their workload. If the total time spent for communication between the cores is significant enough, it can mitigate the total speedup achieved by the parallelization of the program. In short, scheduling, load balancing, time for synchronization and overhead for communication between the cores are significant challenges and they become even greater as the number of processors increases.

When someone is considering the total speedup that a program can achieve through parallelization, one has to consider Amdahl's Law [23]. Amdahl's Law is a formula that gives the theoretical speed-up that can be achieved, in terms of latency, regarding the execution a program in a parallel manner compared to the serial execution. In the early years of high performance computing, Gene M. Amdahl[23] first denoted some inherent constraints in the process of parallel programming and efficiency attainment. Firstly, there is a fraction of computational load in every application, associated with data management, which cannot be executed in parallel with other computations and acts as a constant overhead to the runtime. Secondly, when the problem's dataset is distributed among processors, irregularity problems may occur, such as inhomogeneous interiors, irregular boundaries, inconsistency issues among variables and asymmetric convergence computations. To model the first restriction and set guidelines for coping with irregularity problems, Amdahl introduced his famous law, which captures their effect on the obtainable speedup.

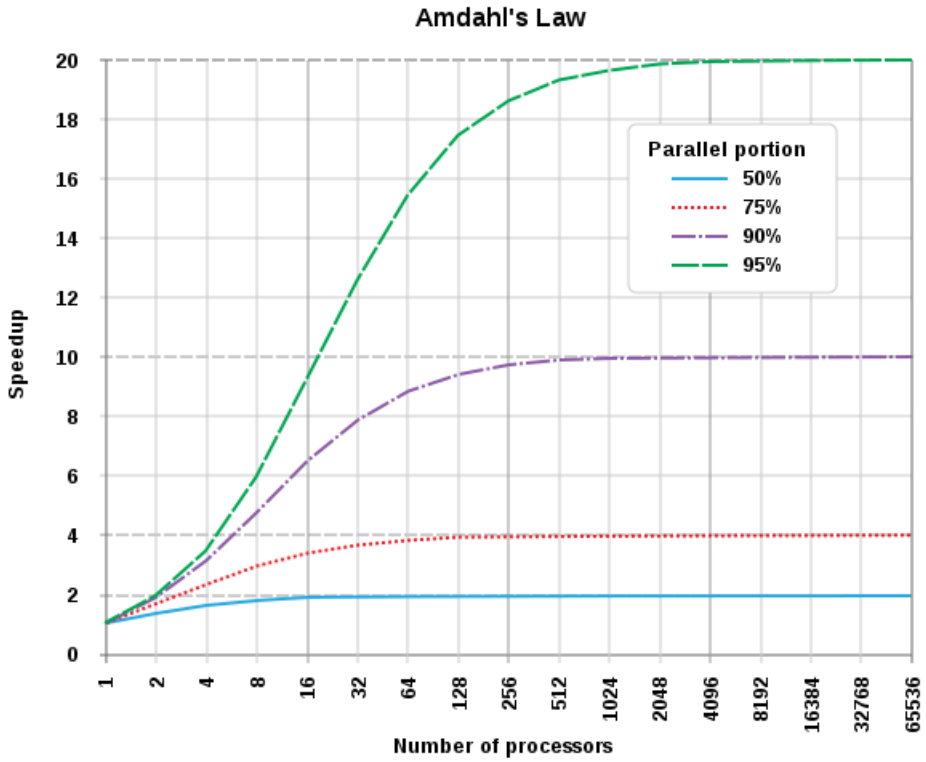


Figure 2.1: Total speedup of a parallel program as parallel fraction and number of processors increase

Before we further define Amdahl's Law, we first have to define speedup in terms of the performance gained during a program's execution.

$$speedup(n\ processors) = \frac{execution\_time(1\ processor)}{execution\_time(n\ processors)}$$

Amdahl's law refers to the increase of the performance that can be achieved by improving a part of the total program, let it be a proportion p. If this improvement makes that part of the

program s times faster, i.e. it has a speedup equal to s, then the total speedup of the program is given by the equation:

Thus we define total speedup of the aforementioned program as:

$$total\_speedup = \frac{1}{(1-p) + \frac{p}{s}}$$

Considering how the number of independent processors that contribute to the parallelization of a program affect the definition of the theoretical speedup achievable, we suffice the following : If p is the proportion of the total program that can be parallelized and $1-p$ is the part of the program that can not be parallelized (i.e. remains sequential), then the maximum total speedup that can be achieved by using N processors is given by the following equation:

$$total\_speedup = \frac{1}{(1-p) + \frac{p}{N}}$$

As N goes to infinity, the maximum speedup goes to $1/(1-p)$. Practically, the analogy between efficiency and cost increases dramatically even if $1-p$ is relatively small. At Figure 2.1 we can see how total speedup is affected while proportion p and number of processors N change. First of all, the overall speedup of a program that utilizes many cores in a parallel portion is bounded by the sequential part of the program. Therefore, using even more cores and improving parallel computer architectures is not panacea. Instead, an application should be redesigned so that a larger part of it is parallelized.

2.1.3 Multicore Architectures

According to Michael J. Flynn's *taxonomy* for computer architectures [24], there are four classifications for computer systems that are based upon the number of concurrent instruction and data streams available in the architecture. Below these four categories are described:

1. **Single Instruction Single Data (SISD):** A sequential computer which exploits no parallelism in either the instruction or data streams.
2. **Single Instruction Multiple Data (SIMD):** It represents the organization of a single computer containing a control unit, processor unit and a memory unit. Instructions are executed sequentially. It can be achieved by pipelining or multiple functional units
3. **Multiple Instruction Single Data (MISD):** Multiple instructions operate on one data stream. This is an uncommon architecture which is generally used for fault tolerance. Heterogeneous systems operate on the same data stream and must agree on the result.
4. **Multiple Instruction Multiple Data (MIMD):** Multiple autonomous processors simultaneously executing different instructions on different data. MIMD architectures include multi-core super-scalar processors, and distributed systems, using either one shared memory space or a distributed memory space.

Multi-core computers are based on MIMD architecture which can be further classified depending on their memory organization into shared-memory architectures, distributed-memory architectures and hybrid architectures, where aspects of both previous architectures are employed. The three respective architectural patterns are explained below.

Shared-Memory Architecture

In a shared-memory architecture, each processor owns a private cache memory hierarchy and collectively, all processors of the system share a single physical address space, a global memory. All processors and the global memory are interconnected through a single shared system bus. Processors communicate through shared variables in memory, with all processors capable of accessing any memory via loads and stores. When all processors are equidistant from the global memory, the memory architecture is also defined as symmetric multiprocessor (SMP) and can be viewed in Figure 2.2.

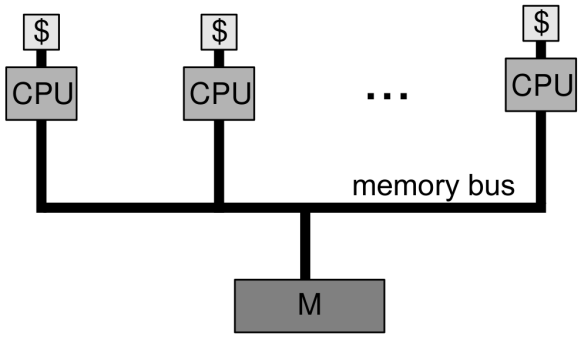


Figure 2.2: Classic organization of a SMP

This model enables fast and uniform data sharing between tasks due to the proximity of memory to CPUs. Global address space also offers ease of programming. The ability to access all shared data efficiently from any of the processors using ordinary loads and stores, together with the automatic movement and replication of shared data in the local caches, makes them attractive for parallel programming. These features are also very useful for the operating system, whose different processes share data structures and can easily run on different processors. However this approach suffers form lack of scalability. Adding more CPUs can geometrically increase traffic on the shared system bus as well as traffic associated with cache coherence. Based on memory access times, shared memory architectures can be classified into two categories

1. **Uniform Memory Access (UMA):** The latency to a word in memory does not depend on which processor asks for it. All processors share the physical memory uniformly and access time to a memory location is independent of which processor makes the request or which memory chip contains the transferred data.

2. **Non-Uniform Memory Access (NUMA):** Memory access latency for each processor depends on the memory location. Each processor has its own local memory and can also access memory owned by other processors. Memory access time in this model depends on the memory location relative to the processor, since processors can access their local memory faster than non-local memory. The NUMA architecture was designed to surpass the scalability limits of the UMA architectures. It alleviates the bottleneck of multiple processors competing for access to the shared memory bus.

Commercial symmetric multiprocessors have come to use the UMA organization. Although programming challenges are harder for a NUMA multiprocessor than for a UMA multiprocessor,

NUMA machines are capable of scaling to larger sizes and can have lower latency to nearby memory. As processors operating in parallel will normally share data, they also need to coordinate when operating on shared data. Otherwise, a processor could start working on a data before another processor is finished with it. Thus, when sharing is supported with a single address space there must be a mechanism for synchronization. An approach to achieve synchronization is the usage of locks for shared variables. Only one processor at a time can acquire the lock and any other processor interested in shared data must wait until the original processor unlocks the variable.

Distributed-Memory Architecture

The alternative approach to sharing an address space is for each processor to have its own cache hierarchy and its own private physical address space. A distributed-memory architecture network is comprised by a group of nodes, each consisting of a processor, a local cache hierarchy and a local main memory. The organization of this memory architecture approach, which is also called message passing, is depicted in Figure 2.3. Message passing, served by the interconnection network, is the only way of communication between the isolated nodes. Provided the system has routines to send and receive messages, coordination is built in with message passing mechanism, since one processor knows when a message is sent, and the receiving processor knows when a message arrives. If the sender needs confirmation that the message has arrived, the receiving processor can then send an acknowledgment message back to the sender. Note that unlike the SMP, the interconnection network is between processor-memory nodes.

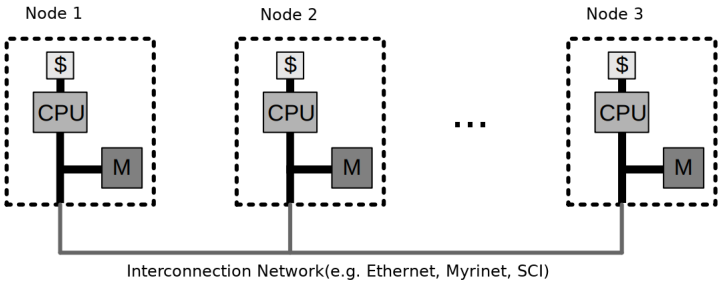


Figure 2.3: Classic organization of distributed memory architecture

There have been several attempts to build large-scale computers based on high-performance message-passing networks, and they do offer better absolute communication performance than clusters built using local area networks. Indeed, many supercomputers today use custom networks. The problem is that they are much more expensive than local area networks like Ethernet. Few applications today outside of high performance computing can justify the higher communication performance, given the much higher costs.

The distributed memory model achieves high scalability, since memory can increase proportionally to the the number of processor, as well as cost effectiveness, since commodity processors and networking infrastructure can be used to build such systems. Moreover, each processor can rapidly access its local memory without the overhead incurred by global cache coherence operations. On the other hand, programming is more challenging as developers are responsible for details associated with communication between processors.

Hybrid Architecture

Hybrid memory architectures combine the benefits of shared-memory and distributed-memory models onto the same computing system. In this memory architecture approach, nodes with shared memory architecture are connected via an interconnection network using the distributed memory architecture. The organization of this approach is depicted in in Figure 2.4. This design permits parallel processing within each node and scales up in the same way as a distributed memory system. Since this approach incorporates the benefits of both previous memory architecture models, it is the typical architecture of the modern clusters and supercomputers.

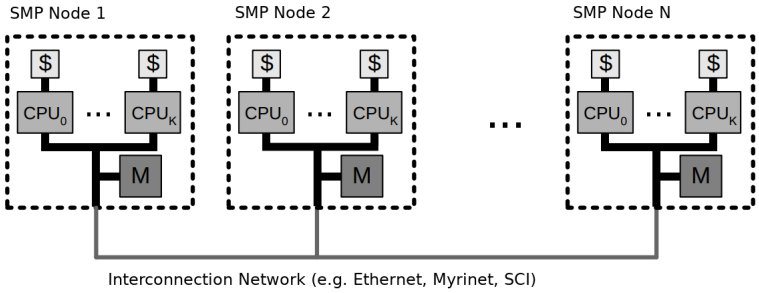


Figure 2.4: Example of a hybrid architecture

2.2 Accelerators

In computing, *hardware acceleration* is defined as the process, during which specialized computer hardware is employed in order to perform functions more efficiently than what is possible for a software running on a more general-purpose CPU. When the hardware that performs the computing acceleration exists as a separate unit from the general-purpose CPU, then it is referred to as a *hardware accelerator*.

The most notable hardware accelerators are detailed bellow.

2.2.1 Graphics Processing Unit (GPU)

A Graphics Processing Unit (GPU) is a single-chip processor that performs rapid mathematical calculations, primarily for the purpose of rendering images. In the early days of computing, the central processing unit (CPU) performed these calculations. As more graphics-intensive applications were developed; however, their demands put strain on the CPU and degraded performance. GPUs came about as a way to offload those tasks from CPUs, freeing up their processing power. In 1999, NVIDIA introduced the world's first GPU, the GeForce 256 [25]. It was introduced as :

> A single chip processor with integrated transform, lighting, triangle setup/clipping, and rendering engines that is capable of processing a minimum of 10 million polygons per second.

It could process 10 million polygons per second, allowing it to offload a significant amount of graphics processing from the CPU. ATI Technologies released the Radeon 9700 in 2002 using the term visual processing unit (VPU). Over the past few years there has been a significant increase in performance and capabilities of GPUs due to market demand for more sophisticated graphics. Moreover, over time, the processing power of GPUs made the chips a popular choice for other resource-intensive tasks unrelated to graphics.

Nowadays, GPUs are widely used in embedded systems, mobile phones, personal computers, and game consoles. Modern GPUs are very efficient at manipulating graphics as well as in image processing. Furthermore, their highly parallel structure makes them more effective than general-purpose CPUs for algorithms where processing of large blocks of data is done in parallel. As a result, a large discrepancy in floating-point capability between the CPU and the GPU was emerged. The main reason is that GPUs are specialized for compute-intensive, highly parallel computations and therefore designed such as more transistors are devoted to data processing rather than data caching and flow control, as is the case for the CPU. More specifically, GPU is especially well-suited to address problems that can be expressed as data-parallel computations with high arithmetic intensity. Because the same program is executed in each data element, there is a lower requirement for sophisticated flow control. Memory access latency can be hidden with calculations instead of big data caches. GPUs can therefore be considered as general-purpose, high-performance, many-core processors capable of very high computation and memory throughput.

General Purpose Graphics Processing Unit (GPGPU)

A general-purpose GPU (GPGPU) is a graphics processing unit (GPU) that performs non-specialized calculations that would typically be conducted by the CPU (central processing unit). Ordinarily, the GPU is dedicated to graphics rendering.

GPGPUs are used for tasks that were formerly the domain of high-power CPUs, such as physics calculations, encryption/decryption, scientific computations and the generation of cypto currencies such as Bitcoin. Because graphics cards are constructed for massive parallelism, they can dwarf the calculation rate of even the most powerful CPUs for many parallel processing tasks.

General-purpose computing on GPUs only became practical and popular after about 2001, with the advent of both programmable shaders and floating point support on graphics processors. Notably, problems involving matrices and/or vectors – especially two-, three-, or four-dimensional vectors – were easy to translate to a GPU, which acts with native speed and support on those types.

The architecture of GPUs, with many cores and high bandwidth in their private memory, designed for the efficient parallel execution of many tasks, is ideal for applications that make many simultaneous calculations (such as math models). This is why GPU acceleration - the parallel use of a GPU, alongside with the CPU, to efficiently execute costly applications - is becoming more and more more popular in recent years in various fields [26, 27, 28, 29]. In order to harness the power that GPUs can provide to general purpose computations, several programming interfaces have been designed and implemented. The most popular and widely used frameworks for GPGPU are listed below.

1. OpenCL: The Open Computing Language (OpenCL) is a framework for writing programs that execute across heterogeneous platforms consisting of central processing units (CPUs), graphics processing units (GPUs), digital signal processors (DSPs), field-programmable

gate arrays (FPGAs) and other processors or hardware accelerators. OpenCL is an open standard maintained by Khronos Group[30].

2. CUDA: The Compute Unified Device Architecture (CUDA) is a parallel computing platform developed by NVIDIA and introduced in 2006. It enables software programs to perform calculations using both the CPU and GPU. By sharing the processing load with the GPU (instead of only using the CPU), CUDA-enabled programs can achieve significant increases in performance [31].

2.2.2 Field-Programmable Gate Array (FPGA)

A field-programmable gate array (FPGA) is an integrated circuit (IC) that can be programmed in the field after manufacture. The FPGA configuration is generally specified using a hardware description language (HDL), similar to that used for an application-specific integrated circuit (ASIC). FPGAs are similar in principle to, but have vastly wider potential application than, programmable read-only memory (PROM) chips. FPGAs contain an array of programmable logic blocks, and a hierarchy of reconfigurable interconnects that allow the blocks to be "wired together", like many logic gates that can be inter-wired in different configurations. Logic blocks can be configured to perform complex combinational functions, or merely simple logic gates like AND and XOR. In most FPGAs, logic blocks also include memory elements, which may be simple flip-flops or more complete blocks of memory. FPGAs have gathered increased attention as they are proven to be able to provide significant performance boost to application in computing centers that employ heterogeneity in their designs.

2.2.3 Coprocessor

A coprocessor is a computer processor used to supplement the functions of the primary processor (the CPU). Operations performed by the coprocessor may be floating point arithmetic, graphics, signal processing, string processing, encryption or I/O Interfacing with peripheral devices. By offloading processor-intensive tasks from the main processor, coprocessors can accelerate system performance.

2.3 Intel Xeon Phi

As a proof-of-concept for our proposed framework, we employ the Intel Xeon Phi coprocessor, member of Intel's Xeon Phi processor family. The Xeon Phi processor family consists of a series of massively-parallel x86 manycore processors that are compatible with standard programming languages and major APIs such as OpenMP[32] and MPI. As of 2017 Intel Xeon Phi powers four out of the ten highest performing supercomputers in the world[33]. Intel's Xeon Phi processor family employs the Many Integrated Core (MIC) architecture, leveraging the x86 architecture and extending it's compatibility with existing programming tools, compilers and libraries. Applications intended for a MIC processor can easily be executed on a standard Intel Xeon x86 processor. The main difference between Xeon Phi and a GPGPU like Nvidia Tesla is that Xeon Phi, with an x86-compatible core, can, with less modification, run software that was originally targeted at a standard x86 CPU. In our work, we explore the benefits of making Xeon Phi's resources available

remotely and shared across a computing cluster. The underlying architecture of an Intel Xeon Phi accelerator is illustrated in Figure 2.5.

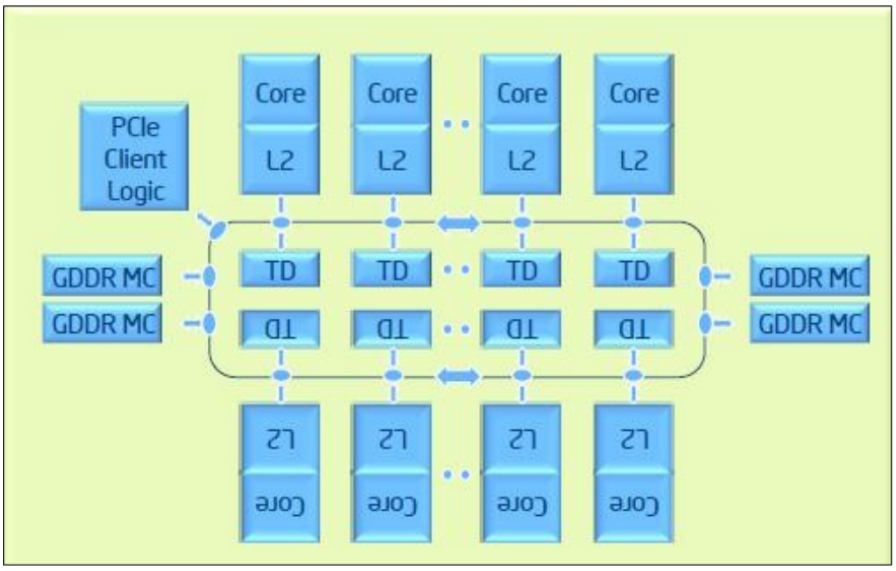


Figure 2.5: Intel Xeon Phi Architecture Ring and Cores

Source: Intel's MPSS User's Guide

2.3.1 Xeon Phi Execution Models

The Xeon Phi coprocessor supports different execution models based on the intended functionality and the execution scheme. The execution models can be summarized in three categories: *native*, *offload* and *symmetric*.

1. Native Mode: In this mode, an application can be executed entirely on to the device. To run natively, the application has to be cross compiled for the Xeon Phi operating environment and transferred directly to the coprocessor. Intel provides the necessary tools to support the native execution mode through Manycore Platform Software Stack, Intel's software collection that support the coprocessor's operations.

2. Offload Mode: Also known as heterogeneous programming mode, the execution of an application is split between the host and the coprocessor with the former offloading computationally intensive parts to be executed on the later. The offloading mode is supported and strengthened by traditional parallel computing frameworks such as OpenMP.

3. Symetric Mode: In this scenario the application is run concurrently in the host machine and in the Intel Xeon Phi. The coprocessor is identified by the host as independent processing node and the two parts communicating through a message passing interface like MPI.

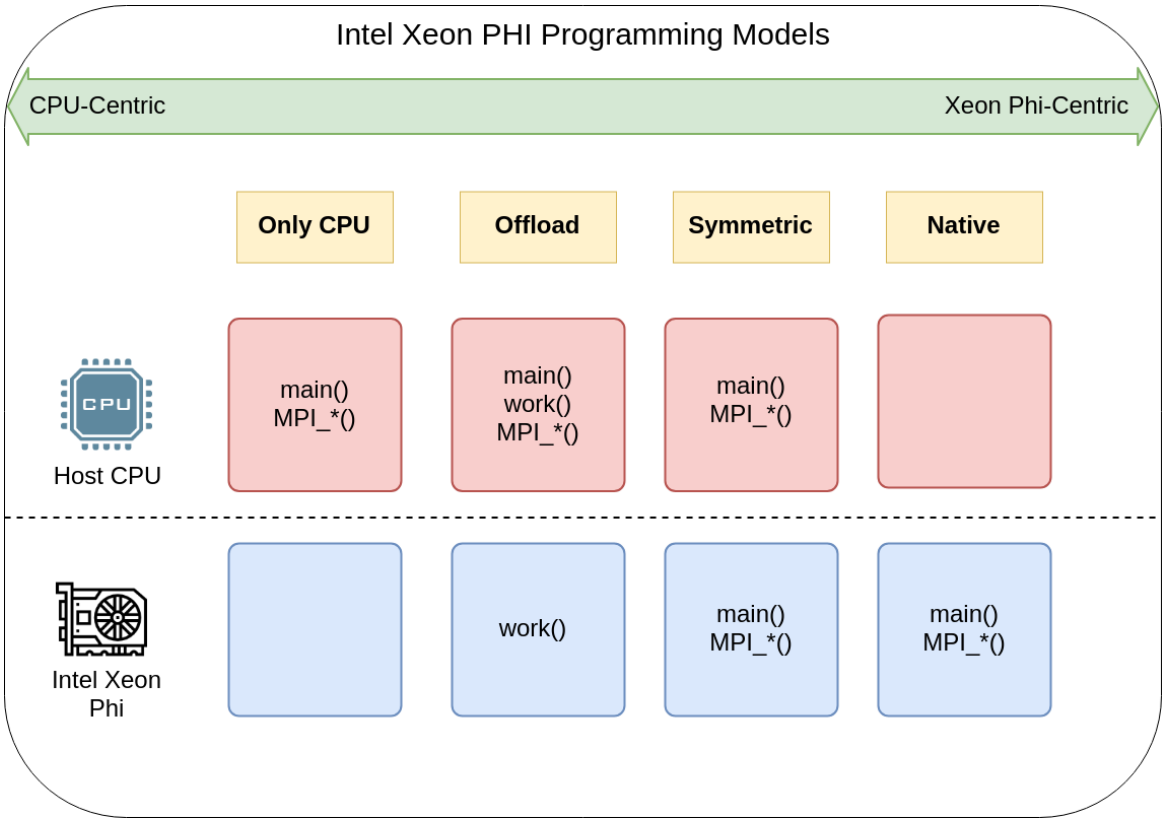


Figure 2.6: Intel Xeon Phi Programming and Execution Models

The programming models that correspond to the aforementioned execution models are illustrated in Figure 2.6. Apart from the Xeon Phi execution models, the "Only CPU" model illustrated in Figure 2.6 corresponds to a normal execution of an application which takes place solely on the CPU with no engagement of the accelerator.

2.3.2 Intel ManyCore Platform Software Stack

Intel provides a collection of inter-dependable software that are necessary to operate the Intel Xeon Phi coprocessor. The MPSS includes the software intended for both the host machine and the coprocessor and the necessary framework for them to efficiently communicate and coordinate the coprocessor's operations. Among the prominent features of Intel's Manycore Platform Software Stack, the Coprocessor Offload Infrastructure (COI) is essential for our work as it exposes the necessary API for offloading executables and data on the coprocessor, and employes the necessary mechanism to provide a virtual shared memory model that simplifies data sharing between processes on the host and each coprocessor. COI and other Intel MPSS components rely on the Symmetric Communication Interface (SCIF) API for PCIe communication services between the host processor and the coprocessors. SCIF delivers very high bandwidth data transfers and subsec write latency to memory shared across PCIe, while abstracting the details of communication over PCIe [34]. A simplified version of the MPSS, with the components relevant to our work, is depicted in Figure 2.7.

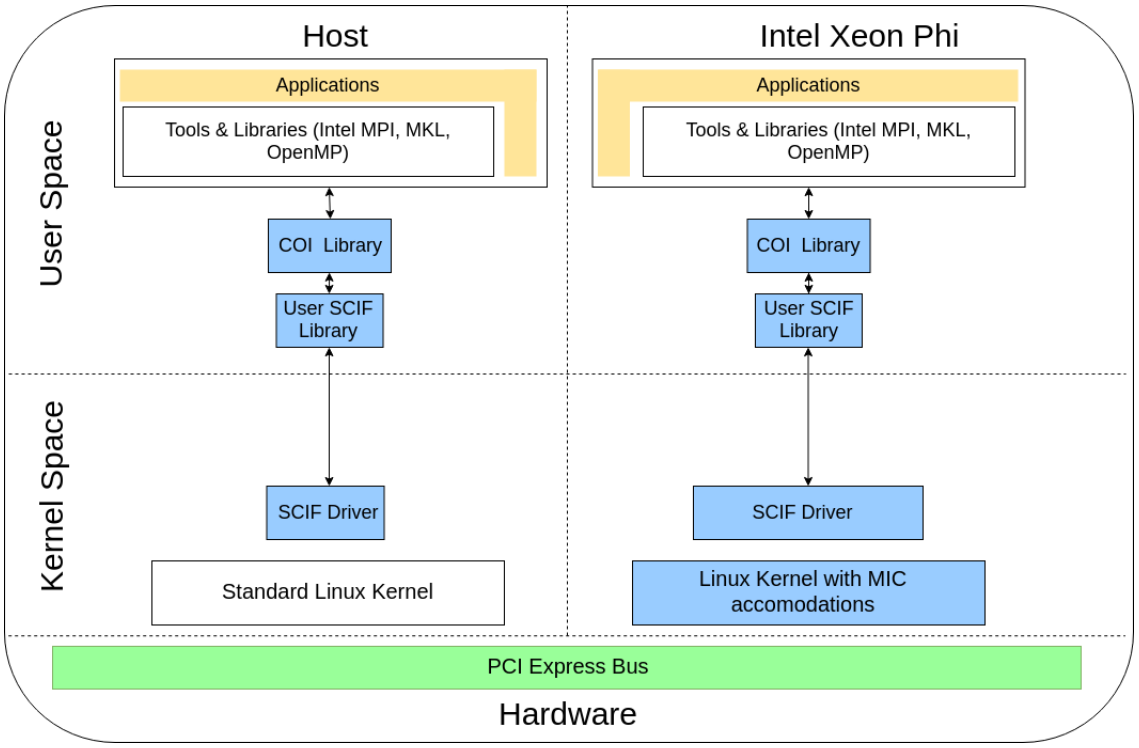


Figure 2.7: ManyCore Platform Software Stack Components Overview

2.3.3 Symmetric Communication Interface (SCIF)

Intel's Symmetric Communication Interface or SCIF provides a mechanism for inter-node communication within a single platform, where a node is an Intel Xeon Phi coprocessor or an Intel Xeon host processor complex. In particular, SCIF abstracts the details of communicating over the PCIe bus while providing an API that is symmetric between the host and MIC Architecture devices. The Intel MIC software architecture supports a computing model in which the workload maybe distributed across both the Intel Xeon host processor complex and Intel MIC Architecture coprocessors. An important property of SCIF is symmetry; SCIF drivers must present the same interface on both the host processor and the Intel MIC Architecture coprocessor in order that software written to SCIF can be executed wherever is most appropriate. Since the Intel MIC Architecture coprocessor may use a different operating system than that running on the host, the SCIF architecture is designed to be operating system independent. This ensures SCIF implementations on different operating systems can inter-communicate. SCIF supports communication between Xeon host processors and Intel MIC Architecture coprocessors within a single platform. Communication between such components that are in separate platforms can be performed using standard communication channels such as Infiniband and TCP/IP. A SCIF implementation on a host or Intel MIC Architecture coprocessor includes both a user mode library and kernel mode driver as shown in Figure 2.7. Most of the components in the Intel MPSS use SCIF for communication.

The SCIF driver provides a reliable connection-based messaging layer, as well as functionality which abstracts RMA operations. SCIF provides a communication mechanism between different

SCIF nodes. A SCIF node is a physical endpoint in the SCIF network. The host and the MIC Architecture devices are SCIF nodes. The process of establishing a connection between different SCIF nodes is similar to socket programming, with similar semantics being utilised by SCIF: `scif_open()`, `scif_bind()`, `scif_listen()`, `scif_connect()`, `scif_accept()`. Accordingly the functionality that the aforementioned SCIF functions implement is similar to that of the sockets functions. A typical connection flow between two different nodes in the SCIF network is depicted in Figure 2.8.

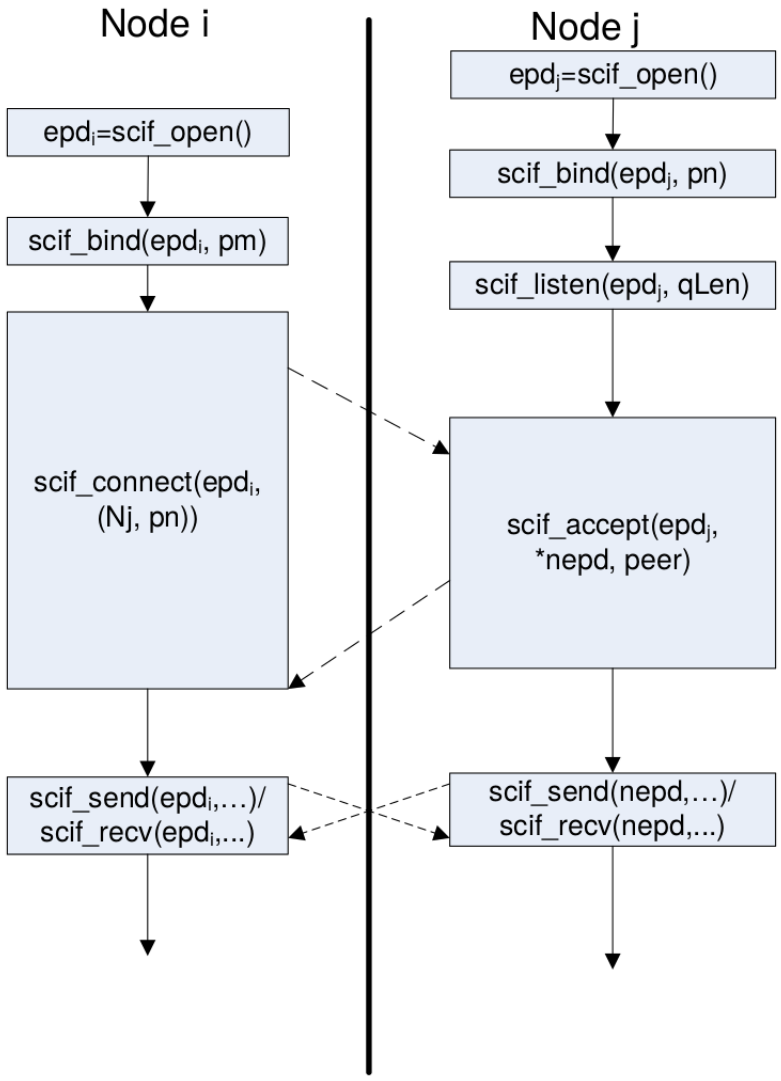


Figure 2.8: Connecting two different SCIF nodes

Source: Intel's SCIF User Guide

SCIF Messaging Layer

After a connection has been established, messages may be exchanged between the processes owning the connected endpoints. A message sent into one connected endpoint is received at the other connected endpoint. Such communication is bi-directional. The Messaging Layer of SCIF is comprised by the socket-like function `scif_send()` and `scif_recv`. Messages are always sent through a local endpoint for delivery at a remote connected endpoint. For each connected pair of endpoints, there is a dedicated pair of message queues – one queue for each direction of communication. In this way, the forward progress of any connection is not gated by progress on another connection, which might be the case were multiple connections sharing a queue pair. A message may be up to 2^{31} -1 bytes long. In spite of this, the messaging layer is intended for sending short command-type messages, not for bulk data transfers. The messaging layer queues are relatively short; a long message is transmitted as multiple shorter queue-length transfers, with an interrupt exchange for each such transfer. Therefore SCIF RMA functionality should be used for sending larger units of data, e.g. longer than 4KiB.

2.3.4 SCIF Remote Memory Access

The SCIF employs the use of a mechanism that enables remote memory access (RMA) from the memory of the host processor to the memory of the co-processor.[35] This mechanism permits high-throughput, low-latency transfer operations between the two parties. The mechanism depends upon Memory Registration in which a process exposes a range of memory pages in its virtual address space, to be accessed by another process, typically by a process residing on a remote node. The remote memory access operations are utilized in order to transfer a binary executable intended to run on the co-processor, shared libraries as well as bilateral transfer of large data buffers.

In order for a process to map memory that belongs to a remote process, residing either on the host or the accelerator, it first has to be registered with the SCIF driver. Each connected endpoint, has a subsequent registered address space. The Register Address Space (RAS) is an abstract address space managed by the SCIF Driver, ranges of which can represent local physical memory. The SCIF Driver registers a range of user-space virtual memory when `scif_register()` is called, in the form of a registered window and returns an offset by which the registered address space can be accessed by both parties involved in an RMA operation. The communicating parties can reference and access the registered window by passing the offset to the relevant SCIF RMA API calls. The mapping between registered address space and physical memory remains even if the specified virtual address range is unmapped or remapped to some different physical pages or object.

In Figure 2.9, the memory registration mechanism is illustrated. The Figure presents a registered window W which was created by `scif_register()`. The pages of W, a range in the registered address space of some local SCIF endpoint, represent some set, P, of physical pages in local memory. P is the set of physical pages which backed a specified virtual address range, VA, at the time that `scif_register()` was executed. Even if the virtual address range, VA, is subsequently mapped to different physical pages, W continues to represent P. Of course, if the virtual address range is unmapped or remapped to different physical pages, the process has no way of accessing the registered memory in order to read or write RMA data unless those physical pages back some other virtual address range.

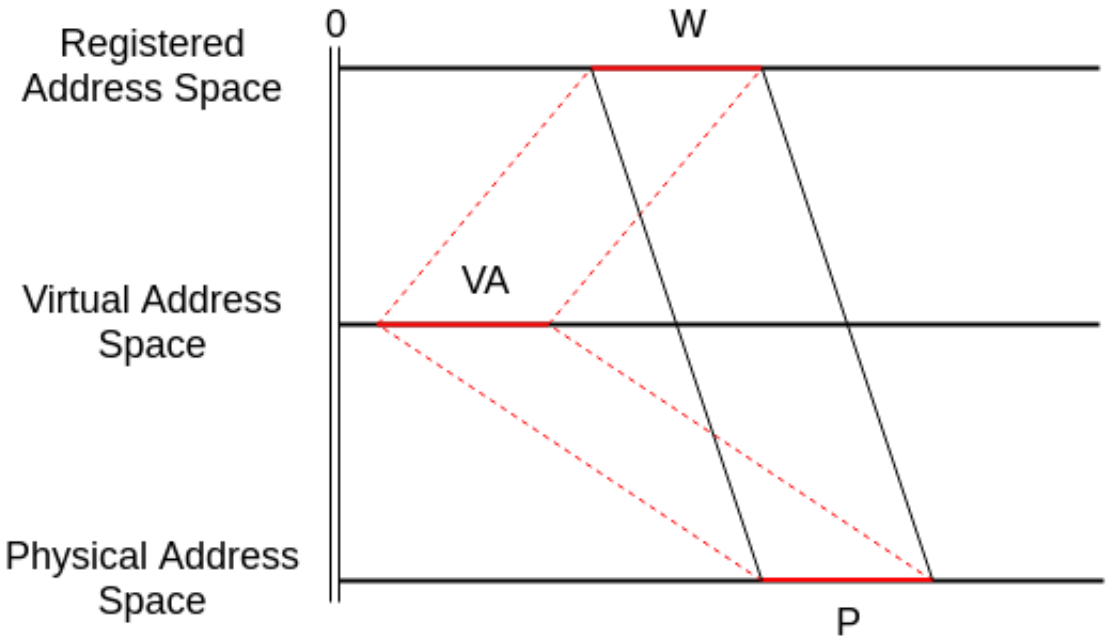


Figure 2.9: Memory Registration Mechanism

SCIF RMA operations are intended to support the one-sided communication model which has the advantage that a read/write operation can be performed by one side of a connection when it knows both the local and remote locations of data to be transferred. One-sided calls can often be useful for algorithms in which synchronization would be inconvenient (e.g. distributed matrix multiplication), or where it is desirable for tasks to be able to balance their load while other processors are operating on data.

The `scif_readfrom()` and `scif_writeto()` functions perform DMA or CPU based read and write operations, respectively, between physical memory of the local and remote nodes of the specified endpoint and its peer. The physical memory is that which is represented by specified ranges in the local and remote registered address spaces of a local endpoint and its peer remote endpoint. Specifying these registered address ranges establishes a correspondence between local and remote physical pages for the duration of the RMA operation. Specific RMA flags passed on to the relevant SCIF RMA operations control whether the transfer is DMA or CPU based.

In Figure 2.10 we illustrate such a mapping between two remote SCIF nodes, who will perform an RMA procedure. The process performing the operation specifies a range, LR, within the registered address of one of its connected endpoints, and a corresponding range, RR, of the same length within the peer endpoint's registered address space. Each specified range must be entirely within a previously registered window or contiguous windows of the corresponding registered address spaces. The solid green lines represent the correspondence between the specified ranges in the local and remote registered address spaces; the dashed green lines represent the projections into their respective physical address spaces. This defines an overall effective correspondence (black lines) between the physical address space of the local node and that of the remote node of the peer registered address space. Hence, a DMA operation will transfer data between LP and RP (again, LP and RP are typically not contiguous).

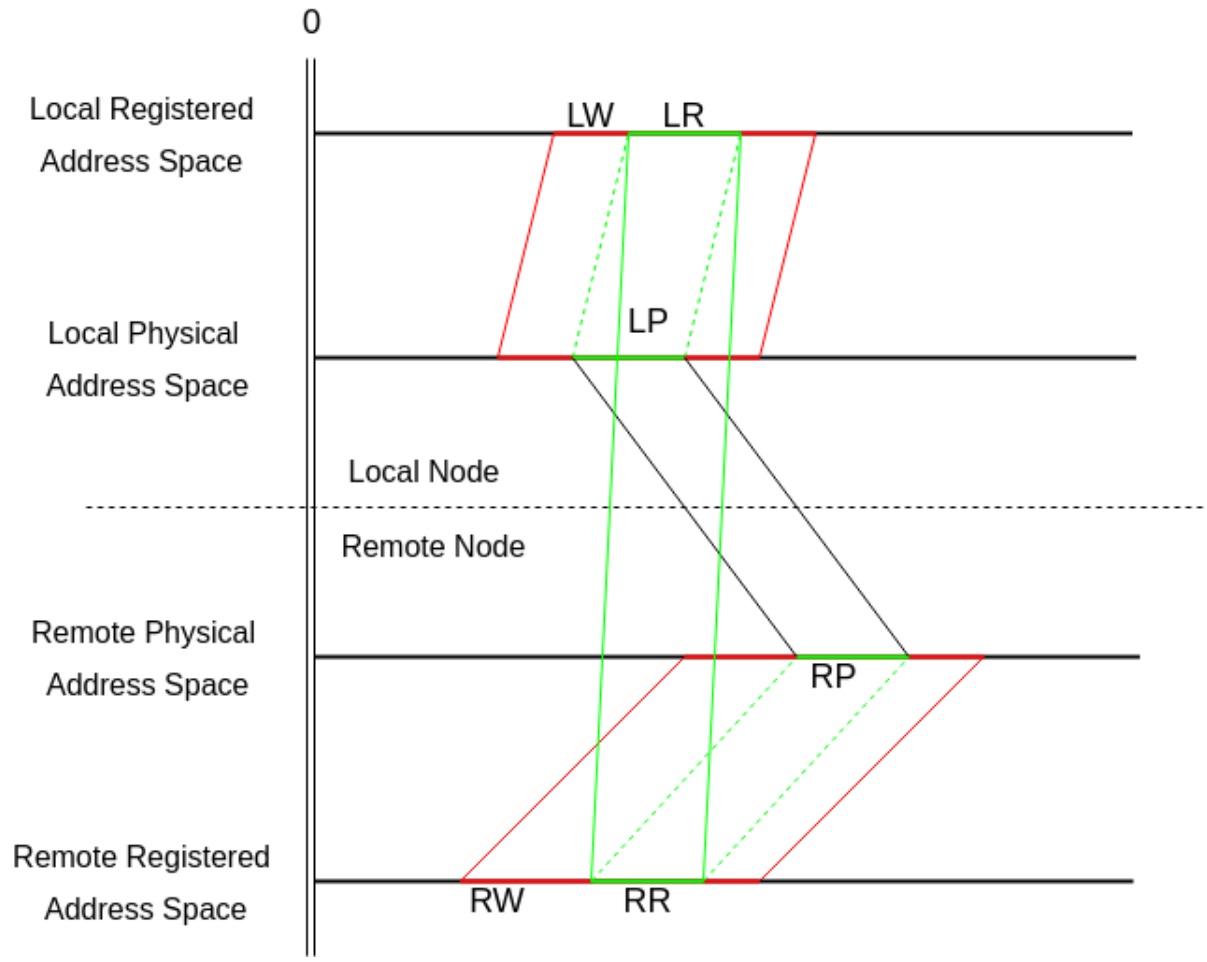


Figure 2.10: SCIF RMA Mapping Between Remote Nodes

2.4 Sockets - TCP/IP

A socket is defined as an endpoint for communication. A pair of processes communicating over a network employs a pair of sockets—one for each process. A socket is identified by an IP address concatenated with a port number. In general, sockets use a client–server architecture. The server waits for incoming client requests by listening to a specified port. Once a request is received, the server accepts a connection from the client socket to complete the connection. When a client process initiates a request for a connection, it is assigned a port by its host computer. This port has some arbitrary number greater than 1024.

In our work, we utilize the Berkeley Sockets or BSD Sockets. Berkeley sockets is an application programming interface (API) for Internet sockets and Unix domain sockets, used for inter-process communication (IPC). It originated with the 4.2 BSD Unix released in 1983. A socket is an abstract representation (handle) for the local endpoint of a network communication path. The Berkeley sockets API represents it as a file descriptor (file handle) in the Unix philosophy that provides a common interface for input and output to streams of data. The BSD Sockets implementation reside on top of the TCP/IP Stack. The TCP/IP Stack or Internet protocol suite provides

end-to-end data communication specifying how data should be packetized, addressed, transmitted, routed, and received. This functionality is organized into four abstraction layers, which classify all related protocols according to the scope of networking involved. From lowest to highest, the layers are the link layer, containing communication methods for data that remains within a single network segment (link); the internet layer, providing internet networking between independent networks; the transport layer, handling host-to-host communication; and the application layer, providing process-to-process data exchange for applications.

2.5 Serialization

Serialization is the process of translating data structures or object state into a format that can be stored (for example, in a file or memory buffer) or transmitted (for example, across a network connection link) and reconstructed later (possibly in a different computer environment). When the resulting series of bits is reread according to the serialization format, it can be used to create a semantically identical clone of the original object. For many complex objects, such as those that make extensive use of references, this process is not straightforward. Serialization of object-oriented objects does not include any of their associated methods with which they were previously linked.

In our work we employ the use of serialization techniques in order to transmit structured data between the communicating parties in our framework. More specifically, we transmit messages over the network stack between remote nodes. Those messages are comprised by scattered memory values. Thus, with the employment of serialization techniques, we are able to take a memory data structure that is comprised by scattered memory regions, into a stream of bytes that can be transmitted over the network and then reconstructed in the same manner as to represent the same data structure.

2.5.1 Protocol Buffers

For our work, we have employed the use of Google's Protocol Buffers [36] as a serializer for our framework. Protocol buffers are Google's language-neutral, platform-neutral, extensible mechanism for serializing structured data. The Protocol Buffer, involve an interface description language that describes the structure of the data that are will be involved in a serialization procedure, together with a program that generates source code from that description that is intended for generating or parsing a stream of bytes that represent the structured data. Google's Protocol Buffers have extensive compatibility with many programming languages and are published under an open-source license. The design goals for Protocol Buffers emphasized simplicity and performance. In particular, it was designed to be smaller and faster than XML.

Chapter 3

Design and Implementation

We design RACEX to be interposed inside the Manycore Platform Software Stack and intercept the Transport Layer communication between the host processor and the accelerator, implemented by the SCIF API. The RACEX framework intercepts, wraps and forwards SCIF API calls through a middleware library that connects to a server-side daemon that is responsible for serving client's requests and executing the original scif calls to the coprocessor. We implement a Client-Server distributed architecture model. We illustrated a high-level overview of RACEX's design in Figure 3.1. The underlying architecture of the framework together with the Data and Control Paths, are depicted in Figure 3.2. The main components that constitute the RACEX framework are the client-side "libracex" library which wraps the SCIF API, in order to make it remotely accessible, and the server-side RACEX daemon, which exposes a remote SCIF execution requests API. The host machine that the Xeon Phi is physically connected to acts as the server in our framework.

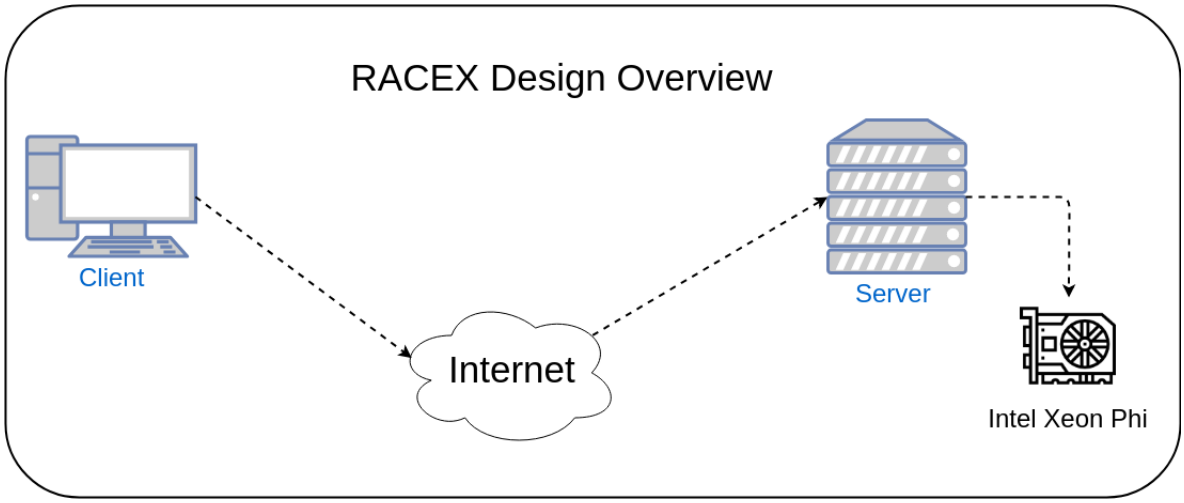


Figure 3.1: RACEX Design Overview

We illustrate in Figure 3.2 the I/O path for our framework when a SCIF API call request is triggered by an application. We represent the I/O path for a scenario that corresponds to an offload execution mode from the aforementioned Xeon Phi modes of execution. Solid lines represent control path and dashed lines represent the subsequent data paths. The client communicates with

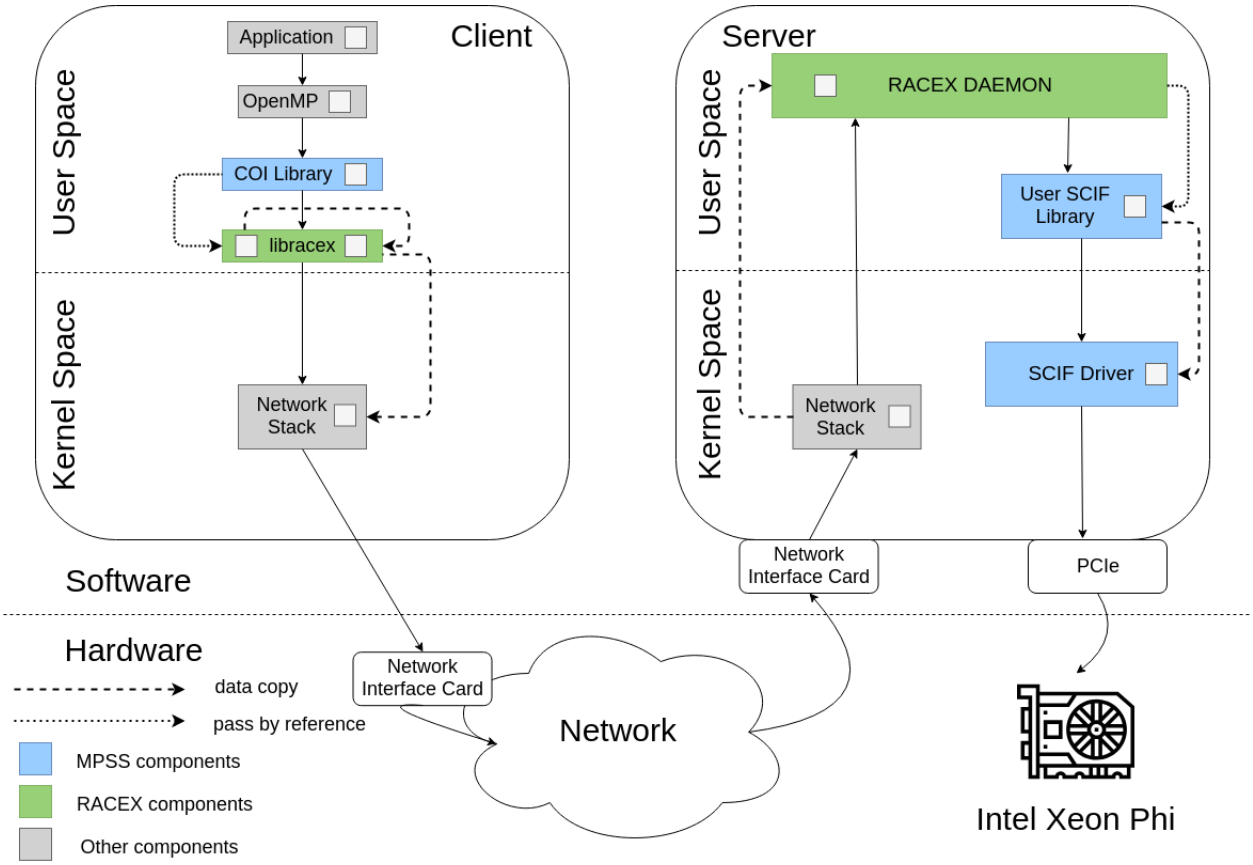


Figure 3.2: Architecture of the RACEX Framework (Data and Control Path)

the server side daemon using the BSD Sockets API and the TCP/IP Stack, represented in the Figure 3.2 as "Network Stack". We provide further details regarding the design of our distributed architecture in the following sections.

3.1 Client-Side Library Architecture

The client part of the middleware framework we are proposing consists of a library that is installed in the system that is requesting remote access to accelerator resources. The client library offers the same Application Programming Interface (API) as does the original Intel SCIF library and it's binary compatible with . Hence we are able to provide compatibility with the rest of the Manycore Platform Software Stack and parallel programming APIs such as OpenMP.

The client-side library intercepts the SCIF library calls either by preloading the library to the dynamic linker of the client's system before the intended execution, or by installing the library in the system and replacing any preexisting `scif` library. The RACEX framework is based on a thread level communication between the client and the server. Each respective thread of a process issuing a SCIF call, initiates a corresponding TCP connection with a dedicated thread on the server-side daemon that will serve incoming requests by the client and return the resulting data. The execution flow of the client-side library is summarized by the Algorithm 1.

Algorithm 1 Client-Side Library Execution Flow

```
1:  procedure SCIF_CALL()
2:      curr_thread ← identify_current_thread()
3:      if !has_established_connection(curr_thread) then
4:          establish_connection()
5:      if is_rma_operation() then
6:          mapping ← get_mapping()
7:      cmd ← pack_phi_cmd(args)
8:      send_phi_cmd(cmd)
9:      res ← recv_phi_cmd_results()
10:     if res− > success() then
11:         set_resulting_values(mapping)
12:     else
13:         set_error_values()
14:     return
```

For each SCIF API call made in the client's system, the execution flow of our framework performs the following tasks: (1) acquire the connection handle to the server side daemon for the calling thread, in case there is no open connection with the server, we initiate a new one (2) pack all the required arguments together with the function identifier in the RACEX's communication protocol designated data structure for forwarding API calls (3) send the SCIF call request to the server daemon (4) wait for the results message from the server (5) unpack the results and copy any required data to their specified memory addresses (6) in case of any error in the remote execution of the SCIF call, set the appropriate error value (7) return the appropriate value to the caller.

3.2 Server Daemon

The server-side daemon of our framework resides on the server that is physically connected to the Intel Xeon Phi coprocessor and serves potential RACEX clients, enabling concurrent access to the accelerator's resources. For each incoming connection, the server daemon will spawn a dedicated serving thread that will serve all the request for remote SCIF API call requests. In order to avoid concurrency issues and also to have clearly separated client-server execution contexts, each calling thread in the client-side has a dedicated serving thread on the server-side. In that way, the same TCP/IP connection is only used by the two communicating threads, the client issuing the request for a remote SCIF API call and the server thread that will serve the client's request. The execution flow of the server-side daemon is summarized in Algorithm 2.

For each RACEX remote call message that the daemon's serving thread receives, the execution flow of our framework performs the following tasks: (1) unpack the remote API call identifier and the relevant arguments (2) optionally perform a memory mapping operation in case of remote `scif_register()` call in order to reserve a memory region for subsequent RMA operations or copy data in the designated memory region to be accessed by following RMA operations. (3) execute the actual SCIF API call(4) pack the output data to be transferred back to the client(5)send the

Algorithm 2 RACEX Server Daemon Execution Flow

```
1: procedure RACEX_DAEMON()
2:     wait_for_incoming_connection()
3:     spawn_serving_thread()
4: serve_client:
5:     cmd ← receive_phi_cmd()
6:     unpack_phi_cmd(cmd)
7:     if is_rma_operation() then
8:         mapping ← retrieve_mapping()
9:         process_phi_cmd(cmd)
10:    execute_phi_cmd(args)
11:    results ← pack_phi_results()
12:    send_phi_cmd(results)
13:    res ← recv_phi_cmd_results()
14:    if client_finished() then
15:        return
16:    else
17:        goto serve_client
18:    return
```

response to caller and (6) block until any new message comes from the calling thread.

3.3 RACEX Communication Protocol

We have implemented a custom remote API call communication protocol that is utilized in RACEX by the two distributed communicating parties, the client and the server daemon. The protocol utilizes Google Protocol Buffers [36] as a fast and light serialization method for serializing structured data. As a result, as it is presented in the data path in Figure 3.2 for both client-side RACEX library and server-side daemon, we experience a data copy inherent in the Protocol Buffer's serialization procedure. In our future work, we plan to remove any overhead presented by the data copy procedure that the serializer features. Instead, we plan to employ the usage of scatter/gather message structs for transmitting the required structured data between client and server and thus eliminating the need for serialization and data copies. In the following Evaluation Section we will see that the overhead latency featured by the aforementioned data copies is insignificant compared to the present network latency. Moreover, the communication protocol was developed in order to accommodate the need from our framework for structured data exchange between the client, who is requesting access to the Xeon Phi's resources and the server-side daemon serving the client's request for remote execution of SCIF API calls. However, the protocol is designed to be generic, able to support structured communication needed for enabling a remote API execution framework and thus making it application independent, contributing to a generic framework for Remote Accelerator Execution.

3.4 Remote Memomy Access Operations

Special care is taken for the Remote Memory Access operations in which the following SCIF functions take part: `scif_register()`, `scif_unregister()`, `scif_writeto()`, `scif_readfrom()`, `scif_vwriteto()`, `scif_vreadfrom()`. In order for the SCIF communicating parties to be able to perform an RMA operation, they first have to register the appropriate memory region that will be accessed during the RMA operation. As we have seen in the background Section, the `scif_register()` API call has the role of opening a window, a range of pages in the registered address space (RAS), ranging for the specified length and return an offset, from the start of the registered address space, to the caller that will be used by API functions involved in RMA operations in order to address the registered space intended for bilateral access. The SCIF driver retains a mapping of the registered window in the register address space (RAS) and the memory range in the virtual address space (VAS) that it corresponds to, as it is represented in Figure 2.9.

Due to the nature of the framework's distributed architecture, we have implemented a mechanism that stores and retrieves mappings between a registered address space offset pointing to a registered window and its corresponding virtual address space memory region. Each mapping is unique for every process context. During an RMA operation, both the client-side library and the server-side daemon will retrieve from their respective mappings data structure, the pointer to the virtual address space memory region that corresponds to the offset of the registered address space (RAS) that the RMA operation is utilizing, in order to transfer the data from and to the client before and after the completion of the RMA operation respectively. The distributed mechanism that we employ to support the SCIF RMA functionality is illustrated in Figure 3.3.

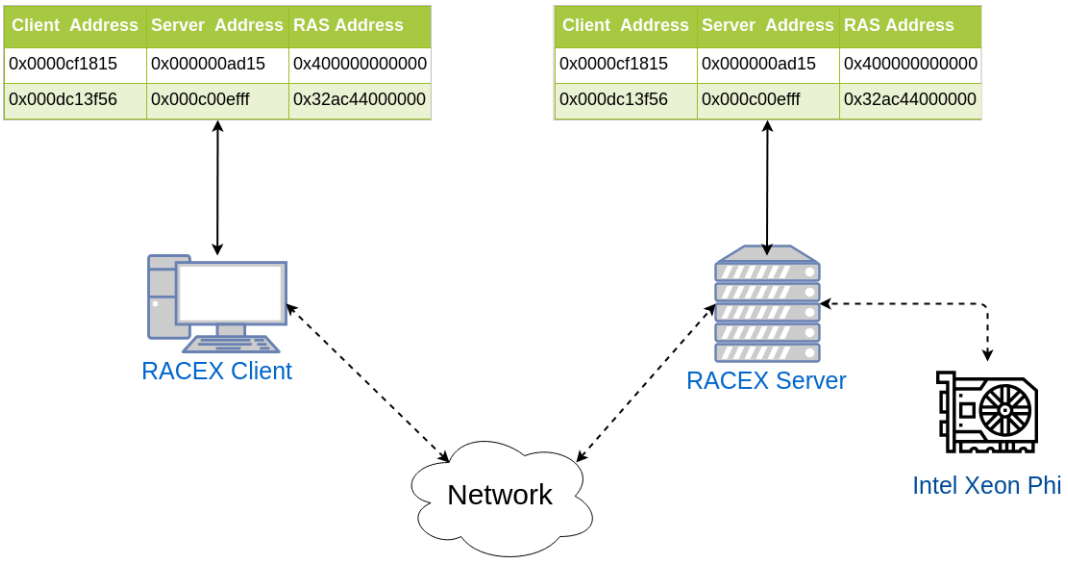


Figure 3.3: RACEX RMA Mappings Mechanism

RACEX is currently supporting all RMA operations at a synchronous execution mode, with all RMA related SCIF functions returning after the completion of the transfer operation.

Chapter 4

Evaluation

In this section we analyze the performance of our framework. First, we evaluate the execution time of our framework on the API level where we compare it with the native execution. We evaluate RACEX framework by conducting a series of microbenchmarks that focus on specific API calls. Then, we explore the performance of the RACEX framework on the native and offload execution mode of the Intel Xeon Phi accelerator, by running various workloads from the Rodinia Benchmark Suite for Heterogeneous Computing [37]. Finally, we conclude with a scalability test where we evaluate how are framework scales in comparison to the native case, for various thread and number of clients configurations.

4.1 Experimental Setup

We evaluate the performance of our framework by setting up the following topology: a server host machine with 1x Intel Core i7-4820K, 32GB RAM, also equipped with one Intel Xeon Phi 3120P coprocessor with 57 cores capable of hyperthreading, with 4 threads per core, summing up to a total 228 threads, which will be running the RACEX's server side daemon and another machine with 1x Intel Core i5-2320, 4GB RAM that RACEX library is installed and will act as the client in our evaluation scenarios. The two machines are connected via Ethernet on the same LAN. The aforementioned topology is depicted in Figure 4.1.

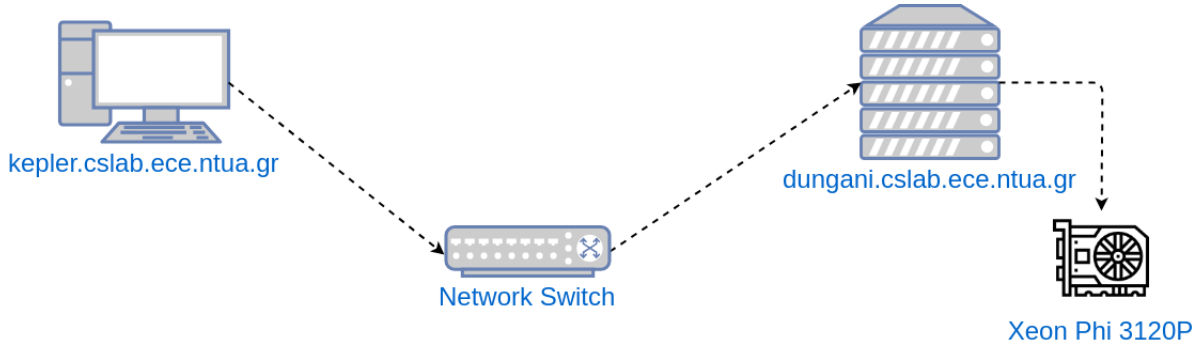


Figure 4.1: RACEX Experimental Setup Topology Overview

In order to evaluate our framework's performance in comparison to the native operation of the Xeon Phi, we conduct our test scenarios for the following network configurations:

1. Local: In this configuration, the client machine which utilizes the RACEX framework to gain access to Xeon Phi's resources is the same machine running the daemon application. In this configuration we simulate the minimum overhead latency that our framework features.

2. Remote: In this configuration, the client machine which utilizes the RACEX framework to gain access to Xeon Phi's resources is in the same LAN as the server machine. In this configuration we simulate the performance of our framework in a realistic environment.

4.2 Microbenchmarks

We use a simple of microbenchmarks, developed by the authors of vPHI[17], that focus on evaluating the performance of our framework during the two major communication operations that the SCIF employs for the host processor to communicate with the accelerator: (1) the Send-Recv socket-like API calls which comprise the Messaging Layer of SCIF, in which short command-type messages may be exchanged between the host and the coprocessor and (2) the Remote Memory Access SCIF API calls which are utilized for heavy bulk data transfers between the host processor and the accelerator [35]. For both benchmark scenarios, we first execute them natively on the server in order to acquire a baseline to which we will compare the performance of our framework in the two network configurations. Also, in order for the benchmarks to work, a corresponding application is spawned on the accelerator which will be the communicating party with which the client-side benchmark will communicate, either receiving data through the `scif_recv()` API call or by registering and reading from a remotely accessed memory region.

4.2.1 Messaging Layer Evaluation

First, we evaluate the latency of our framework by executing the Send-Recv benchmark scenario for different message sizes ranging from 1 Byte to 32K Bytes. In this scenario, the application spawned on the accelerator listens for incoming connection, accepts a connection from a potential client and receives a predefined number of bytes using the `scif_recv()` API call. On the client-host side, the application connects to the communicating process executing on the coprocessor and send the predefined number of bytes using the `scif_send()` API call. We present the corresponding latency that was measured for RACEX framework, for the two network configurations, in comparison with the native execution latency in the Figure 4.2.

In Figure 4.2 we observe that the latency of our framework shows a near constant overhead for message sizes smaller that 4K. The native execution latency for sending 1 Byte from the host to the coprocessor is 5 us. The latency of RACEX framework is 95 us for the localhost network configuration and 250 us for the remote network configuration. Hence, the overhead latency of our framework is calculated to $Localhost\_Overhead = (95 - 5) = 90us, Remote\_Overhead = (250 - 5) = 245us$. From the presented measurements, we experience a significant increase in the measured latency from the Localhost to the Remote network configuration. The noticeable difference between the two network configurations leads us to the conclusion that $245us - 90us = 155us$ is attributed to the network transmission overhead.

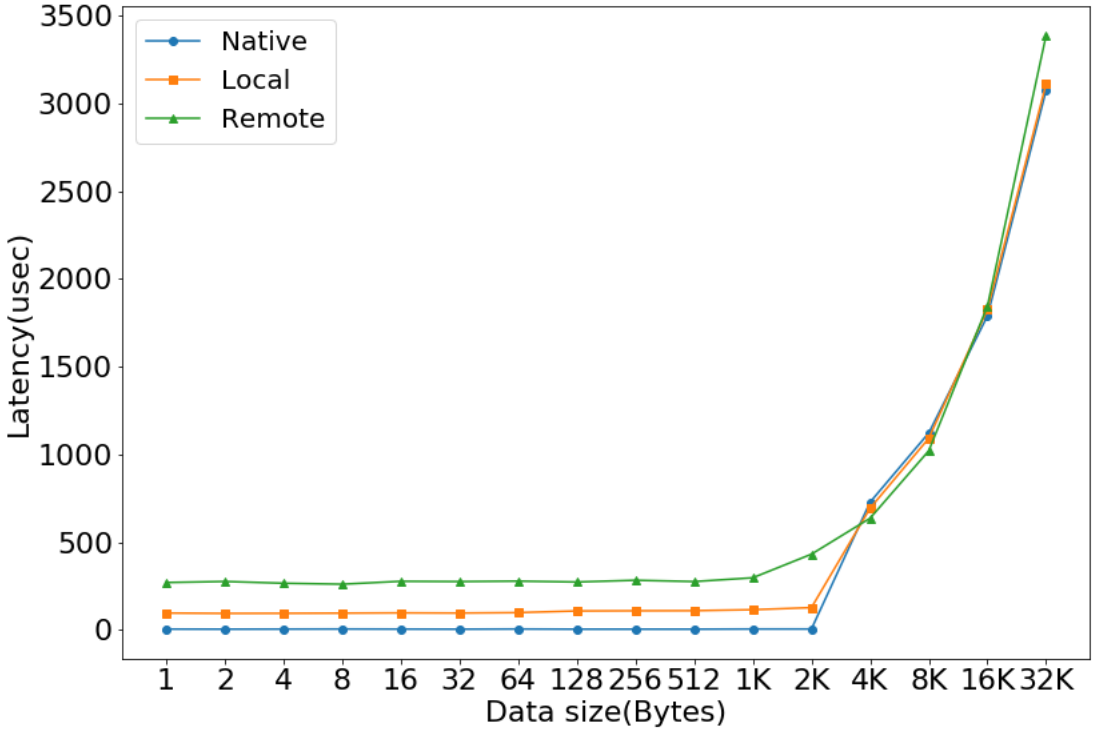


Figure 4.2: Send-Recv Communication Latency

Furthermore, we observe a sharp escalation in the measured latency of the native baseline measurements for message bigger than 4KB. The findings are in accordance with the SCIF specifications that state that for messages with size bigger than 4KB, SCIF RMA operations should be utilized for the data transfer, as the SCIF Messaging Layer that Send-Recv API calls are part of, is intended for small size command-type messages exchange between the processor and the accelerator [35]. Alongside the scaling of the native baseline latency, we observe that the latency of RACEX framework follows a similar escalation where we experience a near native performance for our framework for 4K up to 32K messages.

In order to better understand the overhead that our framework features for the Messaging Layer benchmark, we perform an in-depth breakdown analysis. The findings of our analysis are presented in Figure 4.3.

From the Breakdown analysis, we conclude that approximately 7% of the latency measured from our framework for messages smaller than 4K is attributed to the framework's inherent implementation overhead and 84% is attributed to the Network Transmission and TCP/IP Stack overhead. The main source of the overhead that the framework's inherent implementation presents is attributed to the serialization and deserialization operations that as we explained, contain data copy operations and we can abridge in our future work by utilizing a scatter-gather transfer mechanism. Moreover, for message sizes exceeding the 4K threshold, it is evident that the share of the total execution time that corresponds to the actual SCIF API call execution time is increasing,

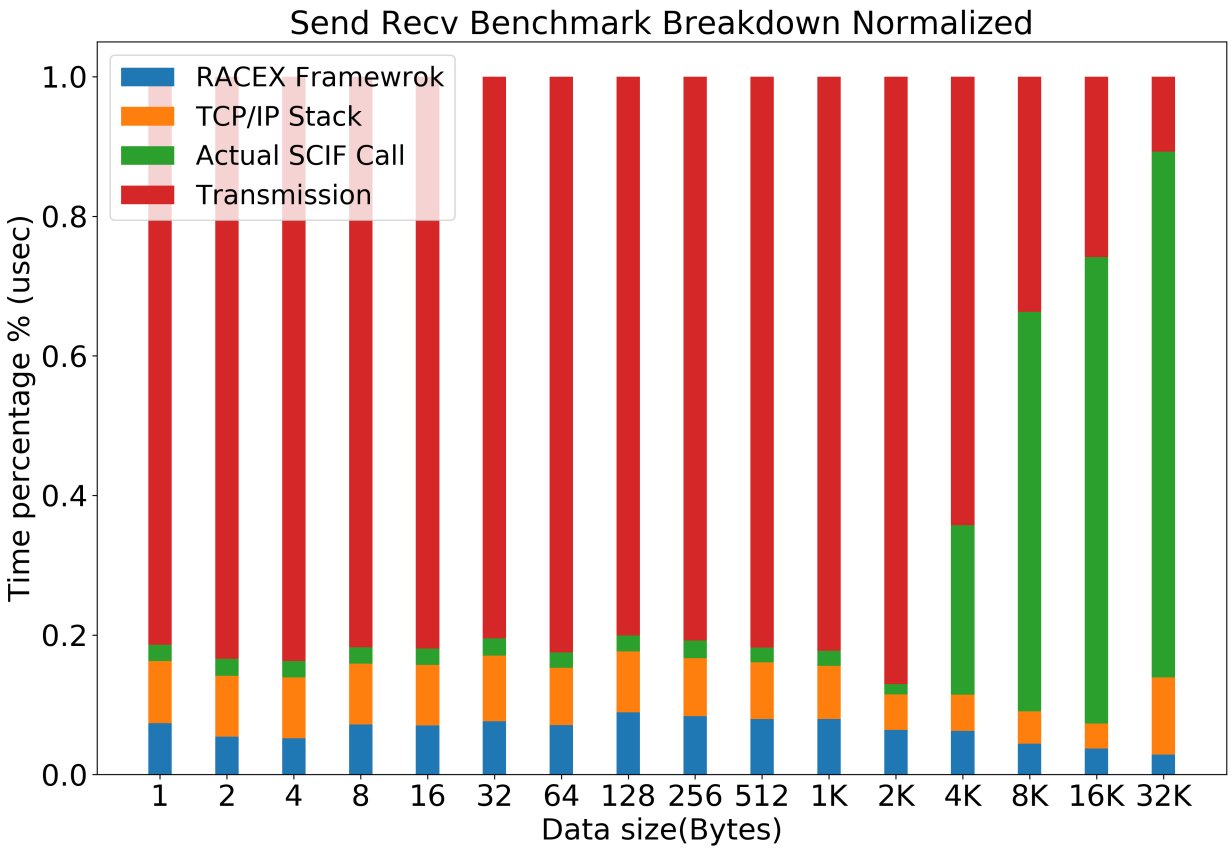


Figure 4.3: Send-Recv Communication Performance Breakdown

as expected, in a manner similar to the baseline native execution. It is worth noticing, that for 32K messages the percentage of the total execution time that is attributed to the actual SCIF API call is calculated to 80%, thus explaining the near-native performance that we observed for RACEX framework.

4.2.2 RMA Operations Evaluation

Next, we evaluate the performance of our framework during Remote Memory Access operations, employed by SCIF protocol for efficient large data transfers, in order to determine the throughput that RACEX framework can achieve. In this scenario, we again spawn a process on the accelerator which listens for incoming connections, accepts a new client and receives a request that registers a memory region comprised of a predefined number of pages which will then be accessed by the remote process on the host. Then, the client-host process of the benchmark registers a corresponding memory window of the same predefined number of pages in size and initiates a Remote Memory Access operation, through the `scif_writeto()` API call, during which a buffer present in the Register Address Space (RAS) of the host is transferred to the Register Address Space(RAS) of the coprocessor. The results from the throughput evaluation for RMA operations are presented in Figure 4.4.

The baseline performance of SCIF presents a throughput of 1.72 GB/s where as the RACEX

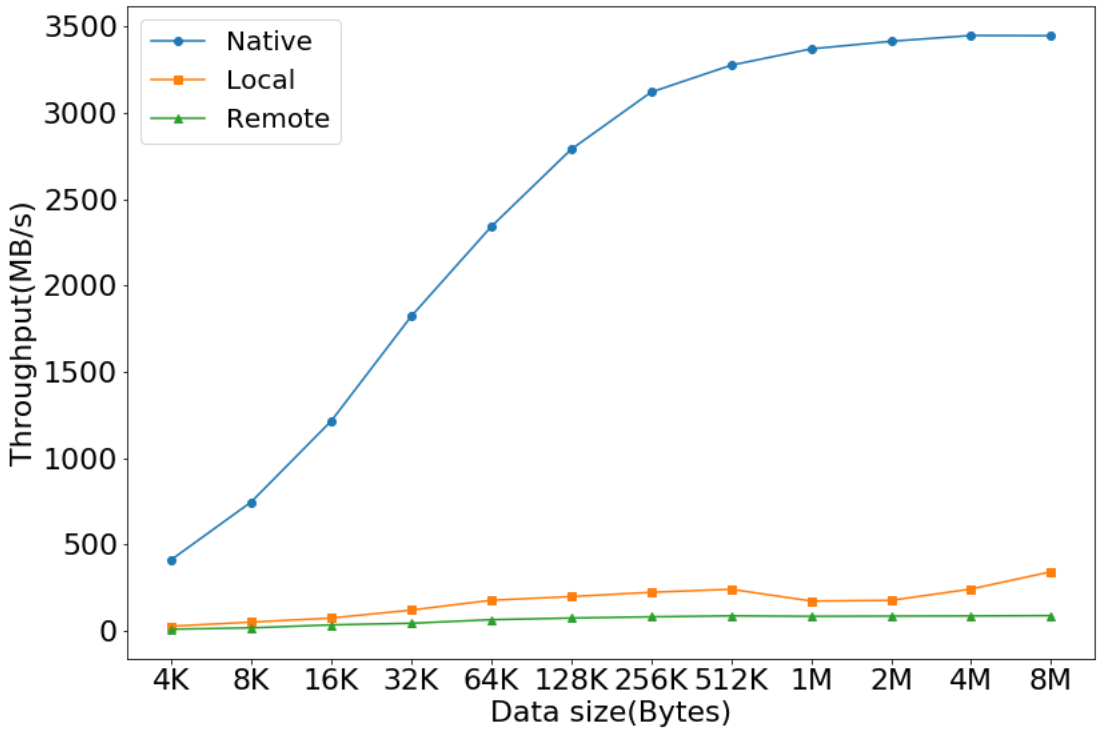


Figure 4.4: Read-Write RMA Operations Throughput

frameworks manages to attain 42.7 MB/s. In order to better understand the deficit of our framework in measured throughput for the RMA operation compared to the native execution we perform an in-depth performance analysis, the results of which are presented in Figure 4.5.

From the presented analysis, it is evident that the biggest share of overhead that is present in the execution time is attributed to network transmission delay and the TCP/IP Stack overhead. More specifically, as the size of data to be transferred through the RMA operation increases, the percentage of the total execution time that represents the network transmission delay abates. At the same time the percentage that represents the overhead of the TCP/IP Stack is steadily increasing as it becomes more prevalent between the other major operations of the execution. Together the Network Transmission and TCP/IP overhead account for approximately **72%** of the total execution overhead and the serialization operations that are employed in order to transmit the structured data account **21%**, thus explaining the observed abridged throughput in comparison the native baseline execution.

However, there is a perceived aberrant behavior between the Send-Recv Messaging Layer latency benchmark and that of RMA Operations throughput benchmark. There is a range of overlapping input data sizes, from 4KB to 32KB, that we evaluated with both SCIF communication mechanisms. As we previously established, the Send-Recv Messaging Layer benchmark presents a near-native performance for messages that range from 4KB to 32KB of size. On the other hand, the RMA benchmark presents a significant overhead to the measured native baseline performance.

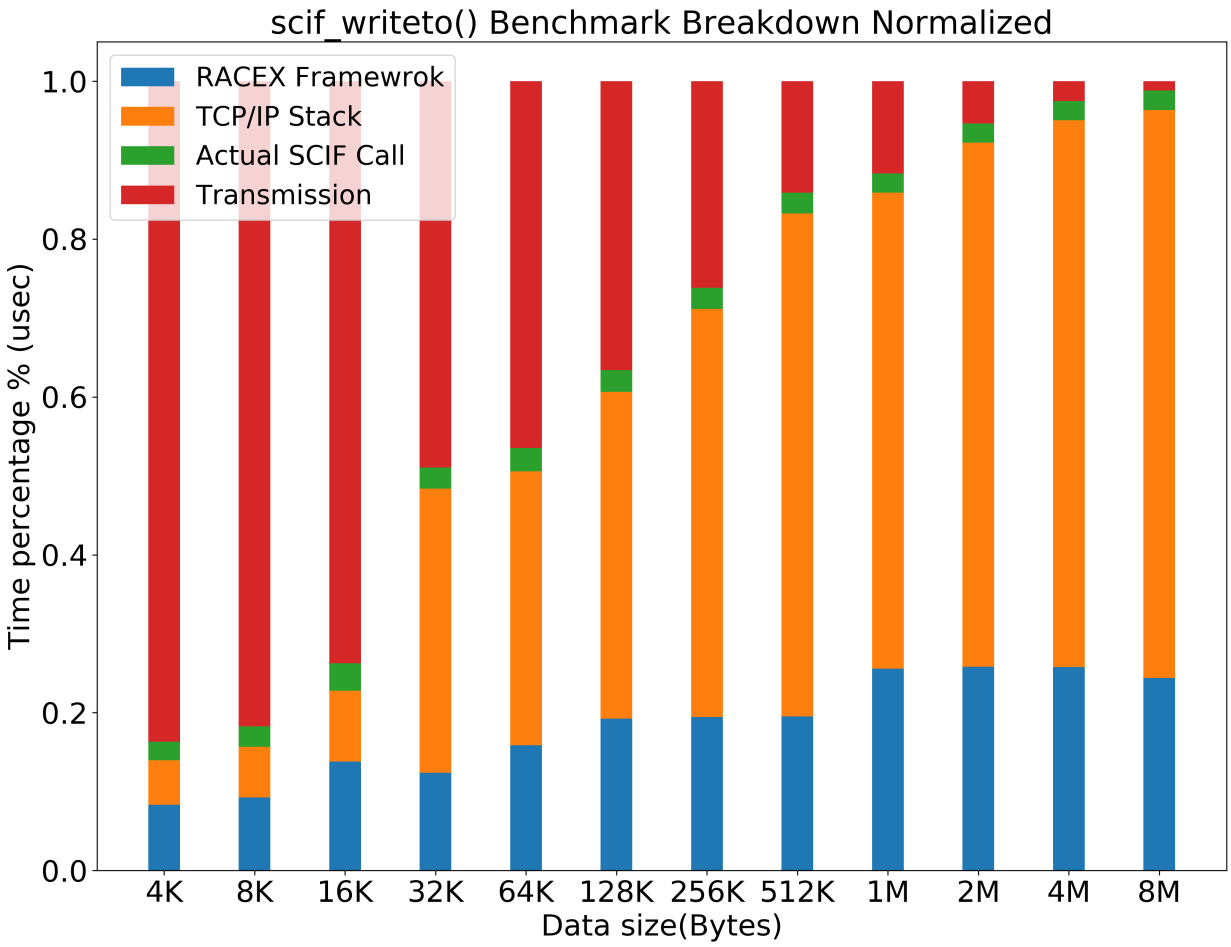


Figure 4.5: Read-Write RMA Operations Performance Breakdown

In order to understand this aberrant behavior, we focus on the results of the breakdown analysis for the specific input data sizes that overlaps between the two benchmarks. For 32KB input data size, we can observe that there is a substantial difference in the execution time of the actual native SCIF API call execution between the two benchmarks. In the Send-Recv Messaging Layer benchmark the execution time of the `scif_send()` call is 3078 us when the `scif_writeto()` RMA equivalent is 18 us for the same data size to be transferred. In this context, as we also observe from the breakdown analysis figures presented above, when the actual SCIF call execution time scales in the Send-Recv benchmark, the percentage of the total execution latency that belongs to the Transmission overhead abates. At the same time, in the RMA benchmark, the percentage of the total execution time that belongs to the Transmission latency holds the prominent position because the actual execution time of the SCIF call is staggeringly lower in contrast to Send-Recv. Having evaluated the two different presented behaviors, we conclude that network latency is the prominent cause for the low RMA throughput for large data transfers.

4.3 Native Execution Mode

We move forward and evaluate the performance of RACEX framework in higher-level applications. We evaluate the performance of RACEX framework in the native execution mode of the Intel Xeon Phi. In this execution mode, the application has to be cross-compiled in the host machine to target the Xeon Phi architecture and then transferred to the coprocessor from where it will be launched and executed. We utilize *micnativeloadex*, a software tool provided by Intel in the Manycore Platform Software Stack, in order to transfer the executable and all it's dependencies to the coprocessor and launch the remote process.

We have selected to stress the performance of our framework against workloads from the Rodinia Heterogeneous Benchmark Suite [37] that are preconfigured for offload execution on an Intel Xeon Phi. The workloads we have selected are: the Computational Fluid Dynamics (CFD) which is a unstructured-grid finite-volume solver for the three-dimensional Euler equations for compressible fluids, the HotSpot (HS) which is a thermal simulation tool used for estimating processors temperature, the LU Decomposition (LUD) which is an algorithm for calculating the solutions of a set of linear equations, the Needleman-Wunsch (NW) which is a global optimizations method for DNA sequence alignment and the Breadth-First Search (BFS) which traverses all the connected components in a graph. We explore the performance overhead of our framework for the aforementioned workloads with 56, 112 and 224 number of threads spawned on the accelerator in the two network configurations. The resulting normalized execution time per workload is presented in Figures 4.6.

The results presented in Figure 4.6 are normalized based on the performance of the baseline native execution that is executed directly on the host machine. As expected, the results show no performance degradation for RACEX compared to the host native baseline performance. Our framework is involved in the transfer of the executable into the coprocessor and the launch of the remote process. After the remote process is spawned, our framework is not involved in the execution of the application, as the execution mode dictates that the application will be executed solely on the accelerator and as-soon-as it's completed, the standard output will be redirected to the host. Thus, we expect the actual performance of the workloads we have evaluated not to show any significant performance degradation, as it has been proven. Any fluctuations present in the results that were presented, showing below native execution times for the RACEX framework, are inside the margin of error of the measurements and are negligible, as they are the mean value of repetitive measurements and their difference from the native baseline execution time is insignificant.

4.4 Offload Execution Mode

In this subsection, we evaluate the performance of RACEX framework in the offload execution mode of the Intel Xeon Phi. In this mode, the execution of an application is split between the host and the coprocessor with the former offloading computationally intensive parts to be executed on the later. On the coprocessor, a predefined number of threads are assigned to the spawned remote processes. The number of spawned threads and their assignment are mandated by user-defined environmental variables that are configured on the host before the execution of the intended application. We evaluate the performance of our framework in the offload execution mode by running the same workloads as before from the Rodinia Heterogeneous Benchmark Suite

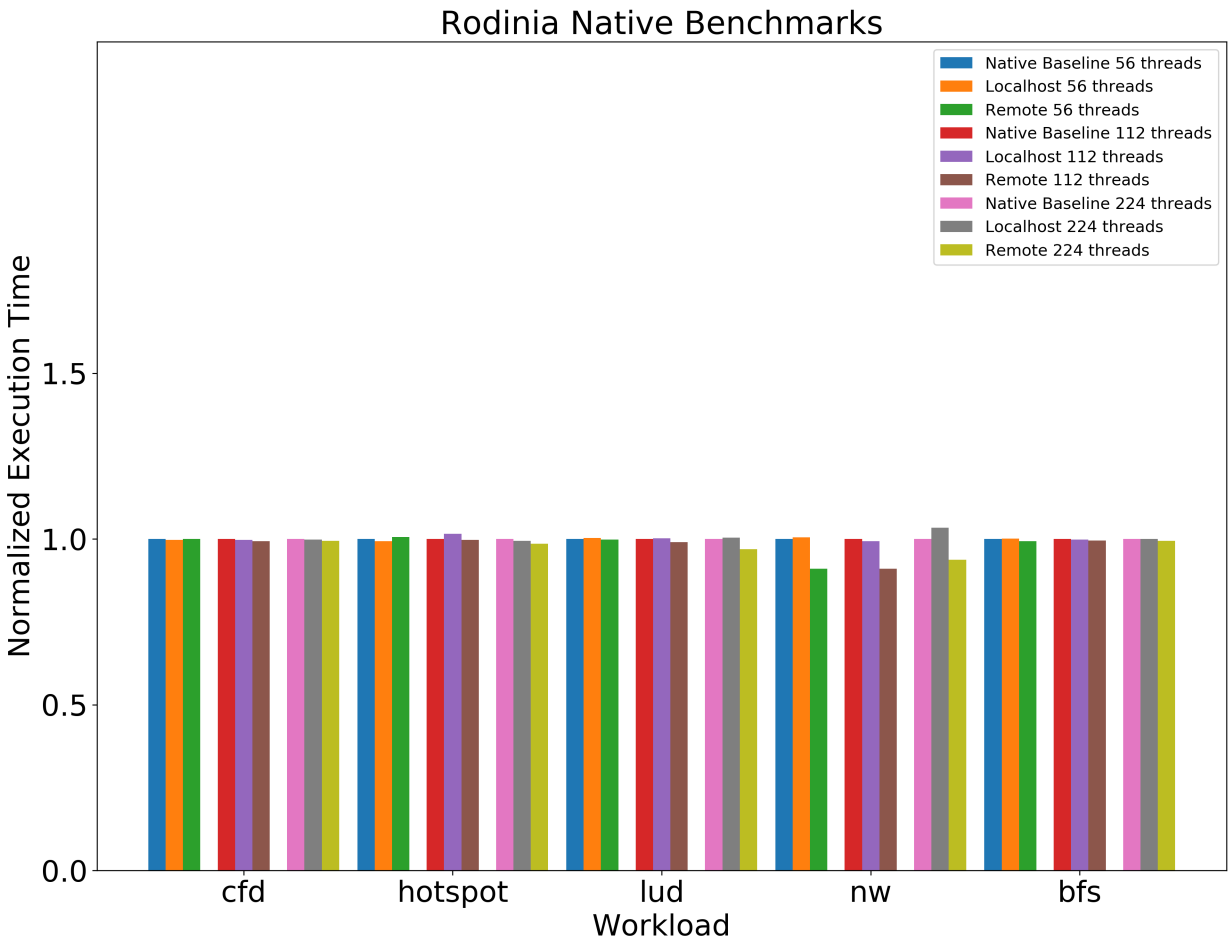


Figure 4.6: Rodinia Benchmarks Native Execution Mode

[37] and the same thread and network configurations. The resulting normalized execution time per workload is presented in Figure 4.7.

The underlying performance we observe from the different workloads is fluctuating between the two network configurations in comparison to the baseline native performance for the three different threads configurations. We observe a near-native performance for the CFD workload across all thread configurations and promising results from the LUD workload. The LUD workload presents near-native performance for the remote network configuration for 56 threads and an overhead in the performance is aggregating as we move to 112 and 224 threads. This can be explained if we take a closer look at the native baseline execution time of the workload for the three thread configurations. When we execute the LUD workload with 56 threads, the execution time is approximately 24 seconds whereas for 112 threads the execution time drops down to 11 seconds and even more so for 224 threads, where it performs a little under 4 seconds. Considering the overhead that our framework presents for the remote network configurations in the LUD workload, which remains constant at approximately 5 seconds, it is evident that for the baseline native performance with 56 threads, the overhead of our framework is negligible compared to the total execution time. However, as the native baseline execution time abates with increasing thread

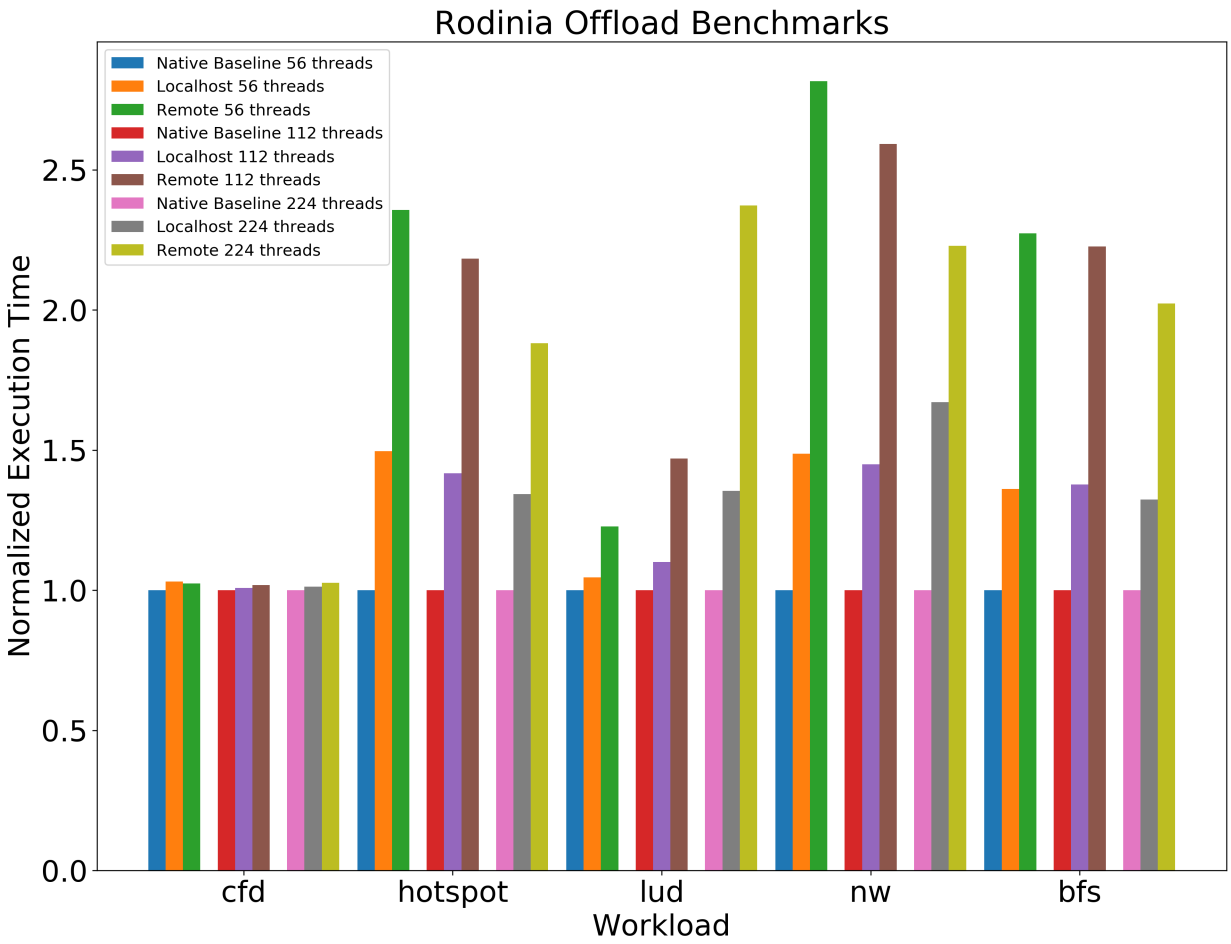


Figure 4.7: Rodinia Benchmarks Offload Execution Mode

count, the overhead of our framework becomes more and more significant. Hence, the resulting behavior depicted in Figure 4.7 for the LUD workload with 224 threads is reasoned. Furthermore, workloads such as HotSpot, Needleman-Wunsch and BFS have native baseline execution time that is low enough so that the network latency that our framework features has a significant toll to the total execution time of the workload as depicted in the presented Figure 4.7.

The fluctuating performance observed between the different workloads, with some workloads having a near-native performance and others experiencing a substantial overhead, is also partly traced back to the total number of offload operations that each workload includes. Workloads with multiple offload operations are burdened with the overhead of multiple network latencies, which as we explored in the microbenchmarks evaluation section are mainly responsible for the overhead latency of our framework. Accordingly, workloads with less offload operations experience less network and overall latency. For the previous remarks we consider a scenario where the size of the data offloaded to the coprocessor remains constant. Thus, we can conclude that due to the nature of the network transmission medium that our framework utilizes and the inherent latency it features, RACEX is well suited and performs better for workloads that include less offload operations. As the number of different offload operations increases, so does also the overhead of

our framework, due to the multiple network latencies.

4.5 Scalability Evaluation

Next, we evaluate the performance of RACEX during concurrent accesses to the accelerator's resources by multiple clients whilst executing a high-level application. We have selected a subset of the workloads that are included in the Rodinia Heterogeneous Benchmark Suite [37] which we already introduced in the previous evaluation sections. We execute the different workloads first in a native context on the host that the coprocessor is directly attached to in order to acquire the baseline scalability performance on the Intel Xeon Phi and then in localhost and remote network context, from which the actual performance of RACEX is obtained. Moreover, we evaluate how the performance differentiates for different number of spawned threads on the coprocessor(56, 112, 224) in conjunction with different thread affinity configuration. More specifically, we explore how the workloads perform when clients are assigned specific threads on the coprocessor, which are pinned to the remote processes that each respective client has launched on the coprocessor. The results of our scalability evaluation are depicted in Figure 4.8.

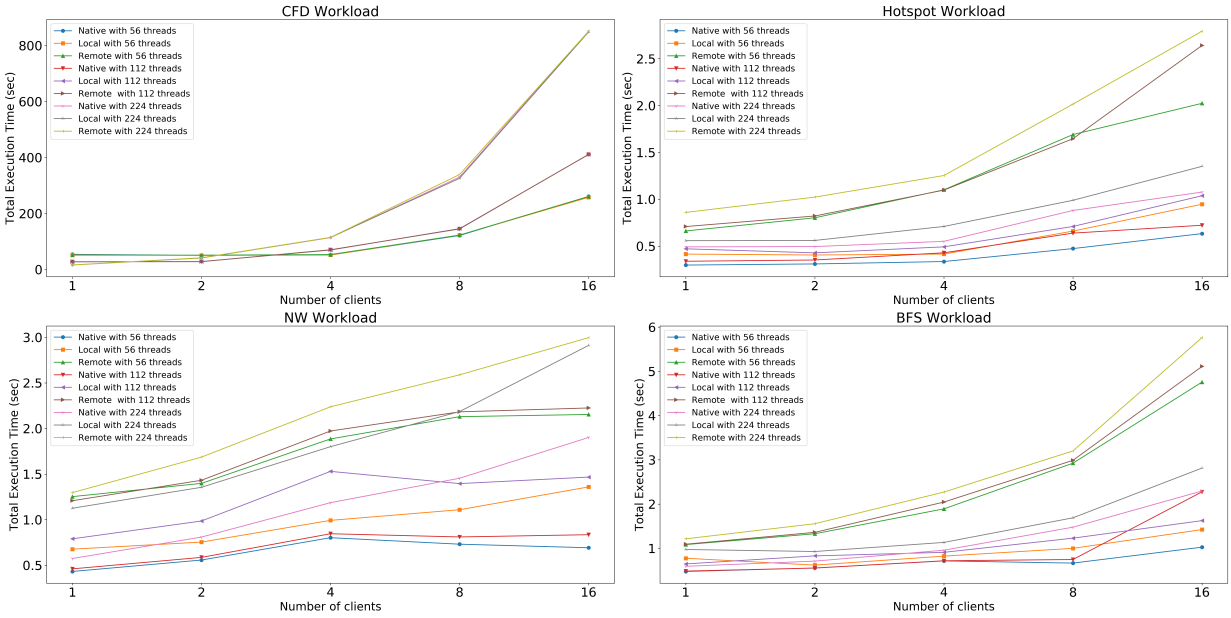


Figure 4.8: RACEX Framework Scalability Evaluation

We make several observations based on the results of the scalability evaluation.

First of all, as we stated in the previous sections, due to the extend of the total execution time of the CFD workload, we observe no performance degradation in the scalability of the RACEX framework in comparison to the native baseline performance scalability for the specific workload 4.9. Considering the scalability of the framework in conjunction with the different thread configurations we make the following remarks, taking CFD workload as an example. As we have clarified, the coprocessor we have executed our evaluation scenarios has 224 hardware threads(1 core dedicated to the operating system). From Figure 4.8 for the CFD workload where

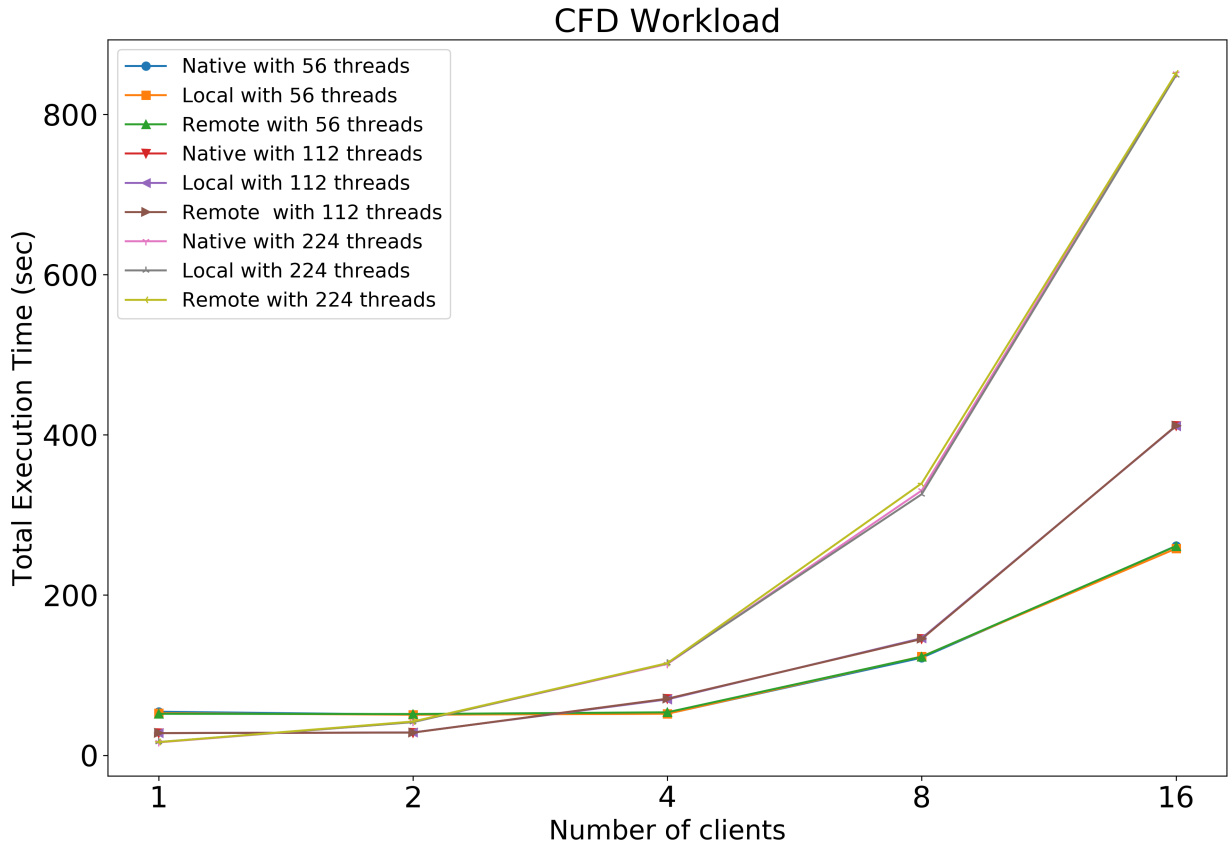


Figure 4.9: RACEX Framework Scalability Evaluation - CFD Workload

we experience the near-native performance for our framework, it is evident that the plotted lines are grouped in three groups, corresponding to the three different thread configurations. During our scalability evaluations scenarios, we were assigning hardware threads to the processes initiated on the coprocessor by the respective remote clients, in a circular manner until all threads had been assigned and then, in case of additional client processes, overlapping assignments would take place. For example, when each process spawns 56 threads, we pin the first 56 threads of the coprocessors to the first client and the next 56 threads to the next client process. Subsequently, for four clients, we spread the demand for the coprocessor's threads in order to fully utilize the capacity of the hardware. Thus, as we observe from the Figure 4.8 for the CFD workload, there is now performance differentiation from 1 to 4 clients for 56 threads configurations and then, as we move to 8 and 16 concurrent clients, we experience significant performance degradation, as we have overlapping hardware threads assignment to processes. Accordingly, similar behavior is notted for the 112 and 224 threads configuration where performance degrades as the number of client processes assigned to the same subset of threads increases.

Similar remarks concerning the performance degradation in comparison to the thread affinity on the coprocessor can also be drawn from the rest of the workloads. However, as we have stated before, the execution time of the rest of the workloads depicted in Figure 4.8 and Figures 4.10,4.11, 4.12 have sub-second execution time and at the same time have multiple independent

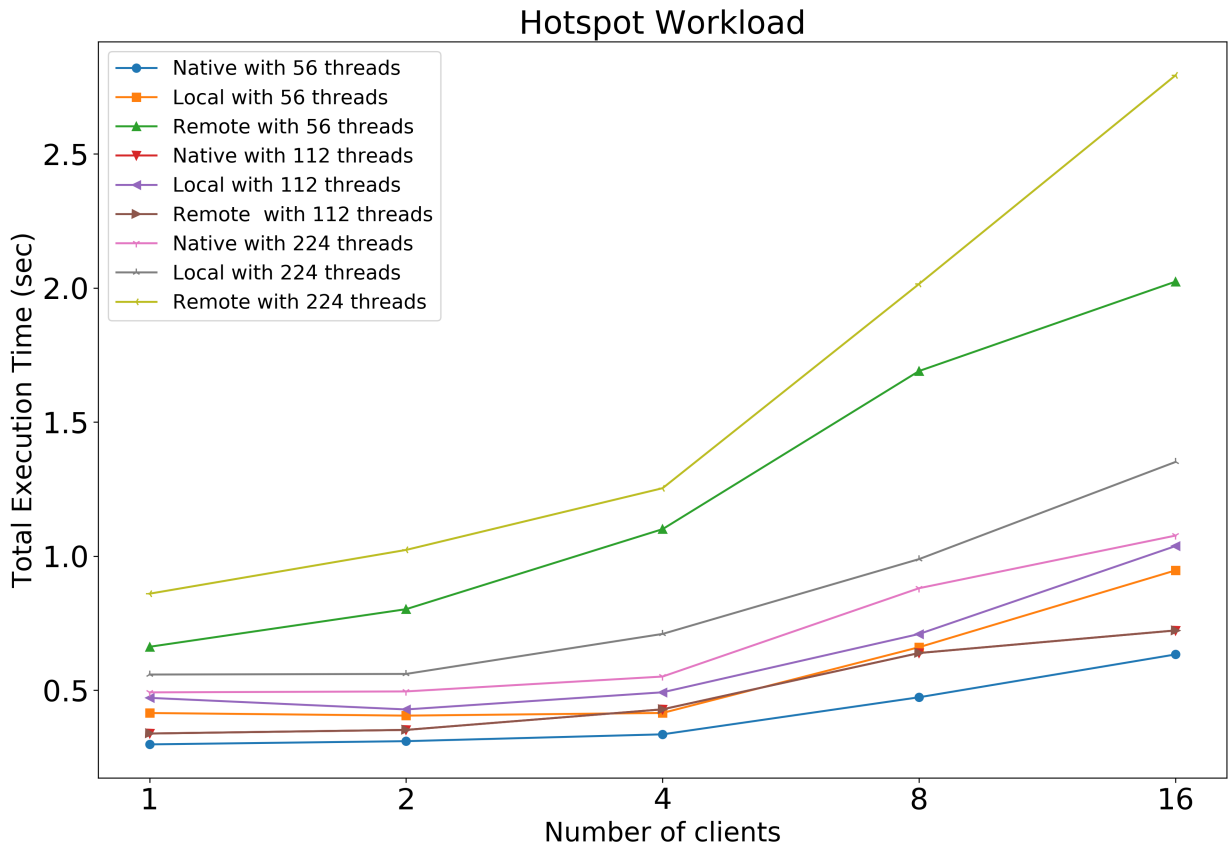


Figure 4.10: RACEX Framework Scalability Evaluation - HotSpot Workload

offload operations that multiply the total network latency, hence the existing network latency has a significant toll to the overall performance.

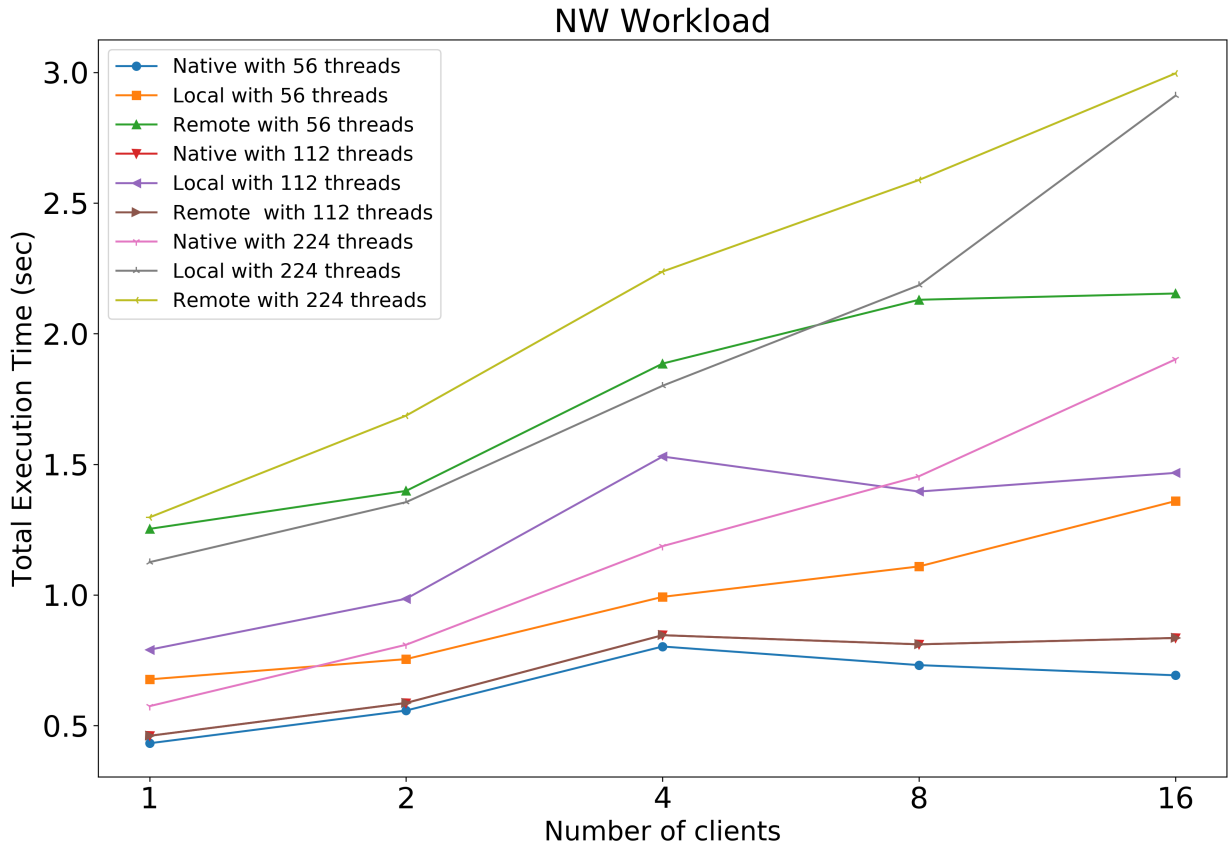


Figure 4.11: RACEX Framework Scalability Evaluation - NW Workload

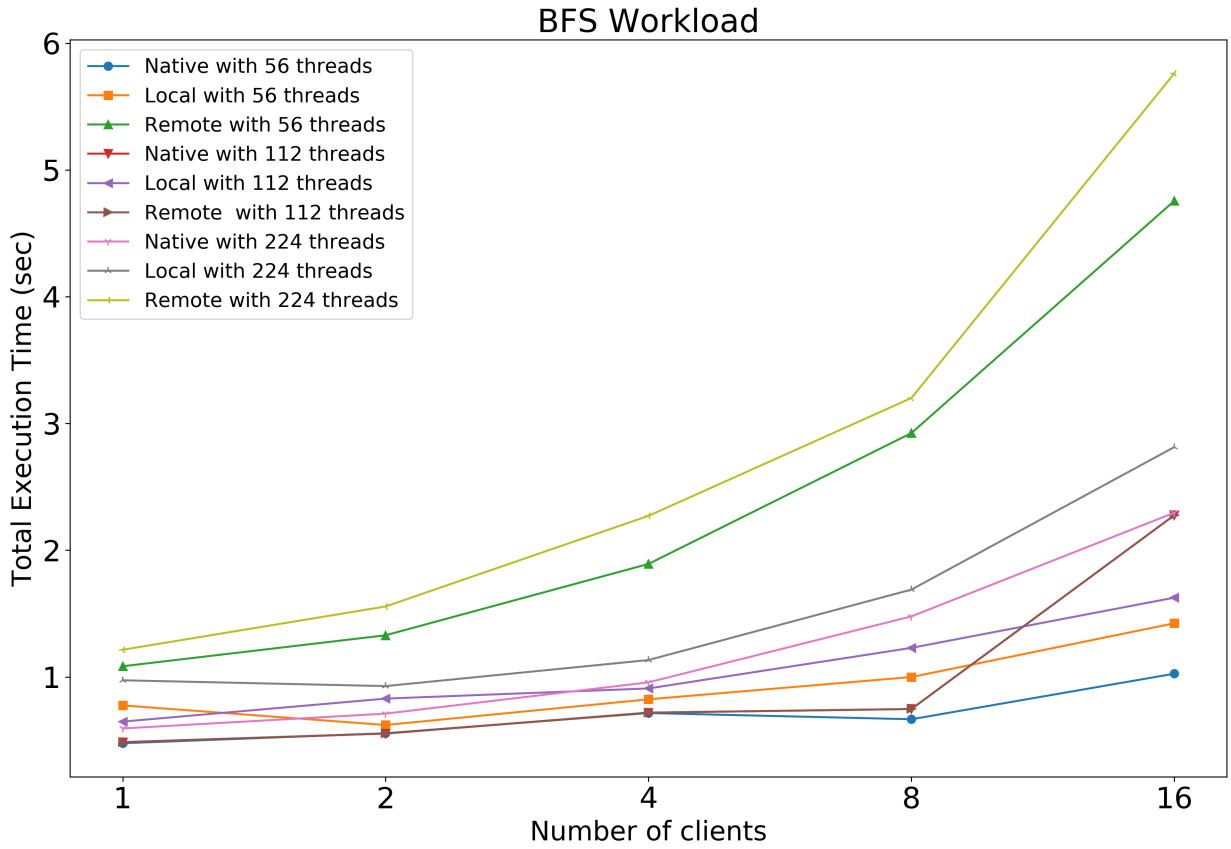


Figure 4.12: RACEX Framework Scalability Evaluation - BFS Workload

Chapter 5

Discussion

High-speed communication between the nodes of a computing cluster is vital for modern supercomputers and data centers in order to achieve the performance that current High Performance Computing applications demand. It is also one of the reasons that in the past the execution of HPC applications on the cloud had been challenged as there was a lack of high-speed connectivity services offered by providers between the different VM instances. Although this problem continues to curtail the adoption of Cloud Computing services by the HPC community, improvements have been made, with providers now offering high bandwidth, low latency networking features for their clients through utilizing high-speed interconnects and specific topological placement of the VMs [38, 39].

In this work, we have proposed RACEX as a Remote Accelerator Execution framework for computing clusters in cloud computing environments that will enable the sharing of accelerating resources on a cluster level, either by granting access to virtual machines on the same server that the accelerator is attached to or to remote cluster nodes. We have implemented a proof-of-concept in order to evaluate our proposed framework, by targeting the Intel Xeon Phi co-processor, utilizing BSD Sockets and Ethernet standardized network connectivity between the host server that the coprocessor is attached to and the client node we used for our evaluation. As we have seen in the evaluation section, the biggest share of the overhead that our framework presents is attributed to the overall network latency comprised by the Transmission Time and the TCP/IP Stack. We argue that with the employment of high performance network interconnect such as InfiniBand and Intel's OmniPath [40] that feature high throughput and low latency, our framework can mitigate the overhead latency present in our framework and present near-native performance.

Moreover, it's fundamental for an accelerator sharing framework for cloud computing environments to be able to provide to the various virtual machines hosted on the same cluster node in which the accelerator is attached, access to the accelerator resources. In our framework we support the aforementioned functionality through regular BSD Sockets, the TCP/IP stack and a virtual network between the VMs and the Host OS. However, as we have explored, the employment of the Network Stack for communication between clients and server daemon is not ideal for the intended usage due to the overhead latency it presents. Instead, we argue that we can integrate in our work, a low-overhead, high-performing intra-node communication framework for co-located VMs, such as V4VSockets [41] which improves the intra-node data exchange in terms of latency and throughput by a factor of 4.5, thus enabling RACEX to provide an efficient high-performing and scalable accelerator sharing framework for co-located VMs.

Chapter 6

Related Work

Providing remote access to accelerators has been a problem that the academia has been occupied with for some time. The most prominent approaches, that have been established in the past years in making accelerators remotely accessible, are those that target GPUs.

6.1 rCUDA

Perhaps the most popular amongst the proposed solutions and the one that has moved beyond the scope of academia and has become commercially available is rCUDA [18]. rCUDA focuses on NVIDIA's CUDA framework, the parallel computing platform and programming model that NVIDIA offers to leverage the computational power of their GPUs. The framework enables the concurrent access to CUDA compatible GPUs remotely. In order to provide remote CUDA services, rCUDA creates virtual CUDA compatible devices on the devices that don't have a CUDA compatible GPU and request access to the resources of a remote CUDA compatible device. rCUDA virtualizes the GPUs present in computing center and provides access to multiple client by utilizing a Client-Server architecture. In this work, CUDA run-time calls are intercepted and forwarded to a remote server that executes them and returns the results. In our work, in comparison to rCUDA, we target the low-level Transport Layer communication between the host and the accelerator, operated by SCIF protocol. We are abstracting away the Transport Layer between the host and accelerator, hence moving towards a centralized accelerator management schema for computing clusters. Moreover, we have published RACEX as an open-source software.

6.2 vCUDA

vCUDA [42] is a general-purpose graphics processing unit (GPGPU) computing solution for virtual machines (VMs). vCUDA allows applications executing within VMs to leverage hardware acceleration from underlying Graphics Proccessing Units. The design of the framework includes an API call interception and redirection and a dedicated RPC system for VMs. vCuda uses a client-server architecture consisted of three user-space components: a user-level library, a data structure used by the library, that represents a virtual GPU, and a server component. The library is responsible for intercepting and redirecting API calls from client to host, where the server executes them and returns the results. Communication between client and server is implemented using

the XML-RPC protocol. In addition to virtualization, vCUDA allows device multiplexing among multiple concurrently executing guest OSes by spawning one thread for each client. The framework provides support for suspend and resume as well, enabling client sessions to be interrupted or moved between computers. The system is implemented in Xen but is portable across different virtualization platforms due toits network transmission mechanism. Experiments in [42] show that time spent in the encoding-decoding steps of the communication protocol causes a considerable negative impact on the overall performance of the solution.

6.3 Distributed-Shared CUDA

DS-CUDA [19] (Distributed-shared compute unified device architecture),is a middleware that enables the use of many GPUs in a cloud environment. It virtualizes GPUs in a cloud such that they appear to be locally installed GPUs in a client machine. The authors present a middleware with the goal to address difficulties in programming multi-node heterogeneous computers. The system implements virtualization of a cluster of computers equipped with GPUs so that they appear as if they were attached to a single node, in order to simplify the programming of multi-GPU applications. The system's architecture consists of a single client node and multiple server nodes, in which one or more CUDA devices are installed. Communication between the remote clients and the server is implemented over InfiniBand Verbs, and can also use TCP sockets in case the network infrastructure does not support InfiniBand. DS-CUDA increases the reliability of GPUs by implementing a redundancy mechanism. Identical calculations are performed on multiple CUDA devices, and the results are compared between the redundant calculations. If any of the results do not match an error handler is invoked.

6.4 Other Related Work

Moreover, in the Cloud Computing realm, providers have already established a market where they provide elastic access to accelerator in order to satisfy the increasing needs of demanding clients that want to leverage the massively parallel. In the IoT community, a plethora of research work has been carried out indicating the need for providing offload capabilities in the Cloud Computing context for low power, internet connected devices [10, 11, 43].

Chapter 7

Conclusions

In our work, we propose RACEX as an efficient framework for enabling remote access to accelerator resources. We implement RACEX as a proof-of-concept targeting the Intel Xeon Phi co-processor. RACEX framework follows a distributed Server-Client architecture, where a server-side daemon can support the concurrent access to the Intel Xeon Phi resources by multiple remote clients. RACEX employs a API call forwarding mechanism, using a custom proprietary communication protocol in order to enable remote execution of Intel's SCIF API calls. The SCIF communication protocol resides on-top of the Transport Layer, thus making RACEX, which wraps and makes SCIF remotely available, flexible, efficient and compatible with higher software stack applications without the need of re-compilation. Preliminary performance evaluation has shown a near-constant overhead present for RACEX compared to the native baseline performance, mainly due to the Network Transmission Latency. However, the evaluation of higher level applications, part of the Rodinia Heterogeneous Computing Benchmark suits has delivered promising results, with RACEX framework performing at low overhead compared to the native baseline performance and can serve multiple concurrent clients with the same behavior.

We implement and evaluate RACEX to support both the *native* and *offload* modes of execution for the Intel Xeon Phi coprocessor. As a future work, we plan to extend our implementation in order to support the *symmetric* mode of execution and enable the use of the MPI framework. Such expansion is easily implemented as the underlying infrastructure of our framework has been designed to support the future *symmetric* execution mode compatibility. Moreover, we plan to utilize Infiniband technology as a High Performing Network Interconnect in order to minimize the network latency inherent in our initial implementation and be able to achieve near-native performance. We have made our work available as open source software, available online at `https://github.com/fertakis/racex`.

Appendix A

Appendix B

Notation und Abbreviations

RACEX	Remote ACcelerator EXecution
SCIF	Symmetric Communication Interface
COI	Coprocessor Offload Infrastructure
MIC	Many Integrated Core
MPSS	ManyCore Platform Software Stack
RMA	Remote Memory Access
DMA	Direct Memory Access
BSD	Berkeley Software Distribution
API	Application Programming Interface

Bibliography

[1] S. Crago, K. Dunn, P. Eads, L. Hochstein, D. I. Kang, M. Kang, D. Modium, K. Singh, J. Suh, and J. P. Walters, "Heterogeneous cloud computing," in *2011 IEEE International Conference on Cluster Computing*, pp. 378–385, Sept 2011.

[2] E. Deelman, G. Singh, M. Livny, B. Berriman, and J. Good, "The cost of doing science on the cloud: The montage example," in *2008 SC - International Conference for High Performance Computing, Networking, Storage and Analysis*, pp. 1–12, Nov 2008.

[3] L. Wang, J. Tao, M. Kunze, A. C. Castellanos, D. Kramer, and W. Karl, "Scientific cloud computing: Early definition and experience," in *2008 10th IEEE International Conference on High Performance Computing and Communications*, pp. 825–830, Sept 2008.

[4] Y. Low, D. Bickson, J. Gonzalez, C. Guestrin, A. Kyrola, and J. M. Hellerstein, "Distributed graphlab: A framework for machine learning and data mining in the cloud," *Proc. VLDB Endow.*, vol. 5, pp. 716–727, Apr. 2012.

[5] A. M. Caulfield, E. S. Chung, A. Putnam, H. Angepat, J. Fowers, M. Haselman, S. Heil, M. Humphrey, P. Kaur, J. Y. Kim, D. Lo, T. Massengill, K. Ovtcharov, M. Papamichael, L. Woods, S. Lanka, D. Chiou, and D. Burger, "A cloud-scale acceleration architecture," in *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pp. 1–13, Oct 2016.

[6] Google, "Accelerated cloud computing." `https://cloud.google.com/gpu/`, 2018.

[7] Nvidia, "Gpu cloud computing." `https://www.nvidia.com/en-us/data-center/gpu-cloud-computing/`, 2018.

[8] Amazon, "High performance computing on aws."

[9] C. Evangelinos and C. N. Hill, "Cloud computing for parallel scientific hpc applications: Feasibility of running coupled atmosphere-ocean climate models on amazon's ec2," vol. 2, 01 2008.

[10] R. M. Shukla and A. Munir, "An efficient computation offloading architecture for the internet of things (iot) devices," in *2017 14th IEEE Annual Consumer Communications Networking Conference (CCNC)*, pp. 728–731, Jan 2017.

[11] F. Samie, V. Tsoutsouras, L. Bauer, S. Xydis, D. Soudris, and J. Henkel, "Computation offloading and resource allocation for low-power iot edge devices," in *2016 IEEE 3rd World Forum on Internet of Things (WF-IoT)*, pp. 7–12, Dec 2016.

[12] V. T. Ravi, M. Becchi, G. Agrawal, and S. Chakradhar, "Supporting gpu sharing in cloud environments with a transparent runtime consolidation framework," in *Proceedings of the 20th International Symposium on High Performance Distributed Computing*, HPDC '11, (New York, NY, USA), pp. 217–228, ACM, 2011.

[13] N. Haydel, S. Gesing, I. Taylor, G. Madey, A. Dakkak, S. G. d. Gonzalo, and W. M. W. Hwu, "Enhancing the usability and utilization of accelerated architectures via docker," in *2015 IEEE/ACM 8th International Conference on Utility and Cloud Computing (UCC)*, pp. 361–367, Dec 2015.

[14] Y. Shen, M. Ferdman, and P. Milder, "Overcoming resource underutilization in spatial cnn accelerators," in *2016 26th International Conference on Field Programmable Logic and Applications (FPL)*, pp. 1–4, Aug 2016.

[15] F. Silla, J. Prades, S. Iserte, and C. Reaño, "Remote gpu virtualization: Is it useful?," in *2016 2nd IEEE International Workshop on High-Performance Interconnection Networks in the Exascale and Big-Data Era (HiPINEB)*, pp. 41–48, March 2016.

[16] S. Mislata and F. Silla, "Using remote accelerators to improve the performance of the fftw library," in *2016 IEEE 18th International Conference on High Performance Computing and Communications; IEEE 14th International Conference on Smart City; IEEE 2nd International Conference on Data Science and Systems (HPCC/SmartCity/DSS)*, pp. 913–920, Dec 2016.

[17] S. Gerangelos and N. Koziris, "vphi: Enabling xeon phi capabilities in virtual machines," in *2017 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, pp. 1333–1340, May 2017.

[18] J. Duato, A. J. Peña, F. Silla, R. Mayo, and E. S. Quintana-Ortí, "rcuda: Reducing the number of gpu-based accelerators in high performance clusters," in *2010 International Conference on High Performance Computing Simulation*, pp. 224–231, June 2010.

[19] M. Oikawa, A. Kawai, K. Nomura, K. Yasuoka, K. Yoshikawa, and T. Narumi, "Ds-cuda: A middleware to use many gpus in the cloud environment," in *2012 SC Companion: High Performance Computing, Networking Storage and Analysis*, pp. 1207–1214, Nov 2012.

[20] R. D. Lauro, F. Giannone, L. Ambrosio, and R. Montella, "Virtualizing general purpose gpus for high performance cloud computing: An application to a fluid simulator," in *2012 IEEE 10th International Symposium on Parallel and Distributed Processing with Applications*, pp. 863–864, July 2012.

[21] Wikipedia, "Digital revolution," Dec. 2017.

[22] G. E. Moore, "Cramming more components onto integrated circuits," *Electronics*, Apr. 1965.

[23] G. M. Amdahl, "Validity of the single processor approach to achieving large scale computing capabilities," in *Proceedings of the April 18-20, 1967, Spring Joint Computer Conference*, AFIPS '67 (Spring), (New York, NY, USA), pp. 483–485, ACM, 1967.

[24] M. J. Flynn, "Some computer organizations and their effectiveness," *IEEE Transactions on Computers*, vol. C-21, pp. 948–960, Sept 1972.

[25] NVIDIA, "Printer friendlynvidia launches the world's first graphics processing unit: Geforce 256."

[26] O. J. D., L. David, G. Naga, H. Mark, K. Jens, L. A. E., and P. T. J., "A survey of general-purpose computation on graphics hardware," *Computer Graphics Forum*, vol. 26, no. 1, pp. 80–113.

[27] T. Messay, C. Chen, R. Ordóñez, and T. M. Taha, "Gpgpu acceleration of a novel calibration method for industrial robots," in *Proceedings of the 2011 IEEE National Aerospace and Electronics Conference (NAECON)*, pp. 124–129, July 2011.

[28] C. L. Hung, P. C. Wu, H. H. Wang, and C. Y. Lin, "Efficient parallel muti-pattern matching using gpgpu acceleration for packet filtering," in *2015 IEEE 17th International Conference on High Performance Computing and Communications, 2015 IEEE 7th International Symposium on Cyberspace Safety and Security, and 2015 IEEE 12th International Conference on Embedded Software and Systems*, pp. 1843–1847, Aug 2015.

[29] K. L. Rice, T. M. Taha, K. M. Iftekharuddin, K. Anderson, and T. Salan, "Gpgpu acceleration of cellular simultaneous recurrent networks adapted for maze traversals," in *The 2011 International Joint Conference on Neural Networks*, pp. 2717–2724, July 2011.

[30] K. Groups, "The open standard for parallel programming of heterogeneous systems."

[31] NVIDIA, "About cuda."

[32] Intel, "Best known methods for using openmp on intel many integrated core (intel mic) architecture." https://software.intel.com/en-us/articles/best-known-methods-for-using-openmp-on-intel-many-integrated-core-intel-mic-architecture, 2013.

[33] TOP500, "The fiftieth top500 list of the fastest supercomputers in the world.." `https://www.top500.org/lists/2017/11/`, 2017.

[34] Intel, "Intel manycore platform software stack."

[35] Intel, "Scif user guide."

[36] G. Developers, "Protocol buffers."

[37] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, S. H. Lee, and K. Skadron, "Rodinia: A benchmark suite for heterogeneous computing," in *2009 IEEE International Symposium on Workload Characterization (IISWC)*, pp. 44–54, Oct 2009.

[38] A. AWS, "Placement groups."

[39] A. AWS, "Ec@ encanced networking."

[40] Intel, "Intel omnipath architecture."

[41] A. Nanos, S. Gerangelos, I. Alifieraki, and N. Koziris, "V4vsockets: Low-overhead intra-node communication in xen," in *Proceedings of the 5th International Workshop on Cloud Data and Platforms*, CloudDP '15, (New York, NY, USA), pp. 1:1–1:6, ACM, 2015.

[42] L. Shi, H. Chen, J. Sun, and K. Li, "vcuda: Gpu-accelerated high-performance computing in virtual machines," *IEEE Transactions on Computers*, vol. 61, pp. 804–816, June 2012.

[43] H. Guo, J. Liu, and H. Qin, "Collaborative mobile edge computation offloading for iot over fiber-wireless networks," *IEEE Network*, vol. 32, pp. 66–71, Jan 2018.

ΕΘΝΙΚΟ ΜΕΤΣΟΒΙΟ ΠΟΛΥΤΕΧΝΕΙΟ
ΤΜΗΜΑ ΗΛΕΚΤΡΟΛΟΓΩΝ ΜΗΧΑΝΙΚΩΝ ΚΑΙ ΜΗΧΑΝΙΚΩΝ ΥΠΟΛΟΓΙΣΤΩΝ

ΤΟΜΕΑΣ ΤΕΧΝΟΛΟΓΙΑΣ ΠΛΗΡΟΦΟΡΙΚΗΣ ΚΑΙ ΥΠΟΛΟΓΙΣΤΩΝ
ΕΡΓΑΣΤΗΡΙΟ ΥΠΟΛΟΓΙΣΤΙΚΩΝ ΣΥΣΤΗΜΑΤΩΝ

Σχεδιασμός και Υλοποίηση Συστήματος Απομακρυσμένης Πρόσβασης Συνεπεξεργαστών

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

του

Κωνσταντίνος Γ. Φερτάκης

Επιβλέπων: Γεώργιος Γκούμας
Επίκουρος Καθηγητής

Αθήνα, Ιούλιος 2018

Εθνικο Μετσοβιο Πολυτεχνειο
ΤΜΗΜΑ ΗΛΕΚΤΡΟΛΟΓΩΝ ΜΗΧΑΝΙΚΩΝ ΚΑΙ ΜΗΧΑΝΙΚΩΝ ΥΠΟΛΟΓΙΣΤΩΝ
ΤΟΜΕΑΣ ΤΕΧΝΟΛΟΓΙΑΣ ΠΛΗΡΟΦΟΡΙΚΗΣ ΚΑΙ ΥΠΟΛΟΓΙΣΤΩΝ
ΕΡΓΑΣΤΗΡΙΟ ΥΠΟΛΟΓΙΣΤΙΚΩΝ ΣΥΣΤΗΜΑΤΩΝ

Σχεδιασμός και Υλοποίηση Συστήματος Απομακρυσμένης Πρόσβασης Συνεπεξεργαστών

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

του

Κωνσταντίνος Γ. Φερτάκης

Επιβλέπων: Γεώργιος Γκούμας
Επίκουρος Καθηγητής

Εγκρίθηκε από την τριμελή εξεταστική επιτροπή την $13_{η}$ Ιουλίου 2018.

.....	.....	.....
Ν. Κοζύρης	Γ. Γκούμας	Δ. Τσουμάκος
Καθηγητής Ε.Μ.Π.	Επ. Καθηγητής Ε.Μ.Π	Αν. Καθηγητής Ιόνιο Πανεπιστήμιο

Αθήνα, Ιούλιος 2018.

.....

Κωνσταντίνος Γ. Φερτάκης
Διπλωματούχος Ηλεκτρολόγος Μηχανικός και Μηχανικός Υπολογιστών Ε.Μ.Π.

Copyright© Κωνσταντίνος Φερτάκης, 2018. Εθνικό Μετσόβιο Πολυτεχνείο.
Με επιφύλαξη κάθε δικαιώματος. All rights reserved.

Απαγορεύεται η αντιγραφή, αποθήκευση και διανομή της παρούσας εργασίας, εξ'ολοκλήρου ή τμήματος αυτής, για εμπορικό σκοπό. Επιτρέπεται η ανατύπωση, αποθήκευση και διανομή για σκοπό μη κερδοσκοπικό, εκπαιδευτικής ή ερευνητικής φύσης, υπο την προϋπόθεση να αναφέρεται η πηγή προέλευσης και να διατηρείται το παρόν μήνυμα. Ερωτήματα που αφορούν τη χρήση της εργασίας για κερδοσκοπικό σκοπό πρέπει να απευθύνονται προς το συγγραφέα. Οι απόψεις και τα συμπεράσματα που περιέχονται σε αυτό το έγγραφο εκφράζουν το συγγραφέα και δεν πρέπει να ερμηνευτεί ότι αντιπροσωπεύουν τις επίσημες θέσεις του Εθνικού Μετσόβιου Πολυτεχνείου.

Περίληψη

Η χρήση επιταχυντών σε υπολογιστικές εγκαταστάσεις που χρησιμοποιούν ετερογένεια για την επίτευξη υψηλότερων επιδόσεων έχουν εδραιωθεί τα τελευταία χρόνια. Οι επιταχυντές βρίσκονται στις καρδιές των σύγχρονων κέντρων δεδομένων και υπολογιστών, υποστηρίζοντας τη λειτουργία της πλειοψηφίας των δέκα ταχύτερων υπερυπολογιστών στον κόσμο. Είναι ουσιαστικής σημασίας για τις κοινότητες υπολογιστών υψηλής απόδοσης και μηχανικής μάθησης, εφαρμόζοντας προσαρμοσμένη αρχιτεκτονική προκειμένου να παρέχουν αποτελεσματική κλιμακούμενη ισχύ επεξεργασίας που στοχεύει σε ένα ευρύ φάσμα επιστημονικών τομέων.

Σε αυτήν την εργασία, αναλαμβάνουμε την πρόκληση να σχεδιάσουμε και να υλοποιήσουμε ένα σύστημα το οποίο θα επιτρέπει την εξ αποστάσεως πρόσβαση στους πόρους ενός επιταχυντή. Παρουσιάζουμε το RACEX, ένα σύστημα που επιτρέπει την αποτελεσματική απομακρυσμένη εκτέλεση εφαρμογών σε επιταχυντή. Στη υλοποίηση της ιδέας μας, έχουμε στοχεύσει το συνεπεξεργαστή Intel Xeon Phi. Η προτεινόμενη λύση επιτρέπει την πλήρη ή μερική εκφόρτωση υπολογισμών και εφαρμογών σε έναν επιταχυντή Intel Xeon PHI προκειμένου αυτές να εκτελεστούν και να αξιοποιήσουν τη δύναμη των μαζικά παράλληλων επεξεργαστών του. Το RACEX εισέρχεται στη στοίβα λογισμικού του επιταχυντή στο επίπεδο στρώματος μεταφοράς που υλοποιείται απο το πρωτόκολλο SCIF της Intel, το οποίο προορίζεται για τη μεταφορά δεδομένων μέσω του PCIe στη συσκευή επιτάχυνσης. Το σύστημα μας υπεισέρχεται στις κλήσεις προς το πρωτόκολλου SCIF και προωθεί αυτές σε κάποιο απομακρυσμο διακομηστή προκειμένου να επιτρέψει την εξ αποστάσεως εκτέλεση. Το σύστημα μας χρησιμοποιεί BSD Sockets για δικτύωση και επικοινωνία μεταξύ των διαδικτυακά κατανεμημένων κόμβων. Τα αρχικά αποτελέσματα αξιολόγησης είναι ελπιδοφόρα, καθώς οι σχετικές μετρήσεις καταδεικνύουν 10% επιβάρυνση του RACEX σε σύγκριση με τη φυσική εκτέλεση όσον αφορά την καθυστέρηση για την ανταλλαγή μεγάλων μηνυμάτων μεταξύ του host και του επιταχυντή.

Λέξεις Κλειδιά: Επιταχυντές, Intel Xeon Phi, Απομακρυσμένη Εκτέλεση, Middleware Framework, Cluster Computing, Cloud Computing, Σύστημα Διαμοιρασμού Επιταχυντών, Hardware Abstraction, Transport Layer Abstractions

Ευχαριστίες

Η διπλωματική εργασία αυτή πραγματοποιηθήκε στο Εργαστήριο Υπολογιστικών Συστημάτων της Σχολής Ηλεκτρολόγων Μηχανικών και Μηχανικών Υπολογιστών του Εθνικού Μετσόβιου Πολυτεχνείου, υπο την επίβλεψη του Επίκουρου Καθηγητή Γεώργιου Γκούμα.

Καταρχήν θα ήθελα να ευχαριστήσω τον υποψήφιο διδάκτορα και μέλος του εργαστηρίου Στέφανο Γεράνγκελο για την συνεισφορά και εποπτεία του κατα τη εκπόνηση της διπλωματικής μου εργασίας, καθώς και για την ενθάρρυνσή του, την υπομονή του και το χρόνο που αφιέρωσε. Η συμβολή του ήταν καθοριστική τόσο στη σύλληψη του θέματος της εργασίας αλλά και για την ολοκλήρωση της.

Έπειτα θα ήθελα να ευχαριστήσω τους καθηγητές μου κ. Γέωργιο Γκούμα και κ. Νεκτάριο Κοζύρη για τη συμβολή τους στη διαμόρφωση μου ως μηχανικού απο τις διδασκαλίες τους. Θα ήθελα επίσης να εκφράσω τις ευχαριστίες μου σε όλα τα μέλη του εργαστηρίου για τις επικοδομητικές συζητήσεις οι οποίες συνεισφέραν στη διαμόρφωση αυτής της εργασίας.

Στην συνέχεια, επιβάλλεται να ευχαριστήσω τους φίλους και συμφοιτητές μου για την παρουσία τους και την πολύτιμη βοήθειά τους, τόσο σε προσωπικό όσο και σε επιστημονικό επίπεδο, καθ'όλη τη διάρκεια των προπτυχιακών μου σπουδών.

Τέλος, οι θερμότερες ευχαριστίες απευθύνονται στους γονείς μου, την αδερφή μου και την οικογένεια μου, για την αγάπη τους, την αμέριστη υποστήριξή τους και την εμπιστοσύνη τους σε κάθε μου επιλογή, από τα πρώτα χρόνια της ζωής μου μέχρι σήμερα, αδιάκοπα και ακούραστα.

Contents

List of Figures

Chapter 1

Εισαγωγή

1.1 Κίνητρο

Η ετερογένεια έχει εδραιωθεί εδώ και πολύ καιρό στο χώρο του High Performance Computing ως πρότυπο σχεδιασμού που εφαρμόζεται σε μεγάλης κλίμακας υπολογιστικά συστήματα και κέντρα δεδομένων. Με τη συγκέντρωση δεδομένων παγκοσμίως να αυξάνεται σε μέγεθος εκθετικά κάθε χρόνο, παράλληλα με την πολυπλοκότητα εφαρμογών που εξυπηρετούν επιστημονικούς και εμπορικούς χρήστες, ο αγώνας για όλο και μεγαλύτερη επεξεργαστική ισχύ είναι σε εξέλιξη. Στο πλαίσιο αυτού, η χρήση επιταχυντών στις σύγχρονες υπολογιστικές εγκαταστάσεις είναι ένα παράδειγμα των έντονων προσπαθειών τόσο του ακαδημαϊκού όσο και του βιομηχανικού κλάδου για την επίτευξη υψηλότερων υπολογιστικών επιδόσεων για το φόρτο εργασιών τους. Οι επιταχυντές έχουν έρθει στο προσκήνιο λόγω των μαζικών δυνατοτήτων παραλληλισμού που μπορούν να προσφέρουν, σε σύγκριση με μια παραδοσιακή CPU γενικής χρήσης και την προσαρμοσμένη αρχιτεκτονική που αξιοποιούν για να προσφέρουν την επιδιωκόμενη αύξηση της απόδοσης.

Επιπλέον, το Cloud Computing έχει εδραιωθεί για όλους τους κλάδους επιστημονικών και επιχειρηματικών δραστηριοτήτων, παρέχοντας στους τελικούς χρήστες ελαστική και κλιμακούμενη πρόσβαση σε υπολογιστικούς πόρους [1, 2, 3]. Παραδοσιακά, οι πάροχοι υπηρεσιών υπολογιστικού νέφους εξυπηρετούν μια πικοίλη ομάδα πελατών, παρέχοντάς τους πρόσβαση σε κατανεμημένους υπολογιστικούς και αποθηκευτικούς πόρους, οι οποίοι σε αντάλλαγμα χρησιμοποιούν την παρεχόμενη υποδομή για ευρύ φάσμα εφαρμογών και σκοπών. Ωστόσο, με τον πολλαπλασιασμό των εφαρμογών μηχανικής μάθησης και τεχνητής νοημοσύνης τόσο στον ακαδημαϊκό χώρο όσο και στη βιομηχανία, υπήρξε μια όλο και αυξανόμενη ζήτηση προς τους παρόχους υπολογιστικού νέφους για εξειδικευμένο υλικό που να μπορεί να ικανοποιήσει τις βαριές υπολογιστικές απαιτήσεις του σύγχρονου φόρτου εργασίας τους [1, ?, 4]. Αυτό έχει οδηγήσει σε μια αλλαγή πρακτικής όπου οι πάροχοι σήμερα έχουν ενσωματώσει επιταχυντές στις υπολογιστικές εγκαταστάσεις και τα κέντρα δεδομένων τους και προσφέρουν ελαστική πρόσβαση στη μαζική παράλληλη υπολογιστική ισχύ τους, γεννώντας επομένως τον κλάδο του Accelerated Cloud Computing [5, 6, 7].

Η κοινότητα υψηλής απόδοσης υπολογιστών έχει ήδη υιοθετήσει τις υπηρεσίες πληροφορικής που παρέχονται από τους παρόχους cloud ως βιώσιμη λύση για την εκτέλεση εφαρμογών HPC με υψηλές επιδόσεις που ανταποκρίνονται στη ζήτηση [8]. Αρχικά, οι υπηρεσίες cloud που προσφέρονταν απο τους παρόχους δεν ήταν κατάλληλες για παραδοσιακές εφαρμογές HPC, κυρίως λόγω της έλλειψης διασυνδέσεων υψηλής απόδοσης. Ωστόσο, πρόσφατα σημαντικοί προμηθευτές υπολογιστικού νέφους

έχουν συμπεριλάβει δυνατότητες δικτύωσης υψηλής απόδοσης στην προσφερόμενη υποδομή τους, καθιστώντας τους μια εναλλακτική λύση έναντι των παραδοσιακών HPC inhouse computing clusters, τα οποία ήταν δαπανηρά για τη συντήρηση και τη λειτουργία τους. Επιπλέον, στην περιοχή του IoT, οι συσκευές χαμηλής κατανάλωσης συγκεντρώνουν ένα τεράστιο φορτίο δεδομένων που πρέπει να υποβληθούν σε επεξεργασία για να παράσχουν την προβλεπόμενη ανάλυση και πληροφορίες. Ωστόσο, παρά τις τεχνολογικές εξελίξεις στο υλικό, οι συσκευές IoT συνδέονται εκ φύσεως με τον περιορισμό της επεξεργαστικής ισχύς και της κατανάλωσης ενέργειας. Έτσι, το Cloud Computing για το IoT προέκυψε παρέχοντας μια εναλλακτική λύση όπου οι συνδεδεμένες συσκευές στο δίκτυο μεταφορτώνουν την επεξεργασία των υπολογιστικών εργασιών τους στην υπολογιστική υποδομή και το εξειδικευμένο υλικό επιτάχυνσης που μπορεί να προσφέρει το Cloud [9, 10].

Υπό το πρίσμα των παραπάνω, η σημασία των επιταχυντών για την κοινότητα Cloud Computing είναι πρωταρχικής σημασίας και παρόλο που πολλές σύγχρονες υπολογιστικές εγκαταστάσεις μπορούν να έχουν ήδη αφομοιώσει την ετερογένεια στο σχεδιασμό τους και να χρησιμοποιήσουν τα οφέλη που προσφέρουν οι επιταχυντές, η χρήση τους δεν είναι ιδανική. Οι σύγχρονοι παροχείς υπηρεσιών Cloud που προσφέρουν υπηρεσίες επιτάχυνσης, παρέχουν στατική και ανελαστική πρόσβαση. Η ενσωμάτωση των επιταχυντών στην υποδομή του cloud ως μέρος των συλλογικών υπολογιστικών πόρων μπορεί να είναι προβληματική, καθώς οι αρχιτεκτονικές τους είναι σχεδιασμένες να χρησιμοποιούνται μονολιθικά [11]. Ως εκ τούτου, οι πάροχοι προσφέρουν επιταχυντές ως σταθερό πόρο σε ένα μοντέλο Υποδομής-ως-Υπηρεσίας (Infrastructure-as-a-Service). Σε αυτό το πλαίσιο, οι πάροχοι αναγκάζονται να αποκτήσουν πρόσθετο υλικό για να ικανοποιήσουν τις αυξανόμενες απαιτήσεις, χωρίς να έχουν το αρχικό τους υλικό στη μέγιστη δυνατή λειτουργία και έτσι να οδηγούντε σε υπερπρομήθευση συσκευών. Η χαμηλή συνολική αξιοποίηση του υποκείμενου υλικού σημαίνει υψηλότερο κόστος λειτουργίας και συντήρησης, υψηλότερη κατανάλωση ενέργειας για τις υπολογιστικές εγκαταστάσεις και μεγαλύτερες απαιτήσεις χώρου, προκειμένου να φιλοξενηθούν οι πρόσθετοι εξυπηρετητές και το υλικό [12, 13].

Προκειμένου να αντιμετωπιστούν οι ανησυχίες που προκύπτουν σχετικά με τη χρήση των επιταχυντών στα σύγχρονα κέντρα δεδομένων, επιχειρηματολογούμε ότι μπορεί να χρησιμοποιηθεί ένα σύστημα διαμοιρασμού επιταχυντών για συμπλέγματα υπολογιστών σε περιβάλλοντα σύννεφων. Παρέχοντας επιταχυντές διαθέσιμους όχι μόνο στη συσκευή με την οποία είναι συνδεδεμένοι, αλλά και στους υπόλοιπους κόμβους ενός συμπλέγματος σε ένα κέντρο δεδομένων, υψηλότερα επίπεδα χρήσης μπορούν να επιτευχθούν ανά επιταχυντή υλικού. Ως εκ τούτου, μπορούμε να μειώσουμε τη συνολική ποσότητα εξαρτημάτων επιταχυντών που απαιτούνται από ένα κέντρο δεδομένων για τις ίδιες δυνατότητες απόδοσης και στη συνέχεια να περιορίσουμε το συνολικό χώρο που απαιτείται για το υλικό να καταλάβει. Αυτό όχι μόνο αυξάνει τη χρήση του υποκείμενου υλικού αλλά και μετριάζει το λειτουργικό κόστος ολόκληρης της εγκατάστασης επεξεργασίας δεδομένων. Υπάρχει Υπήρξε μια πληθώρα από ενδιαφέρουσες ερευνητικές δραστηριότητες που πραγματοποιήθηκαν προσφάτως περιγράφοντας τα οφέλη της διάθεσης επιταχυντών μέσω ενός συστήματος διαμοιρασμού και των επιδόσεων που μπορούν να επιτευχθούν [14, 15, 16, 17].

Λαμβάνοντας υπόψη τις πρόσφατες εξελίξεις στον τομέα της κοινής χρήσης επιταχυντών, προχωρούμε και εξερευνούμε την ιδέα να προσθέσουμε ένα επιπλέον επίπεδο αφαίρεσης υλικού στη στοίβα των συμπλεγμάτων υπολογιστών, μία που θα επιτρέψει την απομακρυσμένη διαχείριση των επιταχυντών που υπάρχουν στο σύμπλεγμα και τον διαμοιρασμό των πόρων τους μεταξύ όλων των κόμβων του συμπλέγματος. Σε σύγκριση με άλλα συστήματα απομακρυσμένης εκτέλεσης επιταχυντών που έχουν εξερευνηθεί [17, 18, 19], μπορούμε να συμπεράνουμε ότι προσθέτοντας ενα επιπλέον επίπεδο αφαίρεσης στην επικοινωνία (Transport Layer) μεταξύ του κεντρικού υπολογιστή και του επιταχυντή,

μπορούμε να αποσυνδέσουμε τον επιταχυντή από τη αποκλιστική χρήση από το μηχάνημα στο οποίο είναι συνδεμένος. Οι τρέχουσες πρακτικές στο cloud computing υπαγορεύουν ότι για έναν χρήστη υπολογιστικού νέφους να αποκτήσει πρόσβαση σε πόρους επιταχυντή, θα πρέπει να δημιουργηθεί μια εικονική μηχανή σε μια μηχανή που συνδέεται απευθείας με τον επιταχυντή και στη συνέχεια να δώσει στην εν λόγω εικονική μηχανή άμεση αποκλειστική πρόσβαση στο επιταχυντή. Μέσω του επιπλέον επιπέδου αφαίρεσης στο Transport Layer, που υπάρχει στη στοίβα λογισμικού του επιταχυντή και είναι υπεύθυνο για τη διμερή επικοινωνία μεταξύ του κεντρικού υπολογιστή και του επιταχυντή, μπορούμε να παρέχουμε κοινή πρόσβαση στους πόρους επιταχυντή σε πολλαπλές εικονικές μηχανές που εκτελούνται σε έναν μοναδικό κόμβο ή ακόμα και σε απομακρυσμένα VM σε όλο το σύμπλεγμα υπολογιστών. Μπορούμε ουσιαστικά να εικονικοποιήσουμε τους επιταχυντές που υπάρχουν σε ένα κέντρο δεδομένων, που παραδοσιακά χρησιμοποιούνται αποκλειστικά από τους διακομιστές στους οποίους ήταν συνδεδεμένοι και να μοιράσουμε τους πόρους τους συλλογικά στο σύμπλεγμα ως μέρος μιας ομάδας επιταχυνόμενου υλικού που ένας κεντρικός προγραμματιστής μπορεί να ενεργοποιήσει και να αναθέσει εργασίες.

1.2 Συνεισφορά

Σε αυτή τη διπλωματική εργασία διερευνάμε τη σκοπιμότητα, τις επιδόσεις και τα οφέλη του παραπάνω επιχειρήματος υλοποιόντας ένα πρωτότυπο του συστήματος που προτείνουμε το οποίο το βασίζουμε στον επιταχυντή Intel Xeon Phi. Παρουσιάζουμε το RACEX (Remote ACcelerator EXecution) ως ένα σύστημα που επιτρέπει την ταυτόχρονη πρόσβαση σε πόρους του επιταχυντή απο πολλούς απομακρυσμένους πελάτες. Το RACEX είναι ένα middleware framework που ακολουθεί Server-Client αρχιτεκτονικό σχεδιασμό. Στο σχεδιασμό μας, το RACEX παρεμβάλλεται στη στοίβα λογισμικού του Intel Xeon Phi ακριβώς πάνω από το στρώμα μεταφοράς, υποκλέπτει και προωθεί τις κλήσεις SCIF API (Symmetric Communication Interface) της Intel. Το SCIF είναι ο μηχανισμός που υλοποιεί το στρώμα μεταφοράς στη στοίβα λογισμικού του επιταχυντή, που λειτουργεί πάνω απο το δίαυλο PCIe, επιτρέποντας την επικοινωνία μεταξύ του κεντρικού επεξεργαστή και του συνεπεξεργαστή επιταχυντή. Το RACEX εκθέτει την ίδια διεπαφή προγραμματισμού εφαρμογών (API) με αυτή του SCIF και είναι συμβατή με τις precompiled εφαρμογές. Υποστηρίζουμε επίσης παραδοσιακά συστήματα παράλληλης εκτέλεσης και εργαλεία που καθιστούν το σύστημα μας διαφανές ως προς τις υψηλού επιπέδου εφαρμογές. Στην τρέχουσα εργασία μας, υποστηρίζουμε τους τρόπους εκτέλεσης τόσο του *native* όσο και του *offload* και με μικρές προσαρμογές μπορούμε να επεκτείνουμε την εργασία μας για να υποστηρίξουμε το μοντέλο εκτέλεσης *symmetric*.

Μια επισκόπηση ενός δυνητικού ετερογενή υπολογιστικού κέντρου που υποστηρίζεται από το RACEX απεικονίζεται στην Εικόνα 3.1. Σε αυτήν την απεικόνιση, οι λεπτές απομακρυσμένες συσκευές αλλά και οι εσωτερικοί υπολογιστικοί κόμβοι και οι εικονικές μηχανές εκφορτίζουν τις υπολογιστικές εργασίες τους στους διαθέσιμους επιταχυντές σε ένα rack server που είναι εξοπλισμένο με πολλαπλούς επιταχυντές. Με αυτό το έργο ανοίγουμε το δρόμο για ένα μελλοντικό υπολογιστικό κέντρο, όπου ένας έξυπνος χρονοπρογραμματιστής που τρέχει το RACEX θα λάβει εισερχόμενη αίτηση για πρόσβαση σε πόρους επιταχυντή και θα είναι σε θέση να κατευθύνει αποτελεσματικά την πρόσβαση σε διαθέσιμους επιταχυντές στο εσωτερικό του υπολογιστικού κέντρου.

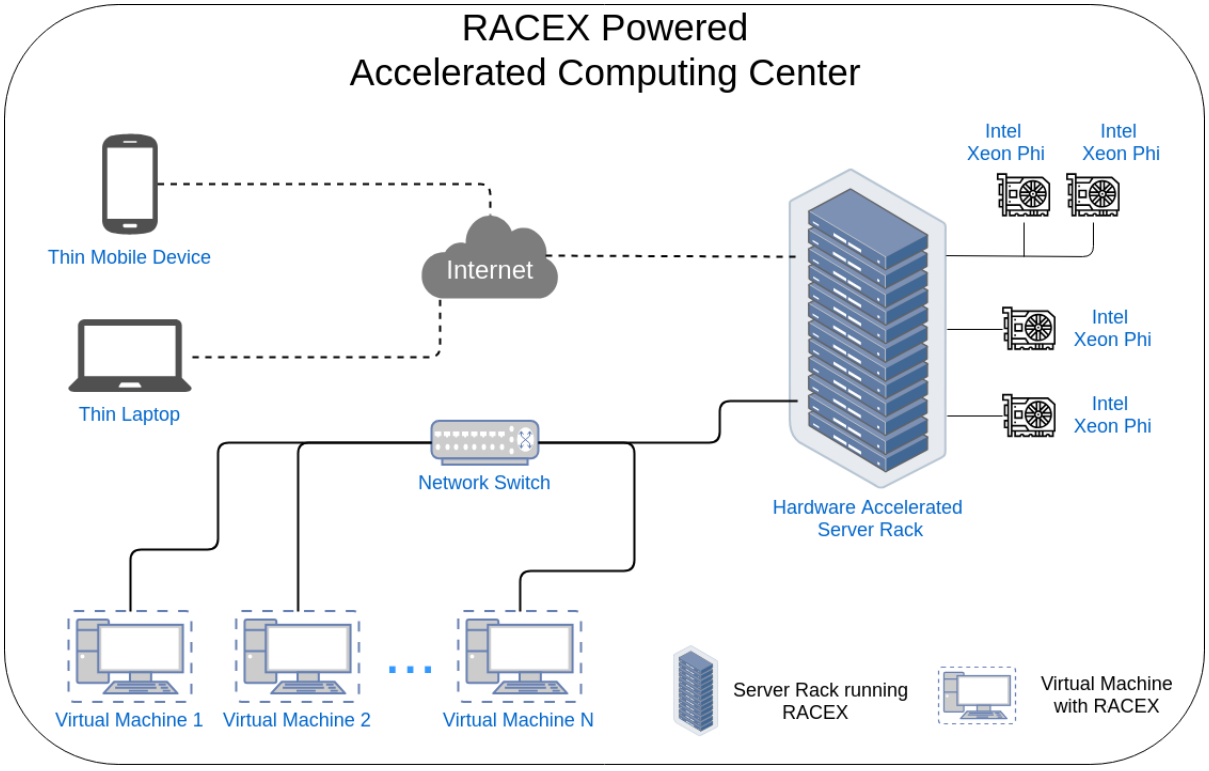


Figure 1.1: Accelerated Computing Center with RACEX Framework

Chapter 2

Θεωρητικό Υπόβαθρο

2.1 Ιστορική Αναδρομή

2.1.1 Η διαδρομή προς τα Πολυπύρηνα Συστήματα

Κατά τη διάρκεια του περασμένου αιώνα ο κόσμος αποτέλεσε μάρτυρας σε μια πρωτοφανή τεχνολογική εξέλιξη, την αποκαλούμενη Ψηφιακή Επανάσταση [20], η οποία σηματοδότησε τη μετάβαση από τη μηχανική και την αναλογική ηλεκτρονική τεχνολογία στα ψηφιακά ηλεκτρονικά. Αυτή η περίοδος θεωρείται η γενέτειρα των υπολογιστών, αλλά και η αυγή μιας νέας εποχής, η αρχή της εποχής της πληροφορίας.

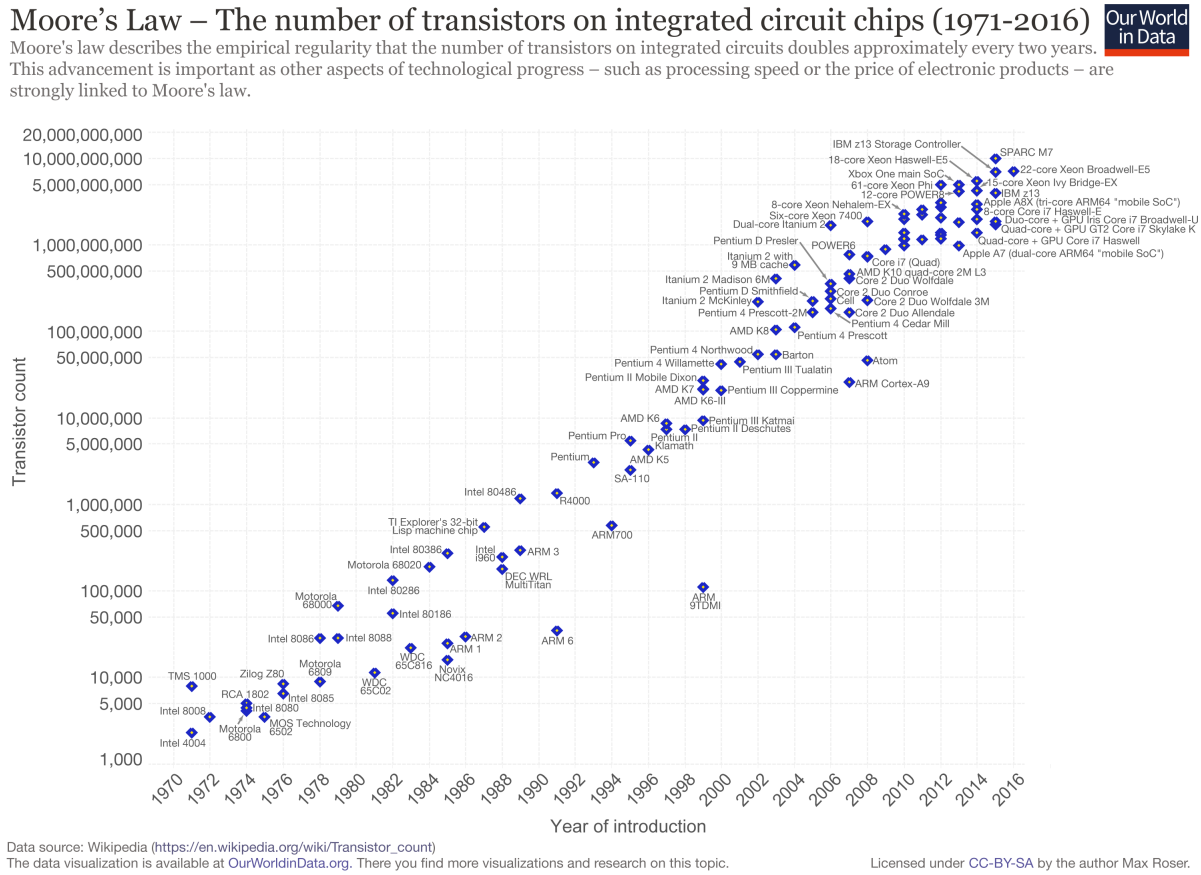
Αρχικά, οι πρώτοι ηλεκτρονικοί υπολογιστές γενικής χρήσης, ήταν τεράστιες μηχανές που καταλαμβάναν ολόκληρα δωμάτια, έπρεπε να επανασυνδέονται και να αναδιαμορφώνονται για να εκτελούν διαφορετικές λειτουργίες, ενώ παράλληλα είχαν σημαντικό λειτουργικό κόστος, καταναλώνοντας τεράστια φορτία ισχύος. Έτσι, η χρήση τους περιοριζόταν κυρίως σε κυβερνητικούς φορείς και άλλους οργανισμούς. Η εφεύρεση του πρώτου τρανζίστορ το 1947 στα Bell Labs θεωρείται από πολούς από τις μεγαλύτερες εφευρέσεις του 20ου αιώνα. Η εφεύρεση του τρανζίστορ, που αποτελεί το βασικό συστατικό όλων των σύγχρονων ηλεκτρονικών συσκευών, έδειξε το δρόμο προς τους πρώτους οικιακούς υπολογιστές, επιτρέποντας για πρώτη φορά σε μη τεχνικούς χρήστες τη λειτουργία ενός υπολογιστή και την αξιοποίηση της ισχύος επεξεργασίας σε προσιτό κόστος.

Από εκείνους τους μικροεπεξεργαστές που τροφοδοτούσαν τους πρώτους εμπορικά διαθέσιμους υπολογιστές, ο τομέας των μονάδων επεξεργασίας βίωσε εξελίξεις σε εκθετικό βαθμό. Η δυνατότητα κατασκευής εξαιρετικά μικρών τρανζίστορ σε ένα ολοκληρωμένο κύκλωμα αύξησε την πολυπλοκότητα και τον αριθμό των τρανζίστορ σε μια ενιαία μονάδα επεξεργασίας. Αυτό οδήγησε σε μια κούρσα μεταξύ του ακαδημαϊκού κόσμου και της βιομηχανίας για τη δημιουργία CPUs με όλο και μεγαλύτερη δύναμη επεξεργασίας, σε μικρότερο μέγεθος και κόστος. Ήταν ο Gordon E. Moore που το 1965, υπό το πρίσμα των πρόσφατων εξελίξεων στους τομείς των τρανζίστορ και των ολοκληρωμένων κυκλωμάτων, δήλωσε αυτό που αργότερα θα γινόταν γνωστό ως ο νόμος του Moore (Moore's Law) [21]

> The complexity for minimum component costs has increased at a rate of roughly a factor of two per year. Certainly over the short term this rate can be expected to continue, if not to increase. Over the longer term, the rate of increase is a bit more uncertain, although there is no reason to believe it will not remain nearly constant

> for at least 10 years. That means by 1975, the number of components per integrated circuit for minimum cost will be 65,000. I believe that such a large circuit can be built on a single wafer.

Η παρατήρηση του Moore έγινε γνωστή ως πρόβλεψη ότι ο αριθμός των τρανζίστορ σε ένα πυκνό ολοκληρωμένο κύκλωμα διπλασιάζεται περίπου κάθε δύο χρόνια. Ο ίδιος εμπειρικός νόμος έγινε ευρέως αποδεκτός ως στόχος για τη βιομηχανία και αναφέρθηκε από κατασκευαστές ημιαγωγών καθώς προσπαθούσαν να αυξήσουν την επεξεργαστική δύναμη.



Παράλληλα με την εμπειρική παρατήρηση του Moore, ένας άλλος νόμος σχετικά με την κλιμάκωση των επιδόσεων των μονάδων επεξεργασίας προέκυψε κατά τη διάρκεια του αγώνα για Κεντρικές Μονάδες Επεξεργασίας (ΚΜΕ) υψηλότερης απόδοσης, ο νόμος "MOSFET Scaling" ή, όπως είναι ευρύτερα γνωστός, "Dennard Scaling" που πήρε το όνομα του απο τον συν-συγγραφέα του αρχικού εγγράφου του 1974, ο οποίος ανέφερε ότι καθώς τα τρανζίστορ γίνονται μικρότερα, η πυκνότητα ισχύος τους παραμένει σταθερή. Η ανάπτυξη των μονάδων επεξεργασίας κατά τη διάρκεια του αγώνα για όλο και περισσότερη δύναμη επιδόσεων, φθάνοντας σε υψηλότερες και υψηλότερες συχνότητες ρολογιού CPU, φάνηκε να τηρεί τον νόμο κλιμάκωσης, αλλά γύρω στο 2005 η κλιμάκωση του Dennard φαίνεται ότι έχει καταρρεύσει. Παρόλο που ο συνολικός αριθμός τρανζίστορ σε ολοκληρωμένα κυκλώματα αυξανόταν, οι βελτιώσεις που προέκυπταν στην απόδοσή τους ήταν πιο σταδιακές από τις επιταχύνσεις που προέκυπταν από σημαντικές αυξήσεις συχνότητας. Ο κύριος λόγος που αναφέρθηκε για την κατάρευση της κλιμάκωσης είναι ότι σε μικρά μεγέθη, η διαρροή ρεύματος δημιουργεί

μεγάλες προκλήσεις και επίσης προκαλεί τη θέρμανση του τσιπ, γεγονός που δημιουργεί απειλή θερμικής διαφυγής και ως εκ τούτου αυξάνει περαιτέρω το ενεργειακό κόστος. Αυτά δημιούργησαν ένα "Ενεργειακό Τοίχος" που έχει θέσει περιορισμούς στη πρακτική συχνότητα single core επεξεργαστών σε περίπου 4 GHz από το 2006.

Καθώς δεν μπορούσαμε να βασιστούμε στα τρανζίστορ για την αύξηση της σειριακής απόδοσης σε έναν επεξεργαστή αρχιτεκτονικής ενιαίου πυρήνα, έπρεπε να βρεθεί μια εναλλακτική λύση προκειμένου να ικανοποιηθεί η συνεχώς αυξανόμενη ζήτηση υψηλότερης απόδοσης επεξεργασίας. Η λύση ήρθε με τη μετάβαση σε πολυπύρηνη αρχιτεκτονική για μονάδες επεξεργασίας. Το κίνητρο για επεξεργαστές πολλών πυρήνων προήλθε από τα σημαντικά μειωμένα κέρδη στις επιδόσεις των επεξεργαστών από την αύξηση της συχνότητας λειτουργίας, η οποία μπορεί να εντοπιστεί κυρίως σε τρεις κύριους παράγοντες:

1. Το τοίχος της μνήμης: το αυξανόμενο χάσμα ανάμεσα στις ταχύτητες του επεξεργαστή και της μνήμης. Αυτό, στην πραγματικότητα, ωθεί τα μεγέθη της κρυφής μνήμης να είναι μεγαλύτερα ώστε να αποκρύψουν την καθυστέρηση της μνήμης. Αυτό βοηθά μόνο στο βαθμό που το εύρος ζώνης μνήμης δεν είναι το εμπόδιο στην απόδοση.

2. Το τοίχος παραλληλισμού επιπέδου εντολής: η αυξανόμενη δυσκολία εξεύρεσης επαρκούς παραλληλισμού σε μια ενιαία ροή εντολων για να διατηρηθεί ένας υψηλής απόδοσης επεξεργαστής ενός πυρήνα απασχολημένος.

3. Το τοίχος της ενέργειας: η τάση της κατανάλωσης εκθετικά αυξανόμενης ισχύος (και συνεπώς επίσης η δημιουργία εκθετικά αυξανόμενης θερμότητας) με κάθε παράγοντα αύξηση της συχνότητας λειτουργίας. Αυτή η αύξηση μπορεί να μετριαστεί με τη "συρρίκνωση" του επεξεργαστή χρησιμοποιώντας μικρότερα ίχνη για την ίδια λογική. Ο τοίχος ισχύος δημιουργεί προβλήματα κατασκευής, σχεδιασμού και εγκατάστασης του συστήματος, τα οποία δεν έχουν δικαιολογηθεί λόγω των μειωμένων κερδών απόδοσης λόγω του τοίχου μνήμης και του τοίχους ILP.

Η αποτυχία της κλιμάκωσης του Dennard οδήγησε τόσο τον ακαδημαϊκό όσο και τον βιομηχανικό κλάδο να επικεντρώσουν τις προσπάθειές τους στην αύξηση του αριθμού των πυρήνων σε τσιπ ως μια βιώσιμη και εναλλακτική λύση για την αύξηση της απόδοσης. Τα συστήματα Multicore επέτρεψαν τη βελτίωση της απόδοσης μέσω παράλληλης επεξεργασίας, επιλύοντας ένα πρόβλημα διαιρώντας το σε πολλαπλές εργασίες που μπορούν να εκτελούνται ταυτόχρονα σε πολλούς επεξεργαστές. Η αύξηση της επεξεργαστικής ισχύς μέσω συστημάτων πολυπύρηνων συστημάτων είναι πιο δύσκολη στην επίτευξη κέρδους απόδοσης, καθώς οι προγραμματιστές πρέπει να αναπτύξουν παράλληλο λογισμικό που εκμεταλλεύεται την πολυπύρηνη αρχιτεκτονική.

2.1.2 Ο νόμος του Amdahl και η πρόκληση του παράλληλου προγραμματισμού

Καθώς μπήκαμε στην πολυπύρηνη εποχή, το υπολογιστικό τοπίο απομακρύνθηκε από το μονοπύρηνο σχεδιασμό συστημάτων και επικεντρώθηκε στους επεξεργαστές πολλαπλών πυρήνων με την αντίληψη ότι η παραλληλοποίηση που θα μπορούσαν να επιτύχουν πολλοί πυρήνες σε ένα ενιαίο τσιπ θα αρκούσε για τη διατήρηση της κλιμάκωσης απόδοσης που απαιτούσε η αγορά και είχε προηγουμένως βιώσει κατά τη χρυσή εποχή του νόμου του Moore. Ωστόσο, η διαθεσιμότητα πολλαπλών πυρήνων στο ίδιο τσιπ δεν εγγυάται ότι η απόδοση των εκτελεσμένων εφαρμογών θα κλιμακωθεί αβίαστα. Σε

πολλές περιπτώσεις, ο προγραμματιστής πρέπει να σχεδιάσει ρητά το λογισμικό για να επωφεληθεί από τους διαθέσιμους πυρήνες. Το λογισμικό που πρέπει να γραφτεί για να επωφεληθεί από τους πολλαπλούς πυρήνες προκειμένου να επιταχυνθεί μια εφαρμογή σε σύγκριση με τις επιδόσεις της σε ένα σύστημα μονοπύρηνου επεξεργαστή είναι εγγενώς δύσκολο.

Σε ένα παράλληλο πρόγραμμα, το υπολογιστικό φορτίο πρέπει να σπάσει σε ίσα μεγέθη και έπειτα κάθε κομμάτι να αντιστοιχιστεί σε έναν πυρήνα. Σε περίπτωση που οι φόρτοι εργασίας των επι μέρους πυρήνων δεν έχουν το ίδιο μέγεθος, κάποιοι πυρήνες θα παραμείνουν αδρανείς ενώ περιμένουν τους άλλους με τα μεγαλύτερα κομμάτια να τελειώσουν. Επιπλέον, διαφορετικοί επεξεργαστές πρέπει να επικοινωνούν μεταξύ τους κατά την εκτέλεση, προκειμένου να επεξεργαστούν με επιτυχία το φόρτο εργασίας τους. Εάν ο συνολικός χρόνος που αφιερώνεται για την επικοινωνία μεταξύ των πυρήνων είναι αρκετά μεγάλος, μπορεί να μετριάσει τη συνολική επιτάχυνση που επιτυγχάνεται με την παραλληλοποίηση του προγράμματος. Εν ολίγοις, ο προγραμματισμός, η εξισορρόπηση φορτίου, ο χρόνος συγχρονισμού και τα γενικά έξοδα για την επικοινωνία μεταξύ των πυρήνων αποτελούν σημαντικές προκλήσεις και γίνονται ακόμα μεγαλύτερες καθώς αυξάνεται ο αριθμός των επεξεργαστών.

Όταν κάποιος σκέφτεται τη συνολική επιτάχυνση που μπορεί να επιτύχει ένα πρόγραμμα μέσω παραλληλισμού, πρέπει κανείς να εξετάσει το νόμο Amdahl [22]. Ο νόμος του Amdahl είναι ένας τύπος που δίνει τη θεωρητική επιτάχυνση που μπορεί να επιτευχθεί, όσον αφορά την εκτέλεση ενός προγράμματος κατά παράλληλο τρόπο σε σύγκριση με τη σειριακή εκτέλεση. Στα πρώτα χρόνια των υπολογισμών υψηλής απόδοσης, ο Gene M. Amdahl [22] ανέφερε αρχικά ορισμένους εγγενείς περιορισμούς στη διαδικασία του παράλληλου προγραμματισμού και της απόδοσης. Πρώτον, υπάρχει ένα κλάσμα υπολογιστικού φόρτου σε κάθε εφαρμογή, που σχετίζεται με τη διαχείριση δεδομένων, το οποίο δεν μπορεί να εκτελεστεί παράλληλα με άλλους υπολογισμούς και ενεργεί ως σταθερή επιβάρυνση στο χρόνο εκτέλεσης. Δεύτερον, όταν το σύνολο δεδομένων του προβλήματος κατανέμεται μεταξύ των επεξεργαστών, ενδέχεται να προκύψουν προβλήματα ανωμαλιών, όπως ανομοιογενείς εσωτερικοί χώροι, ακανόνιστα όρια, προβλήματα ασυνέπειας μεταξύ μεταβλητών και υπολογισμοί ασύμμετρης σύγκλισης. Για να μοντελοποιήσει τον πρώτο περιορισμό και να καθορίσει κατευθυντήριες γραμμές για την αντιμετώπιση των προβλημάτων παρατυπίας, ο Amdahl εισήγαγε τον περίφημο νόμο του, ο οποίος καταγράφει την επίδρασή τους στην ταχύτητα που μπορεί να επιτευχθεί.

Προτού καθορίσουμε περαιτέρω τον νόμο του Amdahl, πρέπει πρώτα να καθορίσουμε την επιτάχυνση όσον αφορά τις επιδόσεις που επιτυχάνοντε κατά την εκτέλεση του προγράμματος.

$$speedup(n\ processors) = \frac{execution\_time(1\ processor)}{execution\_time(n\ processors)}$$

Ο νόμος του Amdahl αναφέρεται στην αύξηση της απόδοσης που μπορεί να επιτευχθεί με τη βελτίωση ενός μέρους του συνολικού προγράμματος, ας είναι το μέρος p του συνολικού προγράμματος. Αν αυτή η βελτίωση κάνει τα μέρη του προγράμματος πιο γρήγορα, δηλ. έχει μια επιτάχυνση ίση με s, τότε η συνολική επιτάχυνση του προγράμματος δίνεται από την εξίσωση:

$$total\_speedup = \frac{1}{(1-p) + \frac{p}{s}}$$

Λαμβάνοντας υπόψη τον τρόπο με τον οποίο ο αριθμός των ανεξάρτητων επεξεργαστών που συμβάλλουν στην παραλληλισμό ενός προγράμματος επηρεάζει τον ορισμό της επιτεύξιμης θεωρητικής επιτάχυνσης, αρκεί το εξής: Εάν το p είναι το ποσοστό του συνολικού προγράμματος που μπορεί να παραλληλισθεί και $1-p$ είναι το τμήμα του προγράμματος που δεν μπορεί να παραλληλιστεί (δηλαδή παραμένει σειριακό), τότε η μέγιστη συνολική επιτάχυνση που μπορεί να επιτευχθεί με τη χρήση N επεξεργαστών δίνεται από την ακόλουθη εξίσωση:

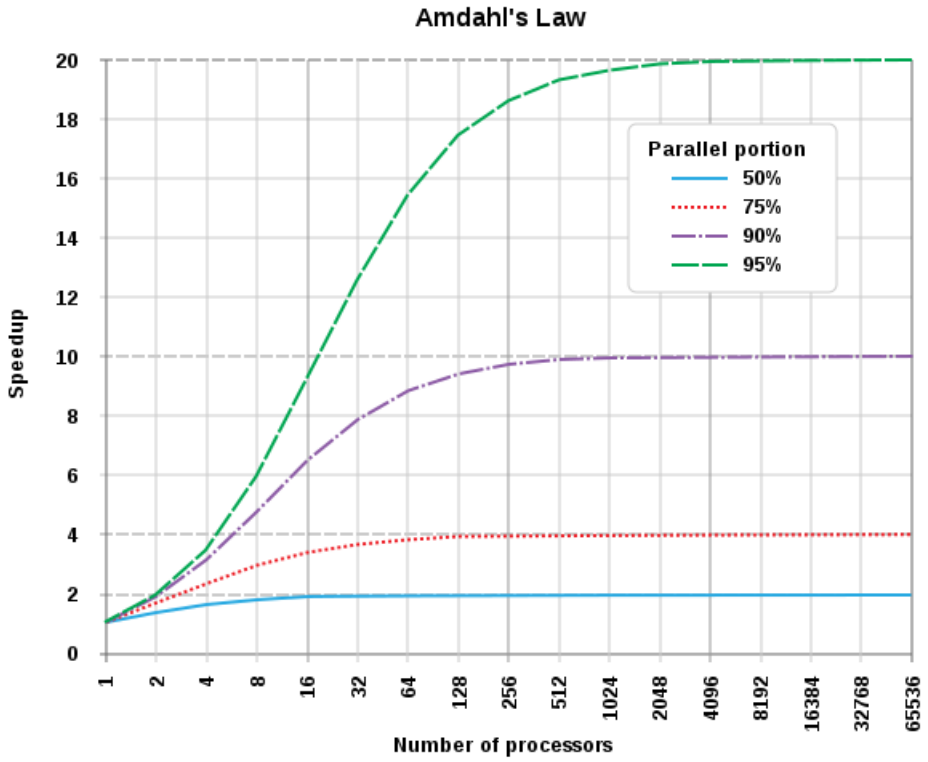


Figure 2.1: Total speedup of a parallel program as parallel fraction and number of processors increase

$$total\_speedup = \frac{1}{(1-p) + \frac{p}{N}}$$

Καθώς το N πηγαίνει στο άπειρο, η μέγιστη ταχύτητα μεταβαίνει σε $1/(1-p)$. Πρακτικά, η αναλογία μεταξύ αποδοτικότητας και κόστους αυξάνεται δραματικά ακόμα και αν το $1-p$ είναι σχετικά μικρό. Στο σχήμα 2.1 μπορούμε να δούμε πώς επηρεάζεται η συνολική επιτάχυνση ενώ η αναλογία p και ο αριθμός των επεξεργαστών N αλλάζει. Πρώτα απ 'όλα, η συνολική επιτάχυνση ενός προγράμματος που χρησιμοποιεί πολλούς πυρήνες σε ένα παράλληλο τμήμα περιορίζεται από το σειριακό τμήμα του προγράμματος. Επομένως, η χρήση ακόμη περισσότερων πυρήνων και η βελτίωση των αρχιτεκτονικών παράλληλων υπολογιστών δεν είναι πανάκεια. Αντ 'αυτού, μια εφαρμογή πρέπει να επανασχεδιαστεί έτσι ώστε ένα μεγαλύτερο μέρος της να είναι παραλληλοποιήσιμο.

2.1.3 Πολυπύρηνες Αρχιτεκτονικές

Σύμφωνα με την *κατηγοριοποίηση* του Michael J. Flynn για τις αρχιτεκτονικές υπολογιστών [23], υπάρχουν τέσσερις ταξινομήσεις για συστήματα υπολογιστών που βασίζονται στον αριθμό των ταυτόχρονων ροών δεδομένων και εντολών που είναι διαθέσιμα στην αρχιτεκτονική. Παρακάτω περιγράφονται αυτές οι τέσσερις κατηγορίες:

1. **Single Instruction Single Data (SISD):** Ένας σειριακός υπολογιστής που δεν εκμεταλλεύεται παραλληλισμό στις ροές δεδομένων και εντολών.

2. **Single Instruction Multiple Data (SIMD):** Αυτό αντιπροσωπεύει την οργάνωση ενός μόνο υπολογιστή που περιέχει μια μονάδα ελέγχου, μια μονάδα επεξεργαστή και μια μονάδα μνήμης. Οι εντολές εκτελούνται σειριακά. Μπορεί να επιτευχθεί μέσω αγωγών ή πολλαπλών λειτουργικών μονάδων.

3. **Multiple Instruction Single Data (MISD):** Πολλές εντολές λειτουργούν σε μία ροή δεδομένων. Αυτή είναι μια ασυνήθιστη αρχιτεκτονική που χρησιμοποιείται γενικά για ανοχή σφάλματος. Τα ετερογενή συστήματα λειτουργούν στην ίδια ροή δεδομένων και πρέπει να συμφωνήσουν στο αποτέλεσμα.

4. **Multiple Instruction Multiple Data (MIMD):** Πολλαπλοί αυτόνομοι επεξεργαστές εκτελούν ταυτόχρονα διαφορετικές εντολές σε διαφορετικά δεδομένα. Οι αρχιτεκτονικές MIMD περιλαμβάνουν υπερ-κλιμακώσιμους επεξεργαστές πολλαπλών πυρήνων και κατανεμημένα συστήματα, χρησιμοποιώντας έναν κοινόχρηστο χώρο μνήμης ή έναν κατανεμημένο χώρο μνήμης.

Οι υπολογιστές πολλαπλών πυρήνων βασίζονται στην αρχιτεκτονική MIMD, η οποία μπορεί να ταξινομηθεί περαιτέρω ανάλογα με την οργάνωση μνήμης σε αρχιτεκτονικές κοινής μνήμης, αρχιτεκτονικές κατανεμημένης μνήμης και υβριδικές αρχιτεκτονικές, όπου χρησιμοποιούνται πτυχές και των δύο προηγούμενων αρχιτεκτονικών. Τα τρία αντίστοιχα αρχιτεκτονικά σχέδια εξηγούνται παρακάτω.

Αρχιτεκτονική Κοινής Μνήμης

Σε μια αρχιτεκτονική κοινόχρηστης μνήμης, κάθε επεξεργαστής διαθέτει μια ιδιωτική ιεραρχία μνήμης cache και συλλογικά, όλοι οι επεξεργαστές του συστήματος μοιράζονται έναν ενιαίο χώρο φυσικής διεύθυνσης, μια παγκόσμια μνήμη. Όλοι οι επεξεργαστές και η παγκόσμια μνήμη διασυνδέονται μέσω ενός ενιαίου κοινόχρηστου διαύλου συστήματος. Οι επεξεργαστές επικοινωνούν μέσω κοινών μεταβλητών στη μνήμη, με όλους τους επεξεργαστές να έχουν πρόσβαση σε οποιαδήποτε μνήμη μέσω loads and stores. Όταν όλοι οι επεξεργαστές βρίσκονται σε ίση απόσταση από την παγκόσμια μνήμη, η αρχιτεκτονική μνήμης ορίζεται επίσης ως συμμετρική πολυεπεξεργαστική (SMP) και μπορεί να προβληθεί στο Σχήμα 2.2.

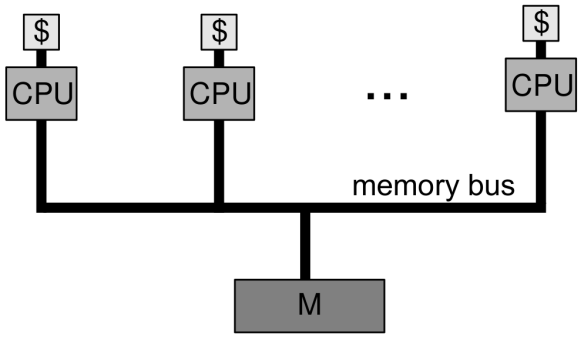


Figure 2.2: Classic organization of a SMP

Αυτό το μοντέλο επιτρέπει γρήγορη και ομοιόμορφη ανταλλαγή δεδομένων μεταξύ εργασιών λόγω της εγγύτητας της μνήμης με τις CPU. Ο συνολικός χώρος διευθύνσεων προσφέρει επίσης

ευκολία προγραμματισμού. Η δυνατότητα πρόσβασης σε όλα τα κοινά δεδομένα αποτελεσματικά από οποιονδήποτε επεξεργαστή που χρησιμοποιεί συνηθισμένα loads and stores, μαζί με την αυτόματη μετακίνηση και αναπαραγωγή των κοινών δεδομένων στις τοπικές κρυφές μνήμες, τα καθιστά ιδανικά για τον παράλληλο προγραμματισμό. Αυτές οι λειτουργίες είναι επίσης πολύ χρήσιμες για το λειτουργικό σύστημα, των οποίων οι διαφορετικές διαδικασίες μοιράζονται δομές δεδομένων και μπορούν εύκολα να τρέξουν σε διαφορετικούς επεξεργαστές. Ωστόσο, αυτή η προσέγγιση υποφέρει από έλλειψη επεκτασιμότητας. Η προσθήκη περισσότερων CPU μπορεί να αυξήσει γεωμετρικά την κυκλοφορία στον κοινό διαύλου συστήματος καθώς και την κυκλοφορία που σχετίζεται με τη συνοχή της κρυφής μνήμης. Με βάση τους χρόνους πρόσβασης στη μνήμη, οι αρχιτεκτονικές κοινής μνήμης μπορούν να ταξινομηθούν σε δύο κατηγορίες

1. **Ομοιόμορφης Πρόσβασης στη Μνήμη (UMA):** Η καθυστέρηση σε μια λέξη στη μνήμη δεν εξαρτάται από τον επεξεργαστή που το ζητάει. Όλοι οι επεξεργαστές μοιράζονται ομοιόμορφα τη φυσική μνήμη και ο χρόνος πρόσβασης σε μια θέση μνήμης είναι ανεξάρτητος από τον επεξεργαστή που κάνει την αίτηση ή ποιο τσιπ μνήμης περιέχει τα μεταφερόμενα δεδομένα.

2. **Μη-Ομοιόμορφης Πρόσβασης στη Μνήμη (NUMA):** Η καθυστέρηση πρόσβασης μνήμης για κάθε επεξεργαστή εξαρτάται από τη θέση της μνήμης. Κάθε επεξεργαστής έχει τη δική του τοπική μνήμη και μπορεί επίσης να έχει πρόσβαση στη μνήμη που ανήκει σε άλλους επεξεργαστές. Ο χρόνος πρόσβασης σε μνήμη σε αυτό το μοντέλο εξαρτάται από τη θέση μνήμης σε σχέση με τον επεξεργαστή, δεδομένου ότι οι επεξεργαστές μπορούν να έχουν πρόσβαση στην τοπική τους μνήμη γρηγορότερα από τη μη τοπική μνήμη. Η αρχιτεκτονική NUMA σχεδιάστηκε για να ξεπεράσει τα όρια κλιμάκωσης των αρχιτεκτονικών UMA. Μειώνει τη συμφόρηση πολλών επεξεργαστών που ανταγωνίζονται για πρόσβαση στον κοινόχρηστο δίαυλο μνήμης.

Εμπορικοί συμμετρικοί πολυεπεξεργαστές χρησιμοποιήσουν το πρότυπο UMA. Παρόλο που οι προκλήσεις προγραμματισμού είναι πιο δύσκολες για έναν πολυεπεξεργαστή NUMA από ό,τι για έναν πολυεπεξεργαστή UMA, οι μηχανές NUMA μπορούν να κλιμακωθούν σε μεγαλύτερα μεγέθη και μπορούν να έχουν χαμηλότερη καθυστέρηση στην κοντινή μνήμη. Δεδομένου ότι οι επεξεργαστές που λειτουργούν παράλληλα κανονικά θα μοιράζονται δεδομένα, θα πρέπει επίσης να συντονίζονται όταν λειτουργούν σε κοινά δεδομένα. Διαφορετικά, ένας επεξεργαστής θα μπορούσε να αρχίσει να επεξεργάζεται δεδομένα πριν από την ολοκλήρωση άλλου επεξεργαστή. Έτσι, όταν η κοινή χρήση υποστηρίζεται με ένα ενιαίο χώρο διευθύνσεων, πρέπει να υπάρχει ένας μηχανισμός συγχρονισμού. Μια προσέγγιση για την επίτευξη συγχρονισμού είναι η χρήση κλειδαριών για κοινές μεταβλητές. Μόνο ένας επεξεργαστής κάθε φορά μπορεί να αποκτήσει την κλειδαριά και οποιοσδήποτε άλλος επεξεργαστής που ενδιαφέρεται για κοινά δεδομένα πρέπει να περιμένει μέχρι ο αρχικός επεξεργαστής να ξεκλειδώσει τη μεταβλητή.

Αρχιτεκτονική Κατανεμημένης Μνήμης

Η εναλλακτική προσέγγιση για την κοινή χρήση ενός χώρου διευθύνσεων είναι για κάθε επεξεργαστή να έχει τη δική του ιεραρχία κρυφής μνήμης και τον δικό του ιδιωτικό φυσικό χώρο διεύθυνσεων. Ένα δίκτυο αρχιτεκτονικής κατανεμημένης μνήμης αποτελείται από μια ομάδα κόμβων, η κάθε μια αποτελούμενη από έναν επεξεργαστή, μια ιεραρχία τοπικής κρυφής μνήμης και μια τοπική κύρια

μνήμη. Η οργάνωση αυτής της προσέγγισης αρχιτεκτονικής μνήμης, η οποία ονομάζεται επίσης μετάδοσης μηνυμάτων, απεικονίζεται στο Σχήμα 2.3. Το πέρασμα του μηνύματος, που εξυπηρετείται από το δίκτυο διασύνδεσης, είναι ο μόνος τρόπος επικοινωνίας μεταξύ των απομονωμένων κόμβων. Υπό την προϋπόθεση ότι το σύστημα έχει ρουτίνες για την αποστολή και λήψη μηνυμάτων, ο συντονισμός είναι ενσωματωμένος με μηχανισμό μετάδοσης μηνυμάτων, καθώς ένας επεξεργαστής γνωρίζει πότε αποστέλλεται ένα μήνυμα και ο επεξεργαστής λήψης γνωρίζει πότε φτάνει ένα μήνυμα. Εάν ο αποστολέας χρειάζεται επιβεβαίωση ότι το μήνυμα έχει φτάσει, ο επεξεργαστής λήψης μπορεί στη συνέχεια να στείλει ένα μήνυμα επιβεβαίωσης πίσω στον αποστολέα. Σημειώστε ότι σε αντίθεση με το SMP, το δίκτυο διασύνδεσης βρίσκεται μεταξύ κόμβων μνήμης επεξεργαστή.

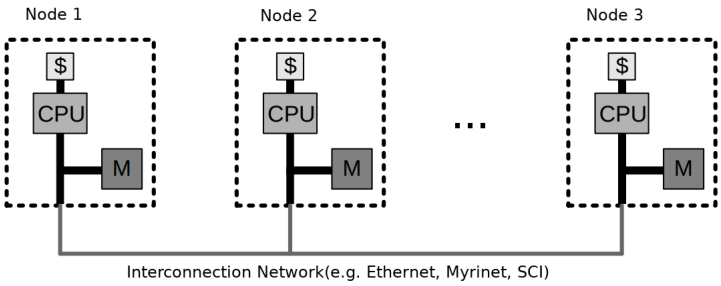


Figure 2.3: Classic organization of distributed memory architecture

Έχουν γίνει αρκετές προσπάθειες για την κατασκευή υπολογιστών μεγάλης κλίμακας βασισμένων σε δίκτυα υψηλής απόδοσης αποστολής μηνυμάτων και παρέχουν καλύτερη απόλυτη απόδοση επικοινωνίας από ό,τι τα clusters που κατασκευάζονται με τοπικά δίκτυα. Πράγματι, πολλοί υπερυπολογιστές χρησιμοποιούν σήμερα προσαρμοσμένα δίκτυα. Το πρόβλημα είναι ότι είναι πολύ πιο ακριβά από τα τοπικά δίκτυα όπως το Ethernet. Λίγες εφαρμογές σήμερα εκτός υπολογιστών υψηλών επιδόσεων μπορούν να δικαιολογήσουν την υψηλότερη απόδοση επικοινωνίας, δεδομένου του πολύ υψηλότερου κόστους.

Το μοντέλο κατανεμημένης μνήμης επιτυγχάνει υψηλή δυνατότητα κλιμάκωσης, καθώς η μνήμη μπορεί να αυξηθεί αναλογικά με τον αριθμό των επεξεργαστών, καθώς και την αποδοτικότητα του κόστους, καθώς οι εμπορικοί επεξεργαστές και η υποδομή δικτύωσης μπορούν να χρησιμοποιηθούν για την κατασκευή τέτοιων συστημάτων. Επιπλέον, κάθε επεξεργαστής μπορεί να έχει γρήγορη πρόσβαση στην τοπική μνήμη του χωρίς τα γενικά έξοδα που προκύπτουν από τις λειτουργίες συνοχής της κρυφής μνήμης. Από την άλλη πλευρά, ο προγραμματισμός είναι πιο δύσκολος καθώς οι προγραμματιστές είναι υπεύθυνοι για τις λεπτομέρειες που σχετίζονται με την επικοινωνία μεταξύ των επεξεργαστών.

Υβριδική Αρχιτεκτονική

Οι αρχιτεκτονικές υβριδικής μνήμης συνδυάζουν τα πλεονεκτήματα των μοντέλων κοινόχρηστης μνήμης και κατανεμημένης μνήμης στο ίδιο υπολογιστικό σύστημα. Σε αυτήν την προσέγγιση αρχιτεκτονικής μνήμης, οι κόμβοι με αρχιτεκτονική κοινής μνήμης συνδέονται μέσω ενός δικτύου διασύνδεσης χρησιμοποιώντας την αρχιτεκτονική κατανεμημένης μνήμης. Η οργάνωση αυτής της προσέγγισης απεικονίζεται στο Σχήμα 2.4. Αυτός ο σχεδιασμός επιτρέπει την παράλληλη επεξεργασία σε κάθε κόμβο και κλιμακώνεται με τον ίδιο τρόπο όπως ένα σύστημα κατανεμημένης μνήμης. Δεδομένου ότι

η προσέγγιση αυτή ενσωματώνει τα πλεονεκτήματα και των δύο προηγούμενων μοντέλων αρχιτεκτονικής μνήμης, είναι η τυπική αρχιτεκτονική των σύγχρονων υπολογιστικών συμπλεγμάτων και των υπερυπολογιστών.

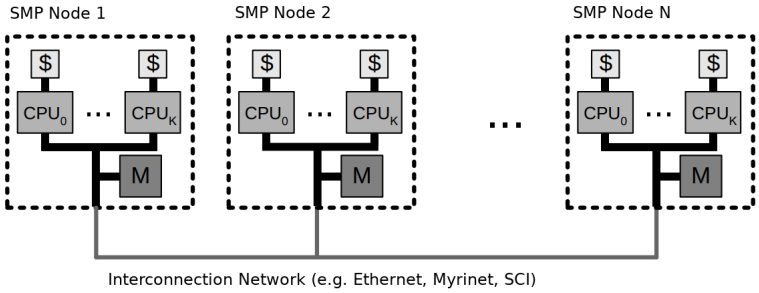


Figure 2.4: Example of a hybrid architecture

2.2 Επιταχυντές

Στην επιστήμη των υπολογιστών, η *επιτάχυνση υλικού* ορίζεται ως η διαδικασία κατά την οποία χρησιμοποιείται εξειδικευμένο υλικό προκειμένου να εκτελούνται λειτουργίες πιο αποδοτικά από ό, τι είναι δυνατό για ένα λογισμικό που τρέχει σε μια CPU γενικού σκοπού. Όταν το υλικό που εκτελεί την επιτάχυνση υπάρχει ως ξεχωριστή μονάδα από τη CPU γενικής χρήσης, τότε αναφέρεται ως *hardware accelerator*.

Οι πιο αξιοσημείωτοι επιταχυντές υλικού αναλύοντε παρακάτω.

2.2.1 Graphics Processing Unit (GPU)

Μια Μονάδα επεξεργασίας γραφικών (GPU) είναι ένας επεξεργαστής ενός τσιπ που εκτελεί ταχείς μαθηματικούς υπολογισμούς, πρωτίστως με σκοπό την απόδοση εικόνων. Στις πρώτες μέρες λειτουργίας των υπολογιστών, η κεντρική μονάδα επεξεργασίας (CPU) πραγματοποιούσε αυτούς τους υπολογισμούς. Καθώς αναπτύχθηκαν περισσότερες εφαρμογές γραφικών, ωστόσο, οι απαιτήσεις τους έφεραν πίεση στην CPU και υποβάθμιζαν τις επιδόσεις. Οι μονάδες GPU εμφανίστηκαν ως ένας τρόπος για να εκφορτωθούν αυτές οι εργασίες από τις CPU, απελευθερώνοντας την ισχύ επεξεργασίας τους. Το 1999, η NVIDIA παρουσίασε την πρώτη GPU στον κόσμο, το GeForce 256 [24]. Εισήχθη στο κόσμο των υπολογιστών ως εξής:

> A single chip processor with integrated transform, lighting, triangle setup/clipping, and rendering engines that is capable of processing a minimum of 10 million polygons per second.

Μπορούσε να επεξεργαστεί 10 εκατομμύρια πολυγώνια ανά δευτερόλεπτο, επιτρέποντάς του να εκφορτώσει μια σημαντική ποσότητα επεξεργασίας γραφικών από την CPU. Η ATI Technologies κυκλοφόρησε το Radeon 9700 το 2002 χρησιμοποιώντας τον όρο μονάδα οπτικής επεξεργασίας (VPU). Τα τελευταία χρόνια σημειώθηκε σημαντική αύξηση στις επιδόσεις και τις δυνατότητες των

GPU λόγω της ζήτησης της αγοράς για πιο εξελιγμένα γραφικά. Επιπλέον, με την πάροδο του χρόνου, η ισχύς επεξεργασίας των GPU έκανε τις κάρτες μια δημοφιλής επιλογή για άλλες υπολογιστικά εντατικές εργασίες που δεν σχετίζονται με τα γραφικά.

Σήμερα, οι GPU χρησιμοποιούνται ευρέως σε ενσωματωμένα συστήματα, κινητά τηλέφωνα, προσωπικούς υπολογιστές και κονσόλες παιχνιδιών. Οι σύγχρονες μονάδες GPU είναι πολύ αποδοτικές στο χειρισμό γραφικών καθώς και στην επεξεργασία εικόνων. Επιπλέον, η άκρως παράλληλη δομή τους, τις καθιστά πιο αποτελεσματικές από τους επεξεργαστές γενικής χρήσης για αλγορίθμους όπου η επεξεργασία μεγάλων τμημάτων δεδομένων γίνεται παράλληλα. Ως αποτέλεσμα, προέκυψε μεγάλη διαφορά στις ικανότητες αριθμητικών πράξεων κινητού σημείου μεταξύ της CPU και της GPU. Ο κύριος λόγος είναι ότι οι μονάδες GPU είναι εξειδικευμένες για υπολογιστικά εντατικές και εξαιρετικά παράλληλες πράξεις και ως εκ τούτου έχουν σχεδιαστεί, όπως τα περισσότερα τρανζίστορ, αφιερωμένα στην επεξεργασία δεδομένων αντί για προσωρινή αποθήκευση δεδομένων και έλεγχο ροής, όπως συμβαίνει στην περίπτωση μιας CPU. Ειδικότερα, η GPU είναι κατάλληλη για την αντιμετώπιση προβλημάτων που μπορούν να εκφραστούν απο παράλληλους υπολογισμούς δεδομένων με υψηλή αριθμητική ένταση. Επειδή το ίδιο πρόγραμμα εκτελείται σε κάθε στοιχείο δεδομένων, υπάρχει μικρότερη απαίτηση για εξελιγμένο έλεγχο ροής. Η λανθάνουσα πρόσβαση στην μνήμη μπορεί να κρυφτεί με υπολογισμούς αντί για μεγάλες κρυφές μνήμες δεδομένων. Ως εκ τούτου, οι GPU μπορούν να θεωρηθούν επεξεργαστές γενικής χρήσης, υψηλής απόδοσης, πολλών πυρήνων, ικανοί για πολύ υψηλό υπολογισμό και διακίνηση μνήμης.

General Purpose Graphics Processing Unit (GPGPU)

Μια GPU γενικής χρήσης (GPGPU) είναι μια μονάδα επεξεργασίας γραφικών (GPU) που εκτελεί μη εξειδικευμένους υπολογισμούς που τυπικά θα διεξάγονται από την CPU (κεντρική μονάδα επεξεργασίας). Συνήθως, η GPU είναι αφιερωμένη στην απόδοση γραφικών.

GPGPUs χρησιμοποιούνται για εργασίες που ήταν παλαιότερα ο τομέας των CPU υψηλής επίδοσης, όπως υπολογισμοί φυσικής, κρυπτογράφηση / αποκρυπτογράφηση, επιστημονικοί υπολογισμοί και η δημιουργία κρυπτονομισμάτων όπως το Bitcoin. Επειδή οι κάρτες γραφικών κατασκευάζονται για μαζικό παραλληλισμό, μπορούν να υπερνικήσουν το ρυθμό υπολογισμού ακόμη και των πιο ισχυρών CPU για πολλές παράλληλες εργασίες επεξεργασίας.

Οι υπολογισμοί γενικής χρήσης σε GPU έγιναν πρακτικοί και δημοφιλείς μετά από περίπου το 2001, με την εμφάνιση και των προγραμματιζόμενων shaders και της υποστήριξης κινητής υποδιαστολής σε επεξεργαστές γραφικών. Συγκεκριμένα, τα προβλήματα με μήτρες ή / και διανυσμάτα - ειδικά δύο, τριών ή τεσσάρων διαστάσεων διανυσμάτα - ήταν εύκολο να μεταφραστούν σε μια GPU, η οποία ενεργεί με εγγενή ταχύτητα και υποστήριξη σε αυτούς τους τύπους.

Η αρχιτεκτονική των GPU, με πολλούς πυρήνες και μεγάλο εύρος ζώνης στην ιδιωτική μνήμη τους, σχεδιασμένη για την αποτελεσματική παράλληλη εκτέλεση πολλών εργασιών, είναι ιδανική για εφαρμογές που κάνουν πολλούς ταυτόχρονους υπολογισμούς (όπως τα μαθηματικά μοντέλα). Αυτός είναι ο λόγος για τον οποίο η επιτάχυνση με GPU - η παράλληλη χρήση μιας GPU,με την CPU, για την αποτελεσματική εκτέλεση δαπανηρών εφαρμογών - γίνεται ολοένα και πιο δημοφιλής τα τελευταία χρόνια σε διάφορους τομείς [25, 26, 27, 28]. Προκειμένου να αξιοποιηθεί η ισχύ που μπορούν να προσφέρουν οι GPU σε υπολογισμούς γενικού σκοπού, έχουν σχεδιαστεί και υλοποιηθεί διάφορες διεπαφές προγραμματισμού. Τα πιο δημοφιλή και ευρέως χρησιμοποιούμενα πλαίσια για το GPGPU παρατίθενται παρακάτω..

1. OpenCL: Το Open Computing Language (OpenCL) είναι ένα framework για την εγγραφή

προγραμμάτων που εκτελούνται σε ετερογενείς πλατφόρμες αποτελούμενες από κεντρικές μονάδες επεξεργασίας (CPU), μονάδες επεξεργασίας γραφικών (GPU), ψηφιακούς επεξεργαστές σημάτων (DSPs), προγραμματισμένες σειρές πύλης (FPGAs) και άλλους επεξεργαστές ή επιταχυντές υλικού. Το OpenCL είναι ένα ανοικτό πρότυπο που διατηρείται από την ομάδα Khronos [29].

2. CUDA: Το Compute Unified Device Architecture (CUDA) είναι μια πλατφόρμα παράλληλων υπολογισμών που αναπτύχθηκε από τη NVIDIA και εισήχθη το 2006. Επιτρέπει στα προγράμματα λογισμικού να εκτελούν υπολογισμούς χρησιμοποιώντας τόσο τη CPU όσο και τη GPU. Μοιράζοντας το φορτίο επεξεργασίας με τη GPU (αντί να χρησιμοποιείτε μόνο την CPU), τα προγράμματα με CUDA μπορούν να επιτύχουν σημαντικές αυξήσεις στην απόδοση [30].

2.2.2 Field-Programmable Gate Array (FPGA)

Μια συσκεύη field-programmable gate array (FPGA) είναι ένα ολοκληρωμένο κύκλωμα (IC) που μπορεί να προγραμματιστεί στον πεδίο μετά την κατασκευή. Η διαμόρφωση FPGA καθορίζεται γενικά χρησιμοποιώντας μια γλώσσα περιγραφής υλικού (HDL), παρόμοια με αυτή που χρησιμοποιείται για ένα ολοκληρωμένο κύκλωμα (ASIC) για συγκεκριμένη εφαρμογή. Τα FPGA είναι παρόμοια κατ 'αρχήν, αλλά έχουν πολύ ευρύτερη πιθανή εφαρμογή απ' ότι τα προγραμματιζόμενα chip μνήμης μόνο για ανάγνωση (PROM). Τα FPGA περιέχουν μια σειρά προγραμματιζόμενων λογικών μπλοκ και μια ιεραρχία επαναπροσδιοριζόμενων διασυνδέσεων που επιτρέπουν τη δέσμευση των μπλοκ μαζί, όπως πολλές πύλες λογικής που μπορούν να διασυνδεθούν σε διαφορετικές διαμορφώσεις. Τα λογικά μπλοκ μπορούν να διαμορφωθούν έτσι ώστε να εκτελούν πολύπλοκες συνδυαστικές λειτουργίες ή απλά λογικές πύλες όπως οι AND και XOR. Στα περισσότερα FPGAs, τα λογικά μπλοκ περιλαμβάνουν επίσης στοιχεία μνήμης, τα οποία μπορεί να είναι απλά flip-flops ή πιο ολοκληρωμένα μπλοκ μνήμης. Τα FPGA έχουν συγκεντρώσει αυξημένη προσοχή καθώς αποδεικνύεται ότι είναι σε θέση να παρέχουν σημαντική ώθηση στις εφαρμογές σε υπολογιστικά κέντρα που χρησιμοποιούν ετερογένεια στα σχέδιά τους.

2.2.3 Coprocessor

Ένας συνεπεξεργαστής είναι ένας επεξεργαστής υπολογιστή που χρησιμοποιείται για να συμπληρώσει τις λειτουργίες του πρωτεύοντος επεξεργαστή (CPU). Οι λειτουργίες που εκτελούνται από τον συνεπεξεργαστή μπορεί να είναι αριθμητική με κινητή υποδιαστολή, γραφικά, επεξεργασία σήματος, επεξεργασία γραμμών, κρυπτογράφηση ή διασύνδεση I/O με περιφερειακές συσκευές. Με την εκφόρτωση εργασιών που απαιτούν επεξεργασία από τον κύριο επεξεργαστή, οι συνεπεξεργαστές μπορούν να επιταχύνουν την απόδοση του συστήματος.

2.3 Intel Xeon Phi

Ως πρώτυπο για το προτεινόμενο σύστημα μας, στοχεύουμε τον Coprocessor Intel Xeon Phi, μέλος της οικογένειας επεξεργαστών Xeon Phi της Intel. Η οικογένεια επεξεργαστών Xeon Phi αποτελείται από μια σειρά μαζικά παράλληλων x86 πολυπύρηνων επεξεργαστών συμβατών με τις τυπικές γλώσσες προγραμματισμού, API και πρότυπα όπως το OpenMP [31] και το MPI. Από το 2017, ο Intel Xeon Phi υπάρχει μέσα στους τέσσερις από τους δέκα κορυφαίους υπερυπολογιστές στον κόσμο [32]. Η οικογένεια επεξεργαστών Xeon Phi της Intel χρησιμοποιεί την αρχιτεκτονική

MIC (Multi Integrated Core), αξιοποιώντας την αρχιτεκτονική x86 και επεκτείνοντας τη συμβατότητά της με υπάρχοντα εργαλεία προγραμματισμού, μεταγλωττιστές και βιβλιοθήκες. Οι εφαρμογές που προορίζονται για επεξεργαστες MIC μπορούν εύκολα να εκτελεστούν σε έναν τυπικό επεξεργαστή Intel Xeon x86. Η κύρια διαφορά μεταξύ του Xeon Phi και ενός GPGPU όπως το Nvidia Tesla είναι ότι ο Xeon Phi, με έναν πυρήνα συμβατό με x86, μπορεί, με λιγότερες τροποποιήσεις, να τρέξει λογισμικό που αρχικά στοχεύει σε μια τυπική CPU x86. Στη δουλειά μας, διερευνάμε τα οφέλη από τη διάθεση των πόρων του Xeon Phi από απόσταση και από κοινού σε ένα σύμπλεγμα υπολογιστών. Η υποκείμενη αρχιτεκτονική ενός επιταχυντή Intel Xeon Phi απεικονίζεται στο σχήμα 2.5.

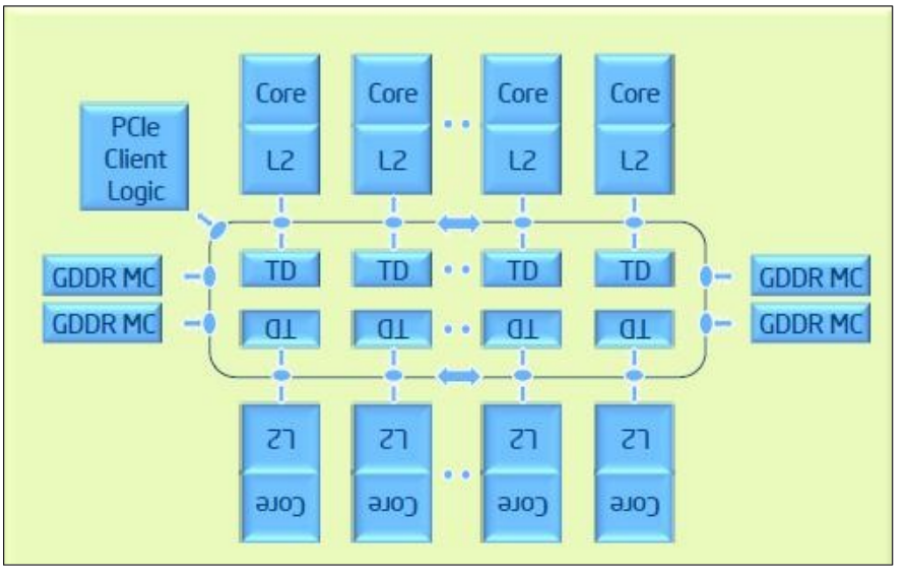


Figure 2.5: Intel Xeon Phi Architecture Ring and Cores

Source: Intel's MPSS User's Guide

2.3.1 Xeon Phi Execution Models

Ο συνεπεξεργαστής Xeon Phi υποστηρίζει διαφορετικά μοντέλα εκτέλεσης με βάση την προβλεπόμενη λειτουργικότητα και το σχήμα εκτέλεσης του προς εκτέελση προγράμματος. Τα μοντέλα εκτέλεσης μπορούν να συνοψιστούν σε τρεις κατηγορίες: *native*, *offload* και *symmetric*.

1. Native Mode: Σε αυτή τη λειτουργία, μια εφαρμογή μπορεί να εκτελεστεί εξ ολοκλήρου στη συσκευή. Για να λειτουργήσει σε native mode, η εφαρμογή πρέπει να γίνει compile για το λειτουργικό περιβάλλον του Xeon Phi και να μεταφερθεί απευθείας στον συνεπεξεργαστή. Η Intel παρέχει τα απαραίτητα εργαλεία για να υποστηρίξει τη λειτουργία native mode εκτέλεσης μέσω του Manycore Platform Software Stack, της συλλογής λογισμικού της Intel που υποστηρίζει τις λειτουργίες του συνεπεξεργαστή.

2. Offload Mode: Επίσης γνωστός και ως ετερογενής τρόπος προγραμματισμού, η εκτέλεση μιας εφαρμογής χωρίζεται μεταξύ του κεντρικού υπολογιστή και του συνεπεξεργαστή με την

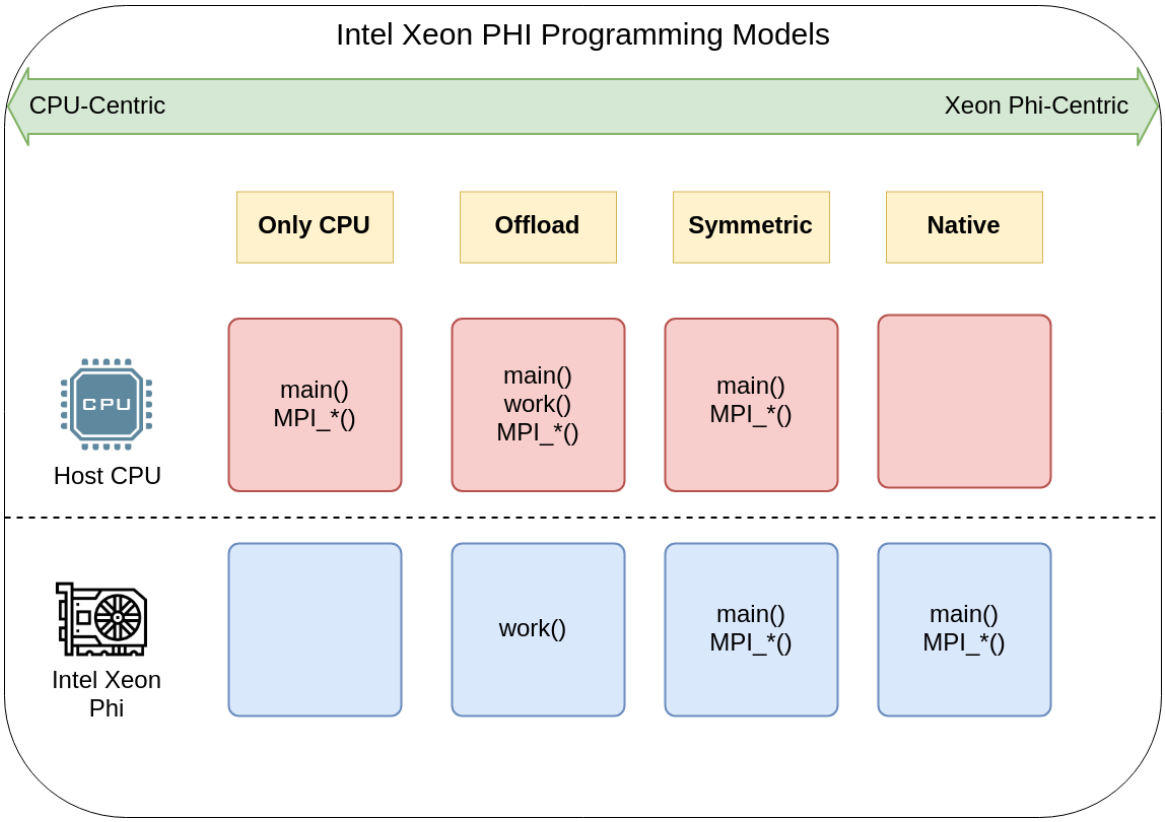


Figure 2.6: Intel Xeon Phi Programming and Execution Models

CPU να εκφορτώνει εργασίες που απαιτούν επεξεργασία για εκτέλεση στον συνεπεξεργαστή. Ο τρόπος εκφόρτωσης υποστηρίζεται και ενισχύεται από παραδοσιακά πλαίσια παράλληλων υπολογιστών όπως το OpenMP.

3. Symetric Mode: Σε αυτό το σενάριο η εφαρμογή εκτελείται ταυτόχρονα στο μηχάνημα υποδοχής και στο Intel Xeon Phi. Ο συνεπεξεργαστής αναγνωρίζεται από τον κεντρικό υπολογιστή ως ανεξάρτητο κόμβο επεξεργασίας και τα δύο μέρη επικοινωνούν μέσω διεπαφής μυνημάτων όπως MPI.

Τα μοντέλα προγραμματισμού που αντιστοιχούν στα προαναφερθέντα μοντέλα εκτέλεσης απεικονίζονται στο Σχήμα 2.6. Εκτός από τα μοντέλα εκτέλεσης Xeon Phi, το μοντέλο "Μόνο CPU" που απεικονίζεται στο Σχήμα 2.6 αντιστοιχεί σε μια κανονική εκτέλεση μιας εφαρμογής που πραγματοποιείται αποκλειστικά στη CPU χωρίς εμπλοκή του επιταχυντή.

2.3.2 Intel ManyCore Platform Software Stack

Η Intel παρέχει μια συλλογή από αξιόπιστο λογισμικό που είναι απαραίτητο για τη λειτουργία του Coprocessor Intel Xeon Phi. Το MPSS περιλαμβάνει το λογισμικό που προορίζεται τόσο για τη μηχανή υποδοχής όσο και για τον συνεπεξεργαστή και το απαραίτητο πλαίσιο για την αποτελεσματική επικοινωνία και τον συντονισμό των λειτουργιών του συνεπεξεργαστή. Μεταξύ των εξέχοντων

χαρακτηριστικών της Multicore Platform Stack της Intel, το Coprocessor Offload Infrastructure (COI) είναι απαραίτητο για την εργασία μας καθώς εκθέτει το απαραίτητο API για την εκφόρτωση εκτελέσιμων αρχείων και δεδομένων στον συνεπεξεργαστή και χρησιμοποιεί τον απαραίτητο μηχανισμό για την παροχή ενός εικονικού μοντέλου μνήμης που απλοποιεί την ανταλλαγή δεδομένων μεταξύ των διαδικασιών στον κεντρικό υπολογιστή και σε κάθε συνεπεξεργαστή. Το COI και άλλα εξαρτήματα του Intel MPSS βασίζονται στο API του Symmetric Interface Communication (SCIF) για τις υπηρεσίες επικοινωνίας PCIe μεταξύ του κεντρικού επεξεργαστή και των συνεπεξεργαστών. Το SCIF παρέχει πολύ υψηλές ταχύτητες μεταφοράς δεδομένων πάνω απο PCIe, ενώ αφαιρεί τις λεπτομέρειες επικοινωνίας μέσω του PCIe [33]. Μια απλοποιημένη έκδοση του MPSS, με τα εξαρτήματα που σχετίζονται με την εργασία μας, απεικονίζεται στο Σχήμα 2.7.

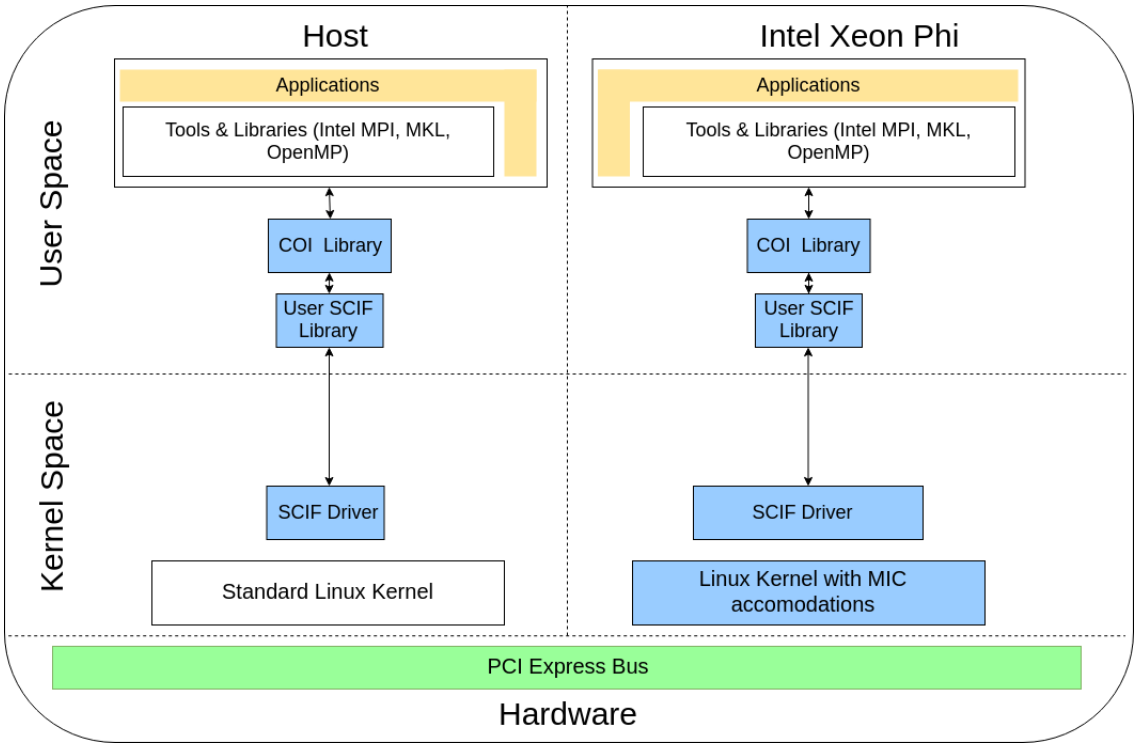


Figure 2.7: ManyCore Platform Software Stack Components Overview

2.3.3 Symmetric Communication Interface (SCIF)

Το Symmetric Communication Interface της Intel ή SCIF παρέχει το μηχανισμό επικοινωνίας μεταξύ κόμβων σε μια ενιαία πλατφόρμα, όπου ένας κόμβος είναι ένας coprocessor Intel Xeon Phi ή ένα σύμπλεγμα επεξεργαστών Intel Xeon. Συγκεκριμένα, το SCIF αφαιρεί τις λεπτομέρειες της επικοινωνίας μέσω του διαύλου PCIe παρέχοντας ένα API που είναι συμμετρικό μεταξύ των συσκευών υποδοχής και της αρχιτεκτονικής MIC. Η αρχιτεκτονική του λογισμικού Intel MIC υποστηρίζει ένα υπολογιστικό μοντέλο στο οποίο το φορτίο μπορεί να διανεμηθεί τόσο στο σύνθετο επεξεργαστή κεντρικών υπολογιστών Intel Xeon όσο και στους συνεπεξεργαστές με Intel MIC Architecture. Μια σημαντική ιδιότητα του SCIF είναι η συμμετρία. Τα προγράμματα οδήγησης SCIF πρέπει να

παρουσιάζουν την ίδια διεπαφή τόσο στον επεξεργαστή κεντρικού υπολογιστή όσο και στον επεξεργαστή Intel MIC Architecture, ώστε να είναι δυνατή η εκτέλεση του λογισμικού που έχει εγγραφεί στο SCIF όπου αυτό είναι καταλληλότερο. Δεδομένου ότι ο συνεργαζόμενος επεξεργαστής Intel MIC Architecture μπορεί να χρησιμοποιεί διαφορετικό λειτουργικό σύστημα από αυτόν που εκτελείται στον κεντρικό υπολογιστή, η αρχιτεκτονική SCIF έχει σχεδιαστεί ώστε να είναι ανεξάρτητη από το λειτουργικό σύστημα. Αυτό εξασφαλίζει ότι οι εφαρμογές SCIF σε διαφορετικά λειτουργικά συστήματα μπορούν να αλληλοσυμπληρώνονται. Το SCIF υποστηρίζει την επικοινωνία μεταξύ των επεξεργαστών υποδοχής Xeon και των συνεπεξεργαστών Intel MIC Architecture σε μια ενιαία πλατφόρμα. Η επικοινωνία μεταξύ τέτοιων στοιχείων που βρίσκονται σε ξεχωριστές πλατφόρμες μπορεί να πραγματοποιηθεί χρησιμοποιώντας τυπικά κανάλια επικοινωνίας όπως Infiniband και TCP / IP. Μια εφαρμογή SCIF σε έναν συμβατικό υπολογιστή ή ένα συνεπεξεργαστή Intel MIC περιλαμβάνει τόσο μια βιβλιοθήκη χώρου χρήστη όσο και ένα πρόγραμμα οδήγησης χώρου πυρήνα όπως φαίνεται στο Σχήμα 2.7. Τα περισσότερα από τα στοιχεία του Intel MPSS χρησιμοποιούν το SCIF για επικοινωνία.

Ο οδηγός SCIF παρέχει ένα αξιόπιστο στρώμα μηνυμάτων που βασίζεται στη σύνδεση, καθώς και λειτουργικότητα που αφαιρεί τις λειτουργίες RMA. Το SCIF παρέχει έναν μηχανισμό επικοινωνίας μεταξύ διαφόρων κόμβων SCIF. Ένας κόμβος SCIF είναι ένα φυσικό τελικό σημείο στο δίκτυο SCIF. Η Κεντρική Μονάδα Επεξεργασίας και ο συνεπεξεργαστής MIC είναι κόμβοι SCIF. Η διαδικασία δημιουργίας σύνδεσης μεταξύ διαφόρων κόμβων SCIF είναι παρόμοια με τον προγραμματισμό socket, με παρόμοια σημασιολογία να χρησιμοποιείται από το SCIF: `scif_open()`, `scif_bind()`, `scif_listen()`, `scif_connect()`, `scif_accept()`. Συνεπώς, η λειτουργικότητα που εφαρμόζουν οι προαναφερθείσες συναρτήσεις SCIF είναι παρόμοια με αυτή των socket συναρτήσεων. Μια τυπική ροή σύνδεσης μεταξύ δύο διαφορετικών κόμβων στο δίκτυο SCIF απεικονίζεται στο Σχήμα 2.8.

SCIF Messaging Layer

Αφού έχει δημιουργηθεί μια σύνδεση, είναι δυνατή η ανταλλαγή μηνυμάτων μεταξύ των διαδικασιών που κατέχουν τα συνδεδεμένα τελικά σημεία. Ένα μήνυμα που αποστέλλεται απο ένα συνδεδεμένο τελικό σημείο λαμβάνεται απο το άλλο συνδεδεμένο τελικό σημείο. Η επικοινωνία αυτή είναι αμφίδρομη. Το στρώμα μηνυμάτων του SCIF αποτελείται από socket-like συναρτήσεις `scif_send()` και `scif_recv()`. Τα μηνύματα αποστέλλονται πάντα μέσω του τοπικού τελικού σημείου για παράδοση σε απομακρυσμένο τελικό σημείο. Για κάθε συνδεδεμένο ζεύγος τελικών σημείων, υπάρχει ένα αποκλειστικό ζεύγος ουρών μηνυμάτων - μία ουρά για κάθε κατεύθυνση επικοινωνίας. Με αυτόν τον τρόπο, η πρόοδος της οποιασδήποτε δεν περιορίζεται από την πρόοδο της άλλης, κάτι που μπορεί να συνέβαινε με πολλαπλές συνδέσεις που μοιράζονται ένα ζεύγος ουράς. Ένα μήνυμα μπορεί να είναι έως και 2^{31}-1 bytes long. Παρ 'όλα αυτά, το στρώμα ανταλλαγής μηνυμάτων προορίζεται για την αποστολή σύντομων μηνυμάτων τύπου εντολής, όχι για μαζικές μεταφορές δεδομένων. Οι ουρές στρώσεων μηνυμάτων είναι σχετικά μικρές. ένα μακρύ μήνυμα μεταδίδεται ως πολλαπλές μικρότερες μεταφορές μήκους ουράς, με μια ανταλλαγή διακοπής για κάθε τέτοια μεταφορά. Επομένως, η λειτουργία SCIF RMA θα πρέπει να χρησιμοποιείται για την αποστολή μεγαλύτερων μονάδων δεδομένων, π.χ. περισσότερο από 4KiB.

2.3.4 SCIF Remote Memory Access

Το SCIF κάνει χρήση ενός μηχανισμού που επιτρέπει την πρόσβαση σε απομακρυσμένη μνήμη (RMA) από τη μνήμη του κεντρικού επεξεργαστή στη μνήμη του συν-επεξεργαστή. [34] Αυτός

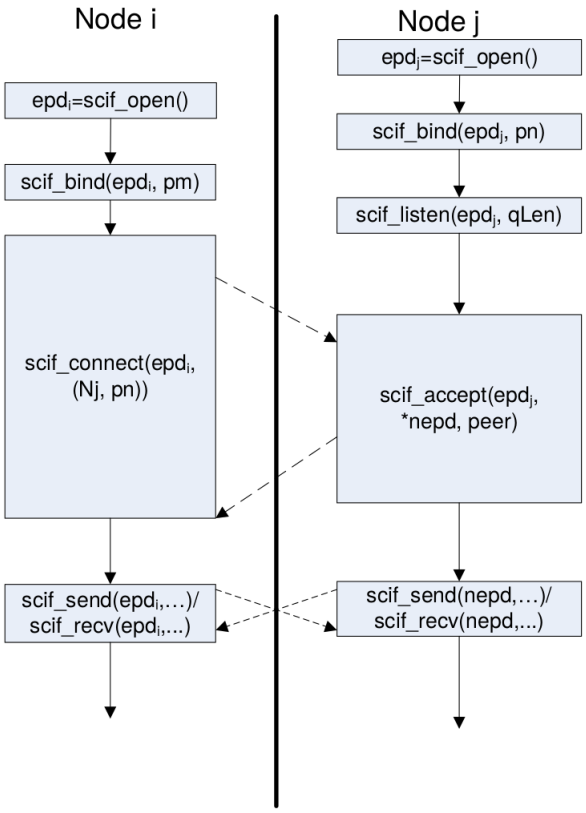


Figure 2.8: Connecting two different SCIF nodes

Source: Intel's SCIF User Guide

ο μηχανισμός επιτρέπει λειτουργίες μεταφοράς υψηλής ταχύτητας, χαμηλής καθυστέρησης μεταξύ των δυο μερών. Ο μηχανισμός εξαρτάται στο μηχανισμό εγγραφής μνήμης (Memory Registration), στην οποία μια διαδικασία εκθέτει ένα φάσμα σελίδων μνήμης στον εικονικό χώρο διευθύνσεών της, για πρόσβαση από άλλη διαδικασία, συνήθως από μια διαδικασία που βρίσκεται σε απομακρυσμένο κόμβο. Οι λειτουργίες πρόσβασης σε απομακρυσμένη μνήμη χρησιμοποιούνται για τη μεταφορά ενός εκτελέσιμου δυαδικού αρχείου που προορίζεται για εκτέλεση στον συν-επεξεργαστή, τις κοινές βιβλιοθήκες καθώς και για τη διμερή μεταφορά μεγάλων προσωρινών δεδομένων.

Προκειμένου μια διεργασία να κάνει map μνήμη που ανήκει σε μια απομακρυσμένη διαδικασία, που βρίσκεται είτε στον κεντρικό υπολογιστή είτε στον επιταχυντή, πρέπει πρώτα να εγγραφεί(register) με το πρόγραμμα οδήγησης (driver) SCIF. Κάθε συνδεδεμένο τελικό σημείο έχει έναν μεμονομένο χώρο διεύθυνσης. Ο χώρος διεύθυνσης καταχώρησης (Registered Address Space) είναι ένας επιπλέον χώρος διευθύνσεων που διαχειρίζεται το πρόγραμμα οδήγησης SCIF, που αντιπροσωπεύει την τοπική φυσική μνήμη. Το SCIF Driver καταχωρεί ένα φάσμα εικονικής μνήμης χώρου χρήστη με `scif_register()`, με τη μορφή καταχωρημένου παραθύρου και επιστρέφεται μια διεύθυνση offset με την οποία μπορεί να έχει πρόσβαση στον Registered χώρος διευθύνσεων και από τα δύο μέρη που εμπλέκονται σε μια RMA λειτουργία. Τα επικοινωνούντα μέρη μπορούν να παραπέμπουν και να έχουν πρόσβαση στο καταχωρημένο παράθυρο διαβιβάζοντας την διεύθυνση offset στις σχετικές κλήσεις

SCIF RMA API. Η αντιστοίχιση μεταξύ του καταχωρημένου χώρου διευθύνσεων και της φυσικής μνήμης παραμένει ακόμη και αν το συγκεκριμένο εικονικό εύρος διευθύνσεων χάσει την αντιστοίχιση ή εαν γίνει map σε διαφορετική θέση μνήμης.

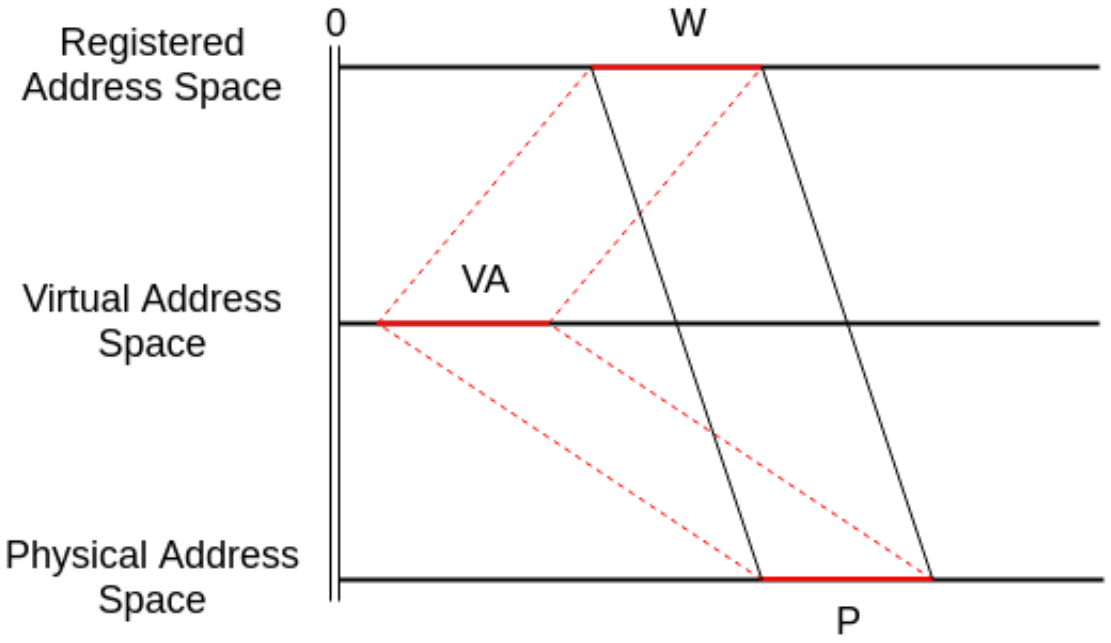


Figure 2.9: Memory Registration Mechanism

Στο σχήμα 2.9 απεικονίζεται ο μηχανισμός εγγραφής μνήμης. Το σχήμα παρουσιάζει ένα καταχωρημένο παράθυρο W που δημιουργήθηκε από κλήση `scif_register()`. Οι σελίδες του W, ένα εύρος στο εγγεγραμμένο χώρο διευθύνσεων κάποιου τοπικού τελικού σημείου SCIF, αντιπροσωπεύουν κάποιο σύνολο, P, φυσικών σελίδων στην τοπική μνήμη. P είναι το σύνολο φυσικών σελίδων που υποστήριζαν ένα καθορισμένο εύρος εικονικών διευθύνσεων, VA, τη στιγμή που εκτελέστηκε η `scif_register()`. Ακόμη και αν το εύρος εικονικών διευθύνσεων, VA, στη συνέχεια χαρτογραφηθεί σε διαφορετικές φυσικές σελίδες, το W συνεχίζει να αντιπροσωπεύει το P. Φυσικά, εάν το εικονικό εύρος διευθύνσεων χάσει το mapping ή γίνει map σε διαφορετικές φυσικές σελίδες, η διαδικασία δεν έχει τρόπο πρόσβασης στην καταχωρημένη μνήμη προκειμένου να διαβάζει ή να γράφει δεδομένα RMA εκτός αν αυτές οι φυσικές σελίδες επιστρέφουν κάποιο άλλο εικονικό εύρος διευθύνσεων.

Οι λειτουργίες SCIF RMA προορίζονται να υποστηρίξουν το μοντέλο επικοινωνίας μονής κατεύθυνσης που έχειτο πλεονέκτημα ότι μια λειτουργία ανάγνωσης / εγγραφής μπορεί να πραγματοποιηθεί από τη μια πλευρά μιας σύνδεσης όταν γνωρίζει τόσο την τοπική όσο και την απομακρυσμένη τοποθεσία των δεδομένων που πρόκειται να μεταφερθούν. Οι μονομερείς κλήσεις μπορούν συχνά είναι χρήσιμοι για αλγορίθμους στους οποίους ο συγχρονισμός θα ήταν δυσάρεστος (π.χ. πολλαπλασιασμός κατανεμημένου πλέγματος) ή όπου είναι επιθυμητό οι εργασίες να είναι σε θέση να εξισορροπούν το φορτίο τους ενώ άλλοι επεξεργαστές λειτουργούν σε δεδομένα.

Οι λειτουργίες `scif_readfrom()` και `scif_writeto()` εκτελούν λειτουργίες ανάγνωσης και εγγραφής DMA ή CPU, αντίστοιχα, μεταξύ της φυσικής μνήμης των τοπικών και των απομακρυσμένων κόμβων του καθορισμένου τελικού σημείου και του ομότιμου κόμβου. Η φυσική μνήμη είναι αυτή που αντιπροσωπεύεται από καθορισμένες κλίμακες στους τοπικούς και απομακρυσμένους

καταχωρημένους χώρους διεύθυνσης ενός τοπικού τελικού σημείου και του απομακρυσμένου τελικού σημείου του. Ο καθορισμός αυτών των καταχωρημένων περιοχών διευθύνσεων δημιουργεί μια αντιστοιχία μεταξύ τοπικών και απομακρυσμένων φυσικών σελίδων για τη διάρκεια της λειτουργίας RMA. Τα συγκεκριμένα RMA flags παράμετροι που μεταβιβάζονται στις σχετικές λειτουργίες του SCIF RMA ελέγχουν αν η μεταφορά είναι βασισμένη σε DMA ή CPU.

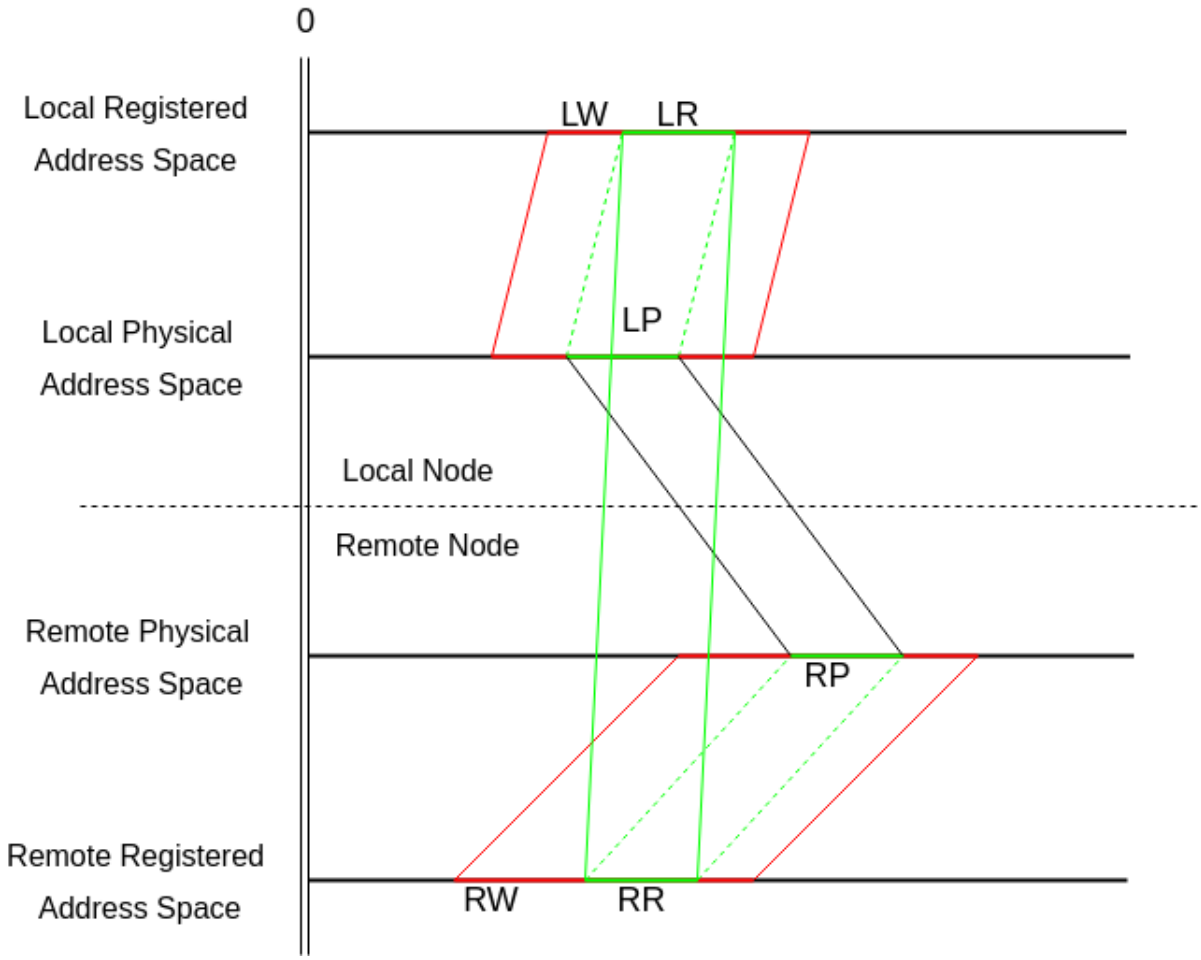


Figure 2.10: SCIF RMA Mapping Between Remote Nodes

Στο σχήμα 2.10 απεικονίζουμε μια τέτοια χαρτογράφηση μεταξύ δύο απομακρυσμένων κόμβων SCIF, οι οποίοι θα εκτελέσουν μια διαδικασία RMA. Η διαδικασία που εκτελεί τη λειτουργία καθορίζει ένα εύρος, LR, εντός της καταχωρημένης διεύθυνσης ενός από τα συνδεδεμένα τελικά σημεία της, και ένα αντίστοιχο εύρος, RR, του ίδιου μήκους εντός του καταχωρημένου χώρου διεύθυνσης του τελικού σημείου του ομότιμου. Κάθε καθορισμένη περιοχή πρέπει να βρίσκεται εξ ολοκλήρου μέσα σε ένα ήδη καταχωρημένο παράθυρο ή παράλληλα παράθυρα των αντίστοιχων καταχωρημένων χώρων διεύθυνσης. Οι σταθερές πράσινες γραμμές αντιπροσωπεύουν την αντιστοιχία μεταξύ των καθορισμένων περιοχών στους τοπικούς και απομακρυσμένους καταχωρημένους χώρους διευθύνσεων. οι διακεκομμένες πράσινες γραμμές αντιπροσωπεύουν τις προβολές στους αντίστοιχους φυσικούς χώρους διεύθυνσης. Αυτό ορίζει μια συνολική αποτελεσματική αλληλογραφία (μαύρες γραμμές) μεταξύ του

φυσικού χώρου διευθύνσεων του τοπικού κόμβου και του απομακρυσμένου κόμβου του χώρου διευθύνσεων που έχει καταχωρηθεί από ομότιμους. Ως εκ τούτου, μια λειτουργία DMA θα μεταφέρει δεδομένα μεταξύ LP και RP (και πάλι, LP και RP δεν είναι συνήθως συνεχόμενα).

2.4 Sockets - TCP/IP

Ένα socket ορίζεται ως ένα τελικό σημείο επικοινωνίας. Ένα ζεύγος διεργασιών επικοινωνούν μέσω δικτύου χρησιμοποιόντας ένα ζεύγος socket - ένα για κάθε διεργασία. Ένα socket αναγνωρίζεται από μια διεύθυνση IP συνδεδεμένη με έναν αριθμό θύρας. Σε γενικές γραμμές, τα socket χρησιμοποιούν μια αρχιτεκτονική πελάτη-διακομιστή. Ο διακομιστής περιμένει για τα εισερχόμενα αιτήματα πελατών, ακούγοντας σε μια προκαθορισμένη θύρα. Μόλις ληφθεί ένα αίτημα, ο διακομιστής δέχεται μια σύνδεση από την υποδοχή προγράμματος-πελάτη για να ολοκληρώσει τη σύνδεση. Όταν μια διαδικασία πελάτη εκκινεί ένα αίτημα για μια σύνδεση, εκχωρείται μια θύρα από τον κεντρικό υπολογιστή της. Αυτή η θύρα έχει κάποια αυθαίρετη αριθμό μεγαλύτερη από 1024.

Στη δουλειά μας, χρησιμοποιούμε τις υποδοχές Berkeley ή BSD Sockets. Οι υποδοχές Berkeley είναι μια διασύνδεση προγραμματισμού εφαρμογών (API) για Internet και Unix sockets, που χρησιμοποιούνται για επικοινωνία μεταξύ διεργασιών (IPC). Προέρχεται από το 4.2 BSD Unix που κυκλοφόρησε το 1983. Μια υποδοχή είναι μια αφηρημένη αναπαράσταση (handle) για το τοπικό τελικό σημείο μιας διαδρομής επικοινωνίας δικτύου. Στο API του Berkeley sockets, ένα socket αντιπροσωπεύετε ως ένας περιγραφέας αρχείου (file handle) στη φιλοσοφία Unix που παρέχει μια κοινή διεπαφή για την είσοδο και την έξοδο σε ροές δεδομένων. Η εφαρμογή BSD Sockets βρίσκεται στην κορυφή της στοίβας TCP/IP. Η σουίτα πρωτοκόλλου TCP/IP Stack ή Internet Protocol παρέχει επικοινωνία δεδομένων από άκρο σε άκρο, καθορίζοντας τον τρόπο με τον οποίο τα δεδομένα πρέπει να πακετοποιούνται, να αντιμετωπίζονται, να διαβιβάζονται, να δρομολογούνται και να λαμβάνονται δεδομένα. Αυτή η λειτουργικότητα είναι οργανωμένη σε τέσσερα επίπεδα αφαίρεσης, τα οποία ταξινομούν όλα τα σχετικά πρωτόκολλα ανάλογα με το πεδίο της σχετικής δικτύωσης. Από το χαμηλότερο στο υψηλότερο, τα στρώματα είναι το στρώμα συνδέσμου, που περιέχει μεθόδους επικοινωνίας για δεδομένα που παραμένουν σε ένα μόνο τμήμα δικτύου (σύνδεσμος). το στρώμα Διαδικτύου, που παρέχει δικτύωση στο Διαδίκτυο μεταξύ ανεξάρτητων δικτύων. το στρώμα μεταφοράς, το οποίο χειρίζεται επικοινωνία μεταξύ κεντρικού υπολογιστή και κεντρικού υπολογιστή. και το επίπεδο εφαρμογής, παρέχοντας την ανταλλαγή δεδομένων μεταξύ των διεργασιών για εφαρμογές.

2.5 Σειριοποίηση

Η σειριοποίηση είναι η διαδικασία μετάφρασης των δομών δεδομένων ή της κατάστασης αντικειμένων σε μορφή που μπορεί να αποθηκευτεί (για παράδειγμα, σε ένα αρχείο ή μνήμη προσωρινής μνήμης) ή να μεταδοθεί (για παράδειγμα μέσω ενός συνδέσμου σύνδεσης δικτύου) και να ανακατασκευαστεί αργότερα (ενδεχομένως σε διαφορετικό υπολογιστικό περιβάλλον). Όταν η προκύπτουσα σειρά δυαδικών ψηφίων επανεξετάζεται σύμφωνα με τη μορφή σειριοποίησης, μπορεί να χρησιμοποιηθεί για τη δημιουργία ενός σημασιολογικά πανομοιότυπου κλώνου του αρχικού αντικειμένου. Για πολλά σύνθετα αντικείμενα, όπως αυτά που κάνουν εκτεταμένη χρήση αναφορών, αυτή η διαδικασία δεν είναι απλή. Η σειριοποίηση των αντικειμενοστραφικών αντικειμένων δεν περιλαμβάνει καμία από τις σχετικές μεθόδους με τις οποίες είχαν προηγουμένως συνδεθεί.

Στο έργο μας χρησιμοποιούμε τεχνικές σειριοποίησης για να μεταδώσουμε δομημένα δεδομένα

μεταξύ των επικοινωνούντων στο σύστημα μας. Συγκεκριμένα, μεταδίδουμε μηνύματα μέσω της στοίβας δικτύου μεταξύ απομακρυσμένων κόμβων. Τα μηνύματα αυτά αποτελούνται από τιμές διάσπαρτης μνήμης. Έτσι, με την εφαρμογή τεχνικών σειριοποίησης, μπορούμε να πάρουμε μια δομή δεδομένων μνήμης που αποτελείται από περιοχές διάσπαρτης μνήμης σε ένα ρεύμα bytes που μπορεί να μεταδοθεί μέσω του δικτύου και στη συνέχεια να ανακατασκευαστεί με τον ίδιο τρόπο ώστε να αντιπροσωπεύει το ίδιο δομή δεδομένων.

2.5.1 Protocol Buffers

Για τη δουλειά μας, χρησιμοποιήσαμε τους Protocol Buffers [35] της Google ως σειριοποιητή για το πρόγραμμά μας. Οι Protocol Buffers είναι ανεξάρτητοι από γλώσσα, πλατφόρμα και επεκτάσιμος μηχανισμός για τη σειριοποίηση δομημένων δεδομένων. Οι Protocol Buffers περιλαμβάνουν μια γλώσσα περιγραφής διεπαφής που περιγράφει τη δομή των δεδομένων που πρόκειται να συμμετάσχουν σε μια διαδικασία serialization, μαζί με ένα πρόγραμμα που παράγει πηγαίο κώδικα από αυτήν την περιγραφή που προορίζεται για τη δημιουργία ή την ανάλυση ροής bytes που αντιπροσωπεύουν τα δομημένα δεδομένα. Τα πρωτόκολλα Buffer του Google έχουν εκτεταμένη συμβατότητα με πολλές γλώσσες προγραμματισμού και δημοσιεύονται με άδεια ανοιχτού κώδικα. Οι στόχοι σχεδίασης για Buffer Protocol υπογράμμισαν την απλότητα και την απόδοση. Συγκεκριμένα, σχεδιάστηκαν για να είναι μικρότερη και ταχύτερη από την XML.

Chapter 3

Σχεδιασμός και Υλοποίηση

Σχεδιάζουμε το RACEX να παρεμβάλλεται μέσα στο Manycore Platform Stack Software και να παρακολουθεί την επικοινωνία Transport Layer μεταξύ του κεντρικού επεξεργαστή και του επιταχυντή που υλοποιείται από το SCIF API. Το RACEX framework συλλαμβάνει, πακετάρει και προωθεί κλήσεις SCIF API μέσω μιας βιβλιοθήκης λογισμικού που συνδέεται με έναν δαίμονα διακομιστή ο οποίος είναι υπεύθυνος για την εξυπηρέτηση των αιτημάτων του πελάτη και την εκτέλεση των αρχικών κλήσεων scif στον coprocessor. Εφαρμόζουμε ένα μοντέλο κατανεμημένης αρχιτεκτονικής Client-Server. Εικονογραφήσαμε μια επισκόπηση υψηλού επιπέδου του σχεδιασμού του RACEX στο σχήμα 3.1. Η υποκείμενη αρχιτεκτονική του συστήματος, μαζί με τα μονοπάτια δεδομένων και ελέγχου, απεικονίζονται στο σχήμα 3.2. Τα βασικά συστατικά που αποτελούν το πλαίσιο RACEX είναι η βιβλιοθήκη client-side "libracex" που περιβάλλει το SCIF API, ώστε να το κάνει διαθέσιμο απο απόσταση, και το δαίμονα RACEX από την πλευρά του διακομιστή, το οποίο εκθέτει ένα απομακρυσμένο API αιτήσεων εκτέλεσης SCIF. Το μηχάνημα που φιλοξενεί το Xeon Phi είναι φυσικά συνδεδεμένο και λειτουργεί ως διακομιστής στο πλαίσιο μας.

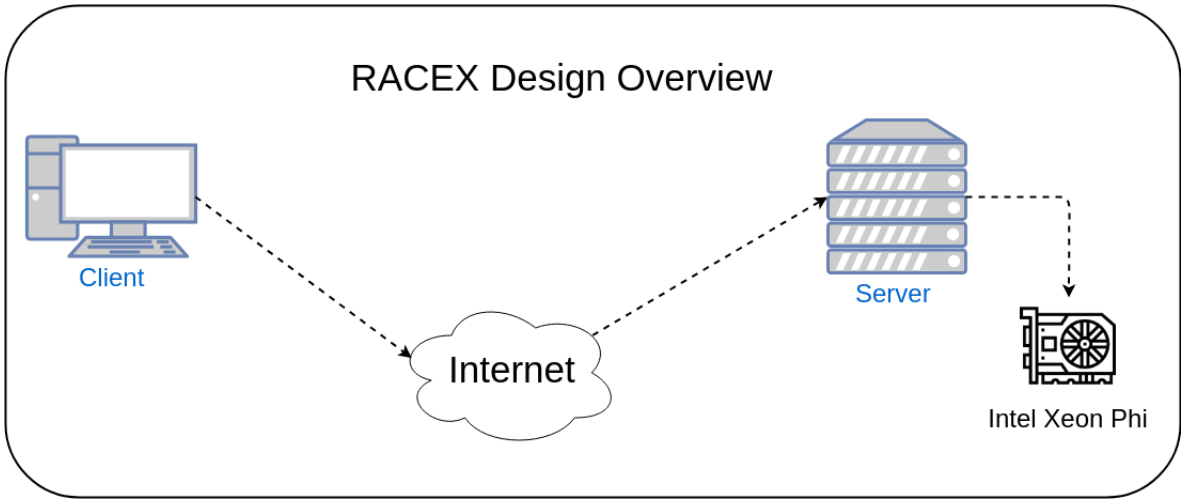


Figure 3.1: RACEX Design Overview

Παρουσιάζουμε στο διάγραμμα 3.2 τη διαδρομή I/O του πλαισίου μας όταν ένα αίτημα SCIF API καλείτε από μια εφαρμογή. Αντιπροσωπεύουμε τη διαδρομή εισόδου/εξόδου για ένα σενάριο που

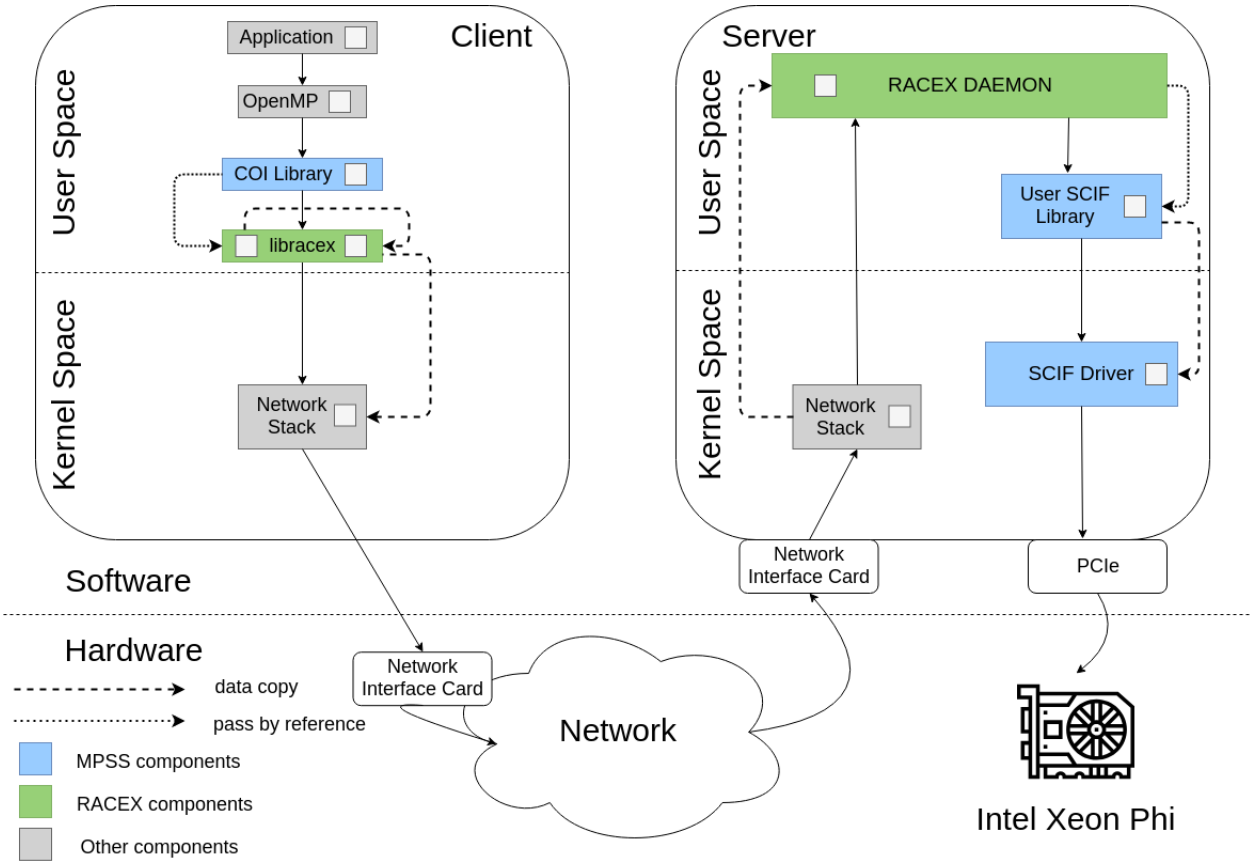


Figure 3.2: Architecture of the RACEX Framework (Data and Control Path)

αντιστοιχεί σε μια κατάσταση εκτέλεσης εκφόρτωσης από τους προαναφερθέντες τρόπους εκτέλεσης του Xeon Phi. Οι συμπαγείς γραμμές αντιπροσωπεύουν τη διαδρομή ελέγχου και οι διακεκομμένες γραμμές αντιπροσωπεύουν τις διαδρομές δεδομένων. Ο πελάτης επικοινωνεί με τον δαίμονα πλευράς διακομιστή χρησιμοποιώντας το Sockets BSD API και τη στοίβα TCP/IP που αναπαρίσταται στο σχήμα 3.2 ως "Network Stack". Παρέχουμε περαιτέρω λεπτομέρειες σχετικά με το σχεδιασμό της κατανεμημένης αρχιτεκτονικής μας στις παρακάτω ενότητες.

3.1 Αρχιτεκτονική Βιβλιοθήκης Πελάτη

Το τμήμα πελάτη του πλαισίου middleware που προτείνουμε αποτελείται από μια βιβλιοθήκη που είναι εγκατεστημένη στο σύστημα που ζητά απομακρυσμένη πρόσβαση σε πόρους του επιταχυντή. Η βιβλιοθήκη πελάτη προσφέρει την ίδια διεπαφή προγραμματισμού εφαρμογών (API) όπως και η αρχική βιβλιοθήκη Intel SCIF και είναι δυαδική συμβατή με αυτήν. Ως εκ τούτου, μπορούμε να προσφέρουμε συμβατότητα με το υπόλοιπο λογισμικό Multicore Platform Software Stack και παράλληλα API προγραμματισμού όπως το OpenMP.

Η βιβλιοθήκη από την πλευρά του πελάτη παρακολουθεί τις κλήσεις της βιβλιοθήκης SCIF είτε προφορτώνοντας τη βιβλιοθήκη στον δυναμικό linker του συστήματος του πελάτη πριν από την προβλεπόμενη εκτέλεση είτε με την εγκατάσταση της βιβλιοθήκης στο σύστημα και την αντικατάσταση

οποιασδήποτε προϋπάρχουσας βιβλιοθήκης `scif`. Το πλαίσιο RACEX βασίζεται σε μια επικοινωνία επιπέδου νήματος μεταξύ του πελάτη και του διακομιστή. Κάθε αντίστοιχο νήμα μιας διεργασίας που εκκινεί μια κλήση SCIF, ξεκινά μια αντίστοιχη σύνδεση TCP με ένα αποκλειστικό νήμα στο δαίμονα της πλευράς του διακομιστή που θα εξυπηρετήσει τις εισερχόμενες αιτήσεις από τον πελάτη και θα επιστρέψει τα προκύπτοντα δεδομένα. Η ροή εκτέλεσης της βιβλιοθήκης πλευράς πελάτη συνοψίζεται από τον αλγόριθμο 1.

Algorithm 1 Client-Side Library Execution Flow

```
1:  procedure SCIF_CALL()
2:      curr_thread ← identify_current_thread()
3:      if !has_established_connection(curr_thread) then
4:          establish_connection()
5:      if is_rma_operation() then
6:          mapping ← get_mapping()
7:      cmd ← pack_phi_cmd(args)
8:      send_phi_cmd(cmd)
9:      res ← recv_phi_cmd_results()
10:     if res− > success() then
11:         set_resulting_values(mapping)
12:     else
13:         set_error_values()
14:     return
```

Για κάθε κλήση SCIF API που γίνεται στο σύστημα του πελάτη, η ροή εκτέλεσης του πλαισίου μας εκτελεί τα ακόλουθα καθήκοντα: (1) να αποκτήσει τον χειριστή σύνδεσης στον δαίμονα του διακομιστή για το νήμα που καλεί, σε περίπτωση που δεν υπάρχει ανοικτή σύνδεση με τον διακομιστή, (3) στέλνουμε το αίτημα κλήσης SCIF στον δαίμονα του διακομιστή (4) περιμένουμε για το μήνυμα των αποτελεσμάτων (2) πακετάρουμε] όλα τα απαιτούμενες παραμέτρους μαζί με το αναγνωριστικό λειτουργίας στο πρωτόκολλο επικοινωνίας του RACEX από το διακομιστή (5) αποσυμπιέζουμε τα αποτελέσματα και αντιγράφουμε τα απαιτούμενα δεδομένα στις διευθύνσεις μνήμης που έχουν καθοριστεί (6) σε περίπτωση σφάλματος στην απομακρυσμένη εκτέλεση της κλήσης SCIF, ορίζετε η κατάλληλη τιμή σφάλματος (7) επιστρέψτε την κατάλληλη τιμή στο καλούντα.

3.2 Αρχιτεκτονική Δαίμονα Διακομηστή

Ο δαίμονας του διακομιστή του πλαισίου μας βρίσκεται στον διακομιστή που είναι φυσικά συνδεδεμένος με τον Coprocessor Intel Xeon Phi και εξυπηρετεί πιθανούς πελάτες RACEX, επιτρέποντας ταυτόχρονη πρόσβαση στους πόρους του επιταχυντή. Για κάθε εισερχόμενη σύνδεση, ο δαίμονας διακομιστή θα δημιουργήσει ένα ειδικό νήμα που θα εξυπηρετεί όλα τα αιτήματα για απομακρυσμένες αιτήσεις κλήσεων SCIF API. Προκειμένου να αποφευχθούν τα προβλήματα ταυτοχρονισμού και επίσης να έχουν ξεκάθαρα διαχωρισμένα πλαίσια εκτέλεσης πελάτη-εξυπηρετητή, κάθε νήμα που καλεί στην πλευρά του πελάτη έχει ένα ειδικό νήμα εξυπηρέτησης στην πλευρά του διακομιστή. Με τον τρόπο αυτό, η ίδια σύνδεση TCP/IP χρησιμοποιείται μόνο από τα δύο νήματα που επικοινωνούν,

ο πελάτης που εκδίδει το αίτημα για απομακρυσμένη κλήση SCIF API και το νήμα διακομιστή που θα εξυπηρετήσει το αίτημα του πελάτη. Η ροή εκτέλεσης του δαίμονα του διακομιστή συνοψίζεται στον Αλγόριθμο 2.

Algorithm 2 RACEX Server Daemon Execution Flow

```
1:  procedure RACEX_DAEMON()
2:      wait_for_incoming_connection()
3:      spawn_serving_thread()
4:  serve_client:
5:      cmd ← receive_phi_cmd()
6:      unpack_phi_cmd(cmd)
7:      if is_rma_operation() then
8:          mapping ← retrieve_mapping()
9:          process_phi_cmd(cmd)
10:     execute_phi_cmd(args)
11:     results ← pack_phi_results()
12:     send_phi_cmd(results)
13:     res ← recv_phi_cmd_results()
14:     if client_finished() then
15:         return
16:     else
17:         goto serve_client
18:     return
```

Για κάθε μήνυμα απομακρυσμένης κλήσης RACEX που λαμβάνει το νήμα εξυπηρέτησης του δαίμονα, η ροή εκτέλεσης του πλαισίου μας εκτελεί τις ακόλουθες εργασίες: (1) αποσυσκευασία του απομακρυσμένου αναγνωριστικού κλήσης API και των σχετικών παραμέτρων (2) προαιρετικά εκτελεί μια λειτουργία χαρτογράφησης μνήμης σε περίπτωση απομακρυσμένης `scif_register()` για να διατηρήσετε μια περιοχή μνήμης για επακόλουθες λειτουργίες RMA ή για να αντιγράψετε δεδομένα στην καθορισμένη περιοχή μνήμης για πρόσβαση μέσω των λειτουργιών RMA. (3) εκτελεί την πραγματική κλήση API SCIF (4) πακετάρει τα δεδομένα εξόδου προς μεταφορά στον πελάτη (5) στέλνει την απάντηση στον καλούντα και (6) μπλοκάρει μέχρι να προκύψει νέο μήνυμα από το νήμα που καλεί.

3.3 Πρωτόκολλο Επικοινωνίας RACEX

Έχουμε εφαρμόσει ένα προσαρμοσμένο πρωτόκολλο επικοινωνίας κλήσης API που χρησιμοποιείται στο RACEX από τα δύο κατανεμημένα μέρη επικοινωνίας, τον πελάτη και το server daemon. Το πρωτόκολλο χρησιμοποιεί τη τεχνολογία Protocol Buffers της Google [35] ως μια γρήγορη και ελαφριά μέθοδο serialization για τη σειριοποίηση δομημένων δεδομένων. Ως αποτέλεσμα, όπως παρουσιάζεται στη διαδρομή δεδομένων στο Σχήμα 3.2 τόσο για τη βιβλιοθήκη RACEX από την πλευρά του πελάτη όσο και για τον δαίμονα της πλευράς διακομιστή, αντιμετωπίζουμε data copy λειτουργία που είναι εγγενές στη διαδικασία σειριοποίησης της τεχνολογίας Protocol Buffers. Στη

μελλοντική δουλειά μας, σχεδιάζουμε να αφαιρέσουμε την επιπλέον καθυστέρηση που παρουσιάζεται απο τις λειουργίες data copy που χρησιμοποιούν οι Protocol Buffers κατα τη σειριοποίηση. Αντ 'αυτού, σκοπεύουμε να χρησιμοποιήσουμε τη χρήση των διανυσμάτων διασποράς / συλλογής (scatter / gather mechanism) μηνυμάτων για τη μετάδοση των απαιτούμενων δομημένων δεδομένων μεταξύ πελάτη και διακομιστή και έτσι εξαλείφουμε την ανάγκη για σειριοποίηση και ατνιγραφή δεδομένων. Στην επόμενη ενότητα αξιολόγησης θα δούμε ότι η καθυστέρηση επιβάρυνσης που εμφανίζεται από τα προαναφερθέντα αντίγραφα δεδομένων είναι ασήμαντη σε σύγκριση με την τρέχουσα καθυστέρηση του δικτύου. Επιπλέον, το πρωτόκολλο επικοινωνίας αναπτύχθηκε για να ικανοποιήσει την ανάγκη από το πλαίσιο μας για δομημένη ανταλλαγή δεδομένων μεταξύ του πελάτη, ο οποίος ζητά πρόσβαση στους πόρους του Xeon Phi και ο δαίμονας του διακομιστή που εξυπηρετεί το αίτημα του πελάτη για απομακρυσμένη εκτέλεση των κλήσεων SCIF API. Ωστόσο, το πρωτόκολλο έχει σχεδιαστεί για να είναι γενικό, ικανό να υποστηρίζει τη δομημένη επικοινωνία που απαιτείται για την ενεργοποίηση ενός απομακρυσμένου πλαισίου εκτέλεσης API και, συνεπώς, καθιστώντας το ανεξάρτητο από την εφαρμογή, συμβάλλοντας σε ένα γενικό πλαίσιο για την Απομακρυσμένη Εκτέλεση Επιταχυντή.

3.4 Λειτουργίες Απομακρυσμένης Πρόσβασης Μνήμης

Ιδιαίτερη προσοχή λαμβάνεται για τις λειτουργίες πρόσβασης σε απομακρυσμένη μνήμη στις οποίες συμμετέχουν οι ακόλουθες λειτουργίες SCIF: `scif_register()`, `scif_unregister()`, `scif_writeto()`, `scif_readfrom()`, `scif_vwriteto()`, `scif_vreadfrom()`. Προκειμένου τα επικοινωνούντα μέρη SCIF να είναι σε θέση να εκτελέσουν μια λειτουργία RMA, πρέπει πρώτα να καταχωρήσουν την κατάλληλη περιοχή μνήμης στην οποία θα γίνεται πρόσβαση κατά τη διάρκεια της λειτουργίας RMA. Όπως έχουμε δει στην ενότητα Θεωρητικού Υποβάθρου, η κλήση API του `scif_register()` έχει το ρόλο να ανοίξει ένα παράθυρο, ένα εύρος σελίδων στο καταχωρημένο χώρο διευθύνσεων (RAS), που κυμαίνεται για το καθορισμένο μήκος και να επιστρέφει μια διεύθυνση offset, από την αρχή του καταχωρημένου χώρου διευθύνσεων, στον καλούντα που θα χρησιμοποιηθεί από τις λειτουργίες API που εμπλέκονται σε λειτουργίες RMA προκειμένου να προσδιοριστεί ο καταχωρημένος χώρος που προορίζεται για διμερή πρόσβαση. Το πρόγραμμα οδήγησης SCIF διατηρεί μια χαρτογράφηση του καταγεγραμμένου παραθύρου στο χώρο διευθύνσεων καταχωρητή (RAS) και το εύρος μνήμης στον εικονικό χώρο διεύθυνσης (VAS) που αντιστοιχεί σε αυτόν, όπως παρουσιάζεται στο Σχήμα **??**.

Λόγω της φύσης της κατανεμημένης αρχιτεκτονικής του πλαισίου, εφαρμόσαμε έναν μηχανισμό που αποθηκεύει και ανακτά τις αντιστοιχίσεις μεταξύ μιας περιοχήγς εγγεγραμμένου χώρου διευθύνσεων που δείχνει σε ένα καταχωρημένο παράθυρο και την αντίστοιχη εικονική περιοχή μνήμης χώρου διευθύνσεων. Κάθε χαρτογράφηση είναι μοναδική για κάθε περιβάλλον διεργασίας. Κατά τη διάρκεια μιας λειτουργίας RMA, τόσο η βιβλιοθήκη πελάτη όσο και ο δαίμονας της πλευράς διακομιστή θα ανακτήσουν από την αντίστοιχη δομή δεδομένων τους αντιστοίχιση, τον δείκτη στην περιοχή μνήμης χώρου εικονικής διεύθυνσης που αντιστοιχεί στην διευθυνση offset του καταχωρημένου χώρου διευθύνσεων (RAS) Η λειτουργία RMA χρησιμοποιείται για τη μεταφορά των δεδομένων από και προς τον πελάτη πριν και μετά την ολοκλήρωση της λειτουργίας RMA αντίστοιχα. Ο κατανεμημένος μηχανισμός που χρησιμοποιούμε για την υποστήριξη της λειτουργικότητας SCIF RMA απεικονίζεται στο σχήμα 3.3.

Το RACEX υποστηρίζει επί του παρόντος όλες τις λειτουργίες RMA σε λειτουργία σύγχρονης εκτέλεσης, με όλες τις λειτουργίες SCIF που σχετίζονται με RMA να επιστρέφουν μετά την ολοκλήρωση της λειτουργίας μεταφοράς.

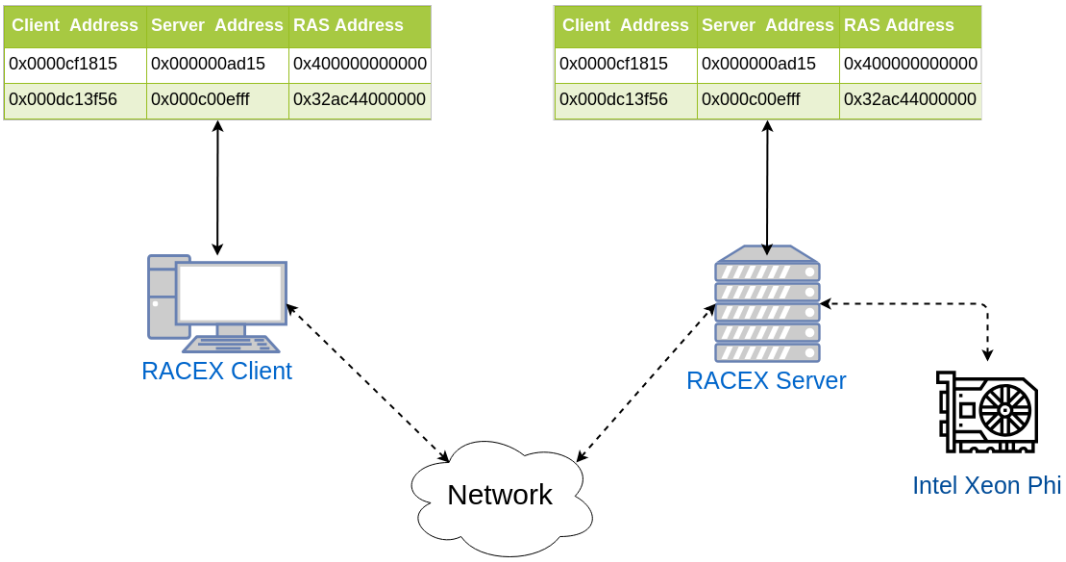


Figure 3.3: RACEX RMA Mappings Mechanism

Chapter 4

Αξιολόγηση

Σε αυτή την ενότητα αναλύουμε την απόδοση του πλαισίου μας. Πρώτον, αξιολογούμε τον χρόνο εκτέλεσης του πλαισίου μας στο επίπεδο του API, όπου το συγκρίνουμε με τη φυσική εκτέλεση. Αξιολογούμε το πλαίσιο RACEX διεξάγοντας μια σειρά microbenchmarks που επικεντρώνονται σε συγκεκριμένες κλήσεις API. Στη συνέχεια, διερευνάμε την απόδοση του πλαισίου RACEX στη λειτουργία native και offlaod του επιταχυντή Intel Xeon Phi, εκτελώντας διάφορα benchmarks από την οικογένεια Rodinia Heterogeneous Benchmark Suite [36]. Τέλος, ολοκληρώνουμε με μια δοκιμασία κλιμάκωσης όπου αξιολογούμε πώς το σύστημα μας κλιμακώνει σε απόδοση με τη φυσική περίπτωση, για διάφορα νήματα και αριθμό απομακρυσμένων πελατών.

4.1 Πειραματική Ρύθμιση

Αξιολογούμε την απόδοση του πλαισίου μας ρυθμίζοντας την ακόλουθη τοπολογία: ένα μηχάνημα host server με 1x Intel Core i7-4820K, 32GB μνήμη RAM, εξοπλισμένο επίσης με έναν επιταχυντή Intel Xeon Phi 3120P με 57 πυρήνες με hyperthreading λειτουργικότητα, με 4 νήματα ανά πυρήνα, συνοψίζοντας συνολικά 228 νήματα, στο οποίο host μηχάνημα θα τρέχει ο δαίμονας διακομιστή του RACEX και ένα άλλο μηχάνημα με 1x Intel Core i5-2320, 4GB μνήμη RAM που θα λειτουργεί ως πελάτης στα σενάρια αξιολόγησης. Τα δύο μηχανήματα συνδέονται μέσω Ethernet στο ίδιο δίκτυο LAN. Η προαναφερθείσα τοπολογία απεικονίζεται στο Σχήμα 4.1.

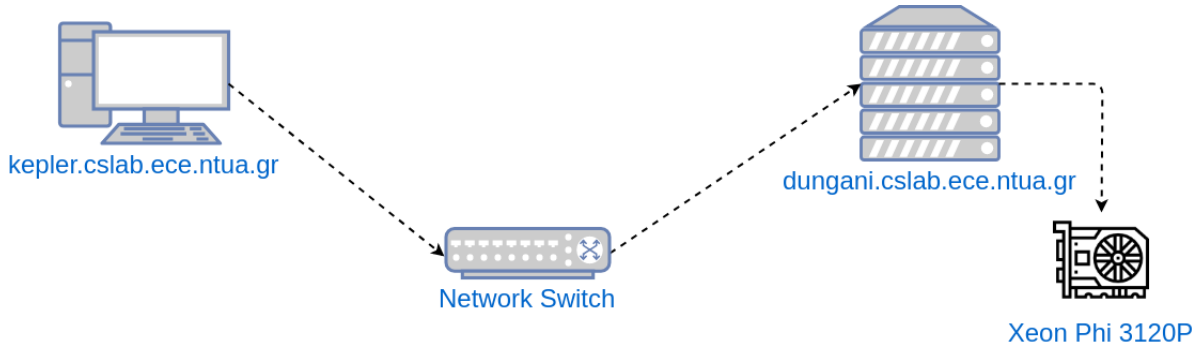


Figure 4.1: RACEX Experimental Setup Topology Overview

Προκειμένου να αξιολογήσουμε την απόδοση του πλαισίου μας σε σύγκριση με την εγγενή λειτουργία του Xeon Phi, διεξάγουμε τα σενάρια δοκιμών για τις παρακάτω διαμορφώσεις δικτύου:

1. Local: Σε αυτήν τη διαμόρφωση, ο πελάτης που χρησιμοποιεί το πλαίσιο RACEX για να αποκτήσει πρόσβαση στους πόρους του Xeon Phi είναι το ίδιο μηχάνημα που τρέχει την εφαρμογή δαίμονα και διαθέτει το Xeon Phi. Σε αυτή τη διαμόρφωση προσομοιώνουμε την ελάχιστη καθυστέρηση δικτύου που προκύπτει για το σύστημα μας.

2. Remote: Σε αυτήν τη διαμόρφωση, ο πελάτης που χρησιμοποιεί το πλαίσιο RACEX για να αποκτήσει πρόσβαση στους πόρους του Xeon Phi βρίσκεται στο ίδιο LAN με το μηχάνημα διακομιστή. Σε αυτή τη διαμόρφωση προσομοιώνουμε την απόδοση του πλαισίου μας σε ένα ρεαλιστικό περιβάλλον.

4.2 Microbenchmarks

Χρησιμοποιούμε ένα απλό σετ microbenchmarks, που αναπτύχθηκε από τους συγγραφείς του vPHI [16], που εστιάζουν στην αξιολόγηση της απόδοσης του πλαισίου μας κατά τις δύο σημαντικές λειτουργίες επικοινωνίας που χρησιμοποιεί η SCIF για να επικοινωνήσει ο επεξεργαστής με τον επιταχυντή: (1) οι κλήσεις socket-like Send-Recv API που περιλαμβάνουν το στρώμα μηνυμάτων του SCIF, στο οποίο μπορούν να ανταλλάσσονται σύντομα μηνύματα τύπου εντολής μεταξύ του κεντρικού υπολογιστή και του συνεπεξεργαστή και (2) οι κλήσεις SCI RMA API απομακρυσμένης μνήμης που χρησιμοποιούνται για βαριά μαζικές μεταφορές δεδομένων μεταξύ του κεντρικού επεξεργαστή και του επιταχυντή [34]. Και για τα δύο σενάρια αναφοράς, τα εκτελούμε αρχικά στο διακομιστή προκειμένου να αποκτήσουμε μια βασική γραμμή στην οποία θα συγκρίνουμε την απόδοση του πλαισίου μας στις δύο διαμορφώσεις δικτύου. Επίσης, για τη εκτέλεση των benchmarks, δημιουργείτε διεργασία στον επιταχυντή η οποία θα έχει το ρόλο να επικοινωνεί με την εφαρμογή που θα τρέχει στο πελάτη και στη μια περίπτωση θα λαμβάνει δεδομένα μέσω της κλήσης API `scif_recv()` και στην άλλη περίπτωση θα διαβάζει απο κάποια κατοχυρωμένη μνήμη που συμμετέχει σε RMA λειτουργία.

4.2.1 Αξιολόγηση Messaging Layer

Αρχικά, αξιολογούμε την καθυστέρηση του πλαισίου μας εκτελώντας το Send-Recv microbenchmark για διαφορετικά μεγέθη μηνυμάτων που κυμαίνονται από 1 Byte έως 32K Bytes. Σε αυτό το σενάριο, η εφαρμογή που δημιουργείτε στον επιταχυντή ακούει για εισερχόμενη σύνδεση, δέχεται μια σύνδεση από έναν δυνητικό πελάτη και λαμβάνει έναν προκαθορισμένο αριθμό bytes χρησιμοποιώντας την κλήση API `scif_recv()`. Στην πλευρά πελάτη-κεντρικού υπολογιστή, η εφαρμογή συνδέεται με τη διεργασία που εκτελείται στον συνεπεξεργαστή και στέλνει τον προκαθορισμένο αριθμό bytes χρησιμοποιώντας την κλήση API `scif_send()`. Παρουσιάζουμε την αντίστοιχη καθυστέρηση που μετρήθηκε για το πλαίσιο RACEX, για τις δύο διαμορφώσεις δικτύου, σε σύγκριση με την απόδοση της φυσικής εκτέλεσης στο σχήμα 4.2.

Στο Σχήμα 4.2 παρατηρούμε ότι η καθυστέρηση του πλαισίου μας δείχνει μια σχεδόν σταθερή επιβάρυνση για μεγέθη μηνυμάτων μικρότερα από 4K. Η καθυστέρηση εκτέλεσης στη φυσική εκτέλεση για την αποστολή ενός 1 Byte από τον κεντρικό υπολογιστή στον coprocessor είναι 5 us. Η καθυστέρηση του πλαισίου RACEX είναι 95 us για τη διαμόρφωση του τοπικού δικτύου και 250 για τη διαμόρφωση του απομακρυσμένου δικτύου. Επομένως, η γενική καθυστέρηση του πλαισίου μας

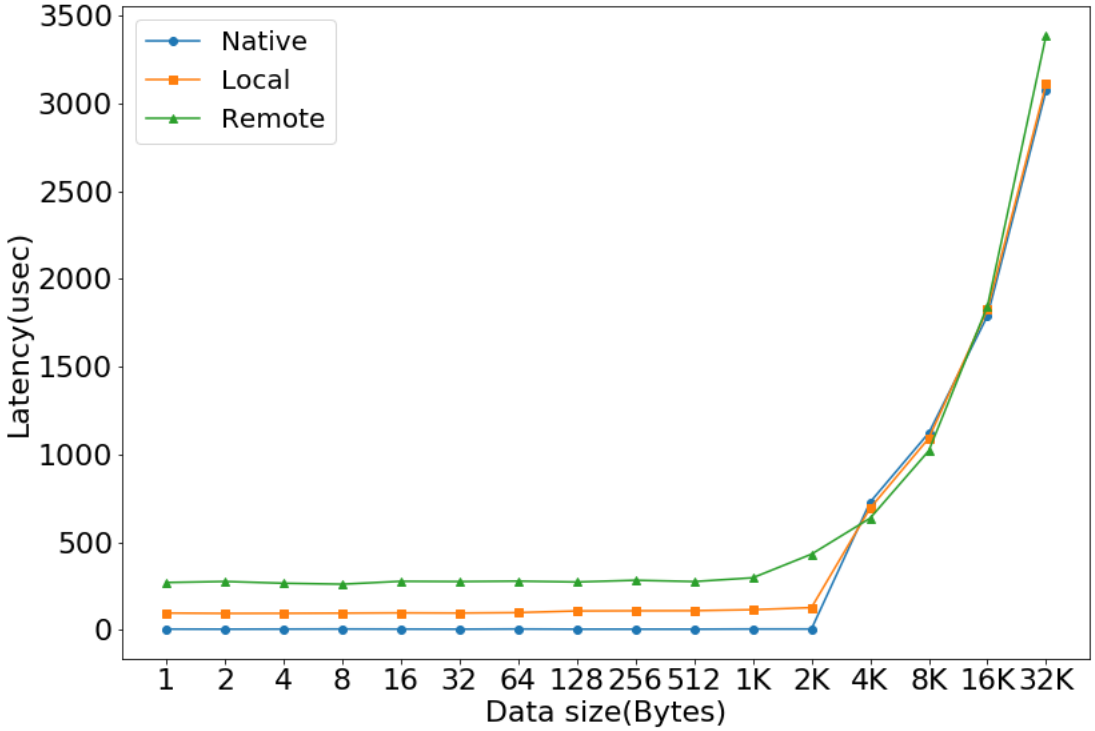


Figure 4.2: Send-Recv Communication Latency

υπολογίζεται σε $Localhost\ _{O}verhead = (95 - 5) = 90us, Remote\ _{O}verhead = (250 - 5) = 245us$. Από τις παρουσιαζόμενες μετρήσεις, παρατηρούμε σημαντική αύξηση της μετρημένης καθυστέρησης από το Localhost στη διαμόρφωση του απομακρυσμένου δικτύου. Η αξιοσημείωτη διαφορά μεταξύ των δύο διαμορφώσεων δικτύου μας οδηγεί στο συμπέρασμα ότι τα $245us - 90us = 155us$ αποδίδονται στη καθυστέρηση μετάδοσης μέσω το δικτύου.

Επιπλέον, παρατηρούμε μια απότομη κλιμάκωση της μετρημένης καθυστέρησης για τη φυσική εκτέλεση για μηνύματα μεγαλύτερα από 4KB. Τα ευρήματα είναι σύμφωνα με τις προδιαγραφές SCIF που δηλώνουν ότι για μηνύματα μεγέθους μεγαλύτερου από 4KB, οι λειτουργίες SCIF RMA θα πρέπει να χρησιμοποιούνται για τη μεταφορά δεδομένων, καθώς το SCIF Messaging Layer που αποτελούν μέρος των κλήσεων API Send-Recv προορίζεται για ανταλλαγή μικρών μηνυμάτων τύπου εντολών μεταξύ του επεξεργαστή και του επιταχυντή [34]. Παράλληλα με την κλιμάκωση της γραμμής φυσικής εκτέλεσης, παρατηρούμε ότι η καθυστέρηση του πλαισίου RACEX ακολουθεί μια παρόμοια κλιμάκωση όπου βιώνουμε μια near-native απόδοση στο πλαίσιο μας για μηνύματα 4K έως και 32K.

Προκειμένου να κατανοήσουμε καλύτερα τα τη καθυστέρηση που εισάγει το σύστημα μας για το επίπεδο ανταλλαγής μηνυμάτων Messaging Layer, κάνουμε μια σε βάθος ανάλυση. Τα ευρήματα της ανάλυσης παρουσιάζονται στο Σχήμα 4.3.

Από την ανάλυση καταλήγουμε στο συμπέρασμα ότι περίπου το 7% της συνολικής καθυστέρησης εκτέλεσης που μετράται για το πλαίσιο μας για μηνύματα μικρότερα από 4K αποδίδεται στην υλοποίηση του πλαισίου και το 84% αποδίδεται στη καθυστέρηση μετάδοσης δικτύου και το TCP/IP Stack.

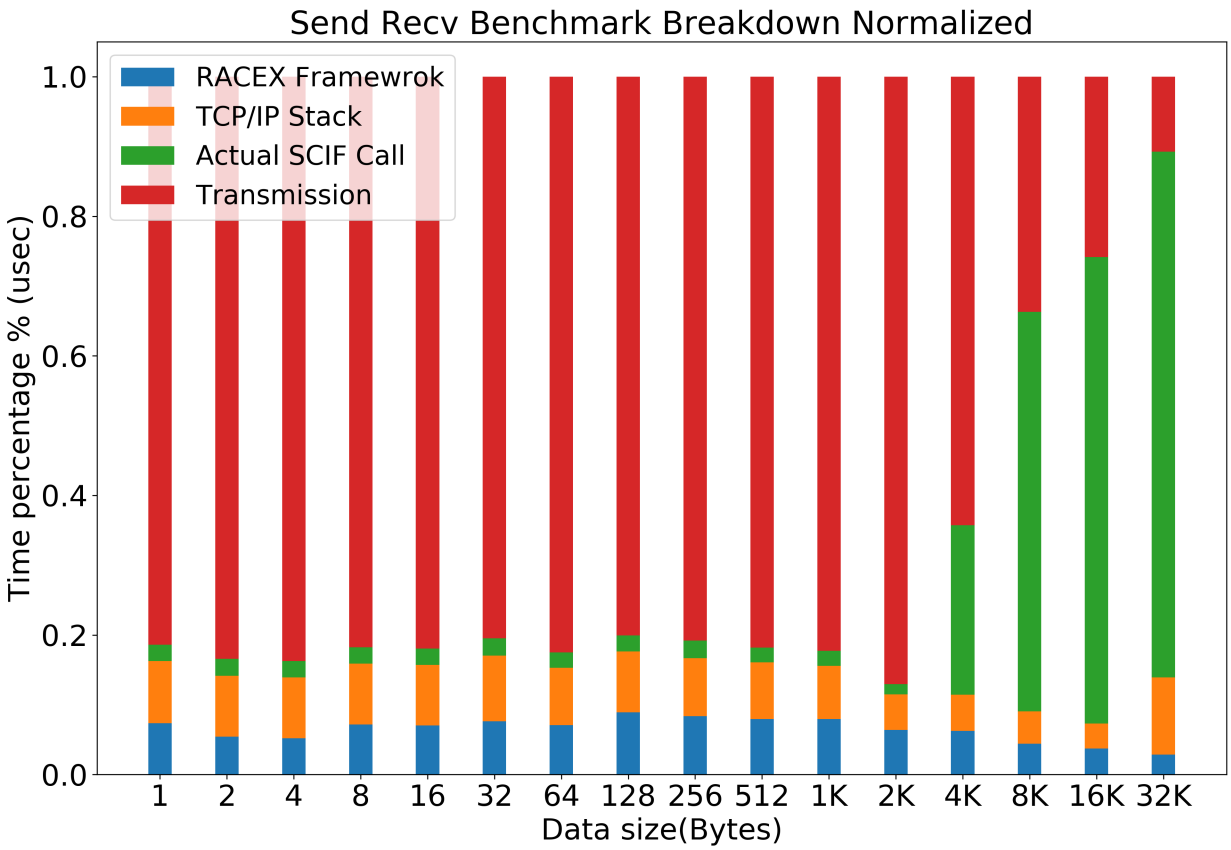


Figure 4.3: Send-Recv Communication Performance Breakdown

Η κύρια πηγή καθυστέρησης που παρουσιάζει η υλοποίηση του συστήματος μας αποδίδεται στις λειτουργίες σειριοποίησης και αποσειριοποίησης, οι οποίες, όπως εξηγήσαμε, περιέχουν λειτουργίες αντιγραφής δεδομένων και μπορούμε να μετριάσουμε στο μελλοντικό μας έργο χρησιμοποιώντας ένα μηχανισμό μεταφοράς διασποράς (scatter / gather). Επιπλέον, για μεγέθη μηνυμάτων που υπερβαίνουν το όριο 4K, είναι εμφανές ότι το μερίδιο του συνολικού χρόνου εκτέλεσης που αντιστοιχεί στον πραγματικό χρόνο εκτέλεσης κλήσεων SCIF API αυξάνεται, όπως αναμένεται, με τρόπο παρόμοιο με την φυσική εκτέλεση. Αξίζει να σημειωθεί ότι για μηνύματα 32K το ποσοστό του συνολικού χρόνου εκτέλεσης που αποδίδεται στην πραγματική κλήση SCIF API υπολογίζεται σε 80%, εξηγώντας έτσι την near-native απόδοση που παρατηρήσαμε για το πλαίσιο RACEX.

4.2.2 Αξιολόγηση Λειτουργιών RMA

Στη συνέχεια, αξιολογούμε την απόδοση του πλαισίου μας κατά τη διάρκεια λειτουργιών πρόσβασης απομακρυσμένης μνήμης, που χρησιμοποιούνται από το πρωτόκολλο SCIF για αποτελεσματικές μεταφορές μεγάλων δεδομένων, προκειμένου να προσδιοριστεί η απόδοση που μπορεί να επιτύχει το πλαίσιο RACEX. Σε αυτό το σενάριο, δημιουργείτε μια διεργασία στον επιταχυντή, η οποία ακούει για εισερχόμενες συνδέσεις, δέχεται έναν νέο πελάτη και λαμβάνει ένα αίτημα που καταγράφει μια περιοχή μνήμης που αποτελείται από έναν προκαθορισμένο αριθμό σελίδων, ο οποίος στη συνέχεια θα δεχτεί πρόσβαση από την απομακρυσμένη διεργασία στον κεντρικό υπολογιστή . Στη συνέχεια, η διαδικασία

πελάτη-κεντρικού υπολογιστή του benchmark καταγράφει ένα αντίστοιχο παράθυρο μνήμης του ίδιου προκαθορισμένου αριθμού σελίδων σε μέγεθος και εκκινεί μια λειτουργία πρόσβασης απομακρυσμένης μνήμης μέσω της κλήσης API `scif_writeto()`, κατά τη διάρκεια της οποίας ένας buffer που υπάρχει στο ο χώρος καταχωρημένων διευθύνσεων (RAS) του κεντρικού υπολογιστή μεταφέρεται στο χώρο καταχωρημένων διευθύνσεων (RAS) του συνεπεξεργαστή. Τα αποτελέσματα από την αξιολόγηση της απόδοσης για τις λειτουργίες RMA παρουσιάζονται στην Εικόνα 4.4.

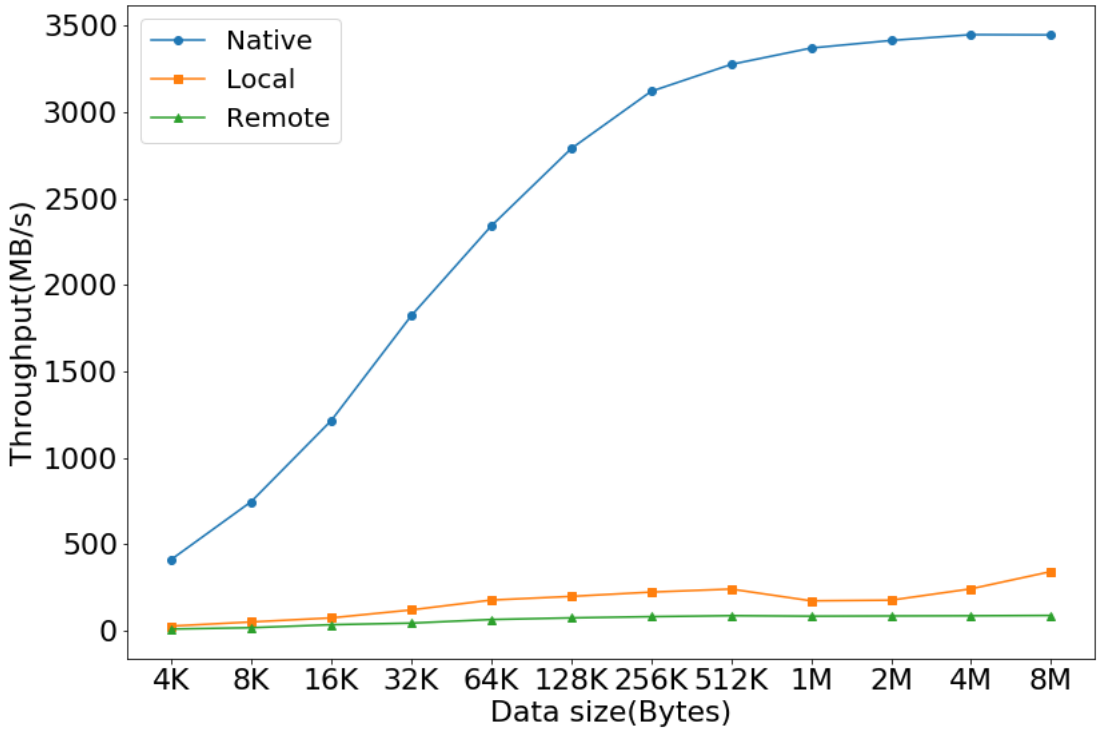


Figure 4.4: Read-Write RMA Operations Throughput

Η απόδοση της φυσικής εκτέλεσης SCIF ανέρχεται στα 1,72 GB/s, όπου το RACEX καταφέρνει να φτάσουν τα 42,7 MB/s. Προκειμένου να κατανοήσουμε καλύτερα το έλλειμμα του πλαισίου μας με τη μέτρηση της απόδοσης για τη λειτουργία RMA σε σύγκριση με τη φυσική εκτέλεση, πραγματοποιούμε μια σε βάθος ανάλυση απόδοσης, τα αποτελέσματα της οποίας παρουσιάζονται στο Σχήμα 4.5.

Από την ανάλυση που παρουσιάζεται, είναι προφανές ότι το μεγαλύτερο μερίδιο των καθυστερήσεων που υπάρχει στον χρόνο εκτέλεσης αποδίδεται στην καθυστέρηση μετάδοσης δικτύου και στο TCP/IP Stack. Ειδικότερα, καθώς το μέγεθος των δεδομένων που μεταφέρονται μέσω της λειτουργίας RMA αυξάνεται, το ποσοστό του συνολικού χρόνου εκτέλεσης που αντιπροσωπεύει την καθυστέρηση μετάδοσης δικτύου μειώνεται. Ταυτόχρονα, το ποσοστό που αντιπροσωπεύει το γενικό κόστος της στοίβας TCP/IP αυξάνεται σταθερά καθώς γίνεται το πιο σημαντικό μεταξύ των άλλων μεγάλων λειτουργιών εκτέλεσης. Από κοινού, η μετάδοση δικτύου και το TCP/IP Stack είναι υπεύθυνα για περίπου το **72%** της συνολικής καθυστέρησης χρόνου εκτέλεσης και οι λειτουργίες σειριοποίησης

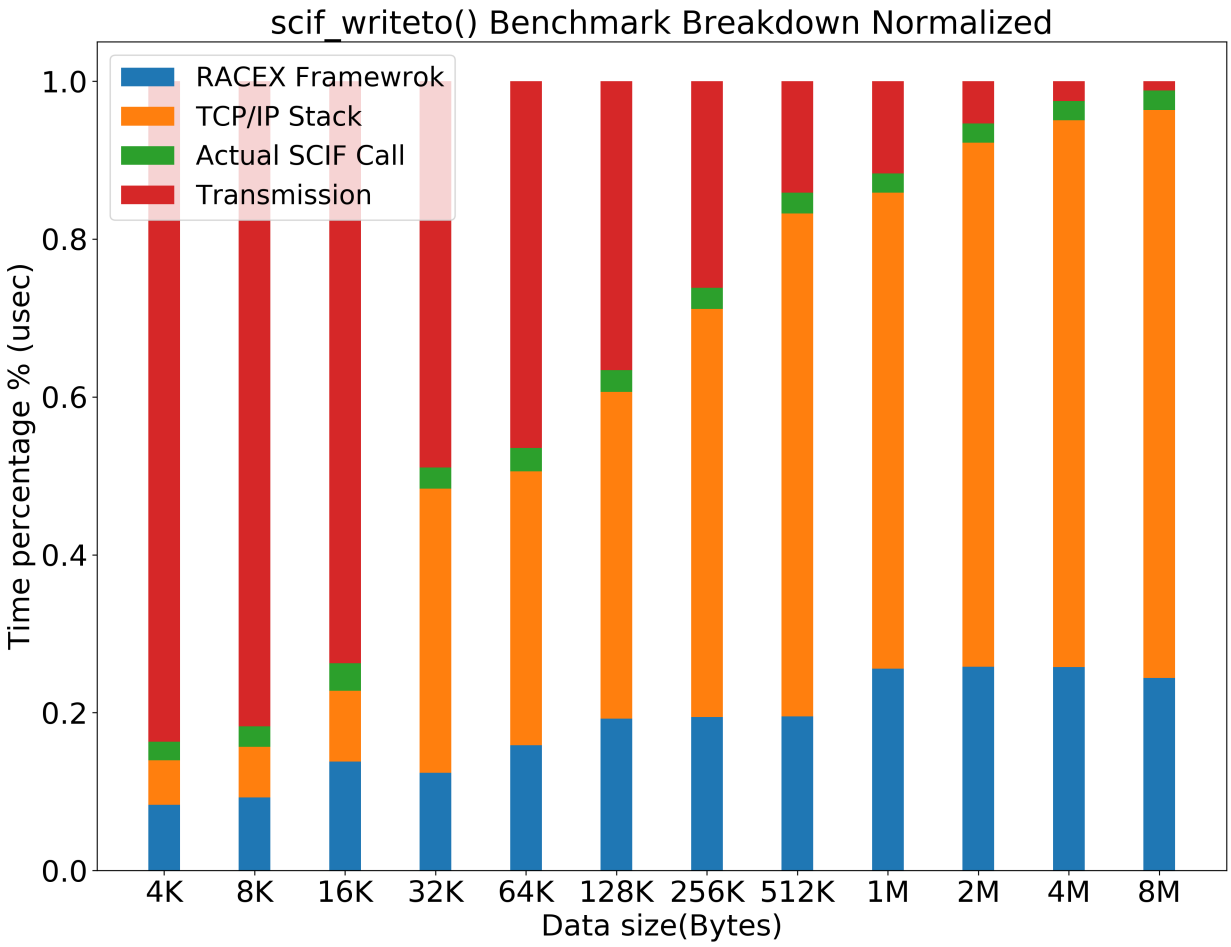


Figure 4.5: Read-Write RMA Operations Performance Breakdown

που χρησιμοποιούνται για τη μετάδοση του δομημένων δεδομένων για το **21%**, εξηγόντας έτσι το περιορισμένο throughput σε σχέση με τη φυσική εκτέλεση.

Ωστόσο, υπάρχει μια αντιληπτή παρεκκλίνουσα συμπεριφορά μεταξύ του Send-Recv benchmarks και εκείνου του RMA benchmark. Υπάρχει μια σειρά από αλληλεπικαλυπτόμενα μεγέθη δεδομένων εισόδου, από 4KB έως 32KB, τα οποία αξιολογήσαμε και με τους δύο μηχανισμούς επικοινωνίας SCIF. Όπως διαπιστώσαμε προηγουμένως, το Send-Recv benchmark του Messagin Layer παρουσιάζει near native απόδοση για μηνύματα μεγέθους από 4KB έως 32KB. Από την άλλη πλευρά, το RMA benchmark παρουσιάζει σημαντική επιβάρυνση σε σχέση με τη φυσική εκτέλεση. Προκειμένου να γίνει κατανοητή αυτή η αποκλίνουσα συμπεριφορά, εστιάζουμε στα αποτελέσματα της ανάλυσης για τα συγκεκριμένα μεγέθη δεδομένων εισόδου που επικαλύπτονται μεταξύ των δύο benchmark . Για το μέγεθος δεδομένων εισόδου 32KB, μπορούμε να παρατηρήσουμε ότι υπάρχει σημαντική διαφορά στον χρόνο εκτέλεσης των πραγματικών κλήσεων SCIF API μεταξύ των δύο benchmark. Στο Send-Recv Messaging Layer benchmark, ο χρόνος εκτέλεσης της κλήσης `scif_send()` είναι 3078 us όταν το ισοδύναμο RMA `scif_writeto()` είναι 18 us για το ίδιο μέγεθος δεδομένων που πρόκειται να μεταφερθεί. Σε αυτό το πλαίσιο, όπως επίσης παρατηρούμε από τα αναλυτικά στοιχεία της ανάλυσης που παρουσιάζονται παραπάνω, όταν ο πραγματικός χρόνος εκτέλεσης κλήσεων SCIF

κλιμακώνει Send-Recv benchmar, το ποσοστό της συνολικής καθυστέρησης εκτέλεσης που ανήκει στις καθυστερήσεις μεταφοράς δικτύου μειώνεται. Ταυτόχρονα, στο RMA benchmark, το ποσοστό του συνολικού χρόνου εκτέλεσης που ανήκει στην καθυστέρηση μετάδοσης δικτύου κατέχει την εξέχουσα θέση, επειδή ο πραγματικός χρόνος εκτέλεσης της κλήσης SCIF είναι μικρότερος σε αντίθεση με το Send-Recv. Αφού αξιολογήσαμε τις δύο διαφορετικές συμπεριφορές, καταλήγουμε στο συμπέρασμα ότι η καθυστέρηση του δικτύου είναι η κύρια αιτία για τη χαμηλή απόδοση RMA για μεγάλες μεταφορές δεδομένων.

4.3 Native Execution Mode

Προχωρούμε και αξιολογούμε την απόδοση του πλαισίου RACEX σε εφαρμογές υψηλότερου επιπέδου. Αξιολογούμε την απόδοση του πλαισίου RACEX στη λειτουργία native execution του Intel Xeon Phi. Σε αυτήν την λειτουργία εκτέλεσης, η εφαρμογή πρέπει να γίνει compile στη μηχανή υποδοχής για να στοχεύσει την αρχιτεκτονική Xeon Phi και μετά να μεταφερθεί στον συνεπεξεργαστή από τον οποίο θα εκκινηθεί και θα εκτελεστεί. Χρησιμοποιούμε το *micnativeloadex*, ένα εργαλείο λογισμικού που παρέχεται από την Intel στη στοίβα λογισμικού Manycore Platform Software Stack, για να μεταφέρουμε το εκτελέσιμο αρχείο και όλες τις εξαρτήσεις του στον συνεργαζόμενο επεξεργαστή και να ξεκινήσει η απομακρυσμένη διαδικασία.

Έχουμε επιλέξει να μετρήσουμε την απόδοση του συστήματος μας ενάντια σε workloads από την Rodinia Heterogeneous Benchmark Suite Family [36]. Τα workloads που επιλέξαμε είναι: η Computational Fluid Dynamics (CFD), η οποία είναι ένας αδιάσπαστος διαχωριστής πεπερασμένων όγκων για τις τρισδιάστατες εξισώσεις Euler για συμπιεσμένα ρευστά, το HotSpot (HS) που είναι ένα εργαλείο θερμικής προσομοίωσης που χρησιμοποιείται για την εκτίμηση των επεξεργαστών, LU Decomposition (LUD) που είναι ένας αλγόριθμος για τον υπολογισμό των λύσεων ενός συνόλου γραμμικών εξισώσεων, το Needleman-Wunsch (NW) που είναι μια μέθοδος βελτιστοποίησης για την ευθυγράμμιση ακολουθιών DNA και την Breadth-First Search (BFS) η οποία διασχίζει όλα τα συνδεδεμένα στοιχεία σε ένα γράφημα. Εξετάζουμε τις γενικές επιδόσεις του πλαισίου μας για τα προαναφερθέντα workloads με αριθμό νημάτων 56, 112 και 224 που δεσμεύονται στον επιταχυντή στις δύο διαμορφώσεις δικτύου. Ο προκύπτων κανονικοποιημένος χρόνος εκτέλεσης ανά φόρτο εργασίας παρουσιάζεται στα Σχήματα **??**.

Τα αποτελέσματα που παρουσιάζονται στο Σχήμα **??** κανονικοποιούντε με βάση την απόδοση της φυσικής εκτέλεσης που εκτελείται απευθείας στο μηχάνημα υποδοχής. Όπως αναμενόταν, τα αποτελέσματα δεν δείχνουν υποβάθμιση της απόδοσης για το RACEX σε σύγκριση με τις επιδόσεις της φυσικής εκτέλεσης. Το πλαίσιο μας εμπλέκεται στη μεταφορά του εκτελέσιμου στον coprocessor και στην εκκίνηση της απομακρυσμένης διαδικασίας. Αφού δημιουργηθεί η απομακρυσμένη διαδικασία, το πλαίσιο μας δεν εμπλέκεται στην εκτέλεση της εφαρμογής, καθώς ο τρόπος εκτέλεσης υπαγορεύει ότι η εφαρμογή θα εκτελείται αποκλειστικά στον επιταχυντή και μόλις ολοκληρωθεί, η τυποποιημένη έξοδος θα μεταφερθεί στον host. Έτσι, αναμένουμε ότι η πραγματική απόδοση των φόρτων εργασίας που αξιολογήσαμε δεν δείχνει σημαντική υποβάθμιση της απόδοσης, όπως έχει αποδειχθεί. Οι τυχόν διακυμάνσεις που παρουσιάζονται στα αποτελέσματα που παρουσιάστηκαν και δείχνουν κάτω από τους φυσικούς χρόνους εκτέλεσης για το πλαίσιο RACEX, βρίσκονται εντός του περιθωρίου σφάλματος των μετρήσεων και είναι αμελητέες, καθώς είναι η μέση τιμή των επαναλαμβανόμενων μετρήσεων και η διαφορά τους από την φυσική εκτέλεση είναι ασήμαντη.

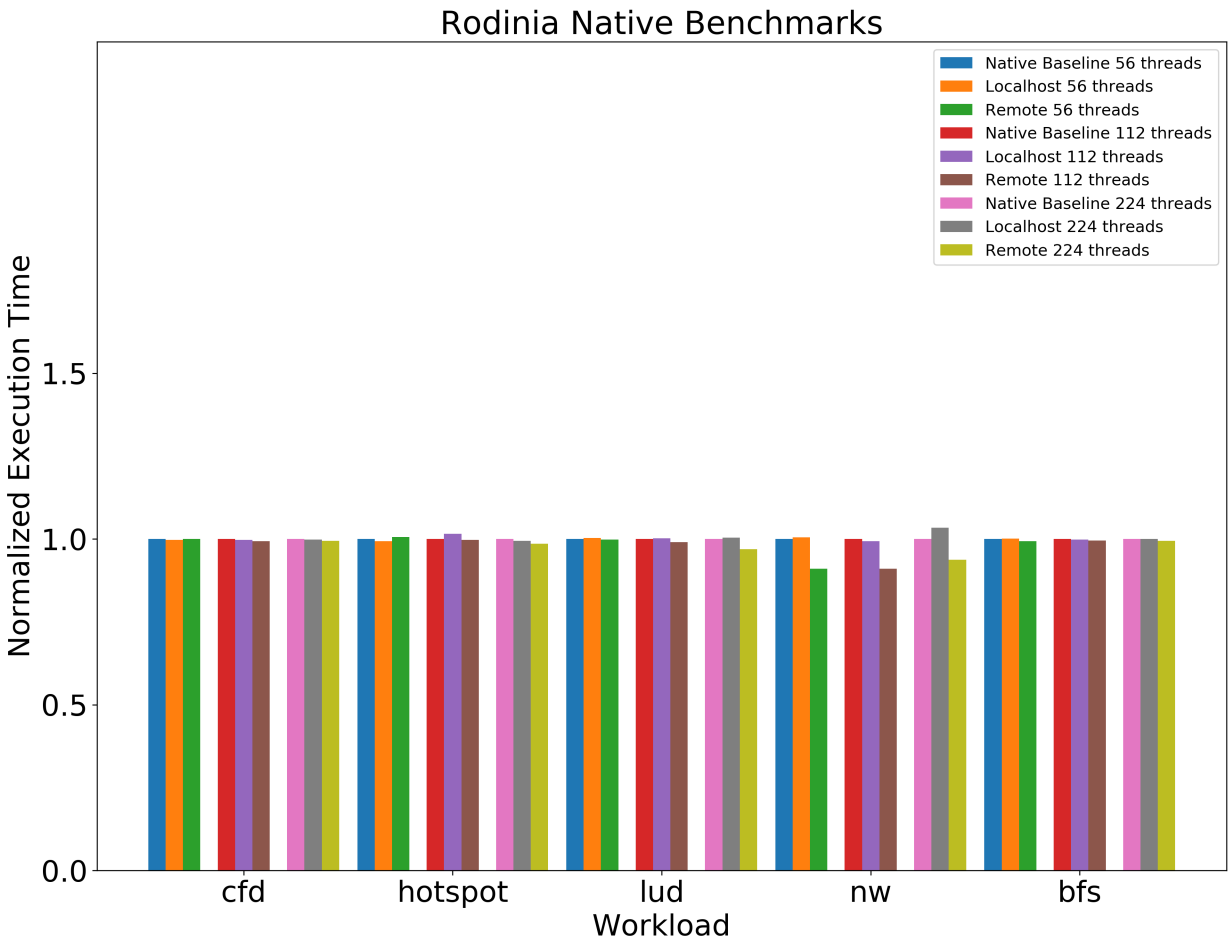


Figure 4.6: Rodinia Benchmarks Native Execution Mode

4.4 Offload Execution Mode

Σε αυτήν την υποενότητα, αξιολογούμε την απόδοση του πλαισίου RACEX στον offload τρόπο εκτέλεσης του Intel Xeon Phi. Σε αυτή τη λειτουργία, η εκτέλεση μιας εφαρμογής χωρίζεται μεταξύ του κεντρικού υπολογιστή και του συνεπεξεργαστή με τον πρώτο να εκφορτώνει τμήμα εκτέλεσης που απαιτούν υπολογιστική ένταση. Στον συνεργαζόμενο επεξεργαστή, καθορίζεται ένας προκαθορισμένος αριθμός νημάτων στις απομακρυσμένες διεργασίες. Ο αριθμός των δημιουργούμενων νημάτων και η ανάθεση τους καθορίζονται από καθορισμένες από το χρήστη περιβαλλοντικές μεταβλητές που έχουν διαμορφωθεί στον κεντρικό υπολογιστή πριν από την εκτέλεση της προβλεπόμενης εφαρμογής. Αξιολογούμε την απόδοση του πλαισίου μας στον τρόπο εκτέλεσης του offload με την εκτέλεση των ίδιων φόρτων εργασίας όπως προηγουμένως από την οικογένεια Rodinia Heterogeneous Benchmark Suite [36] και τις ίδιες διαμορφώσεις νήματος και δικτύου. Ο προκύπτων κανονικοποιημένος χρόνος εκτέλεσης ανά φόρτο εργασίας παρουσιάζεται στο Σχήμα **??**.

Η απόδοση που παρατηρούμε από τα διαφορετικά φορτία εργασίας κυμαίνεται μεταξύ των δύο διαμορφώσεων δικτύου σε σύγκριση με τη φυσική απόδοση για τις τρεις διαφορετικές διαμορφώσεις των νημάτων. Παρατηρούμε μια near nattive απόδοση για το φόρτο εργασίας CFD σε όλες τις

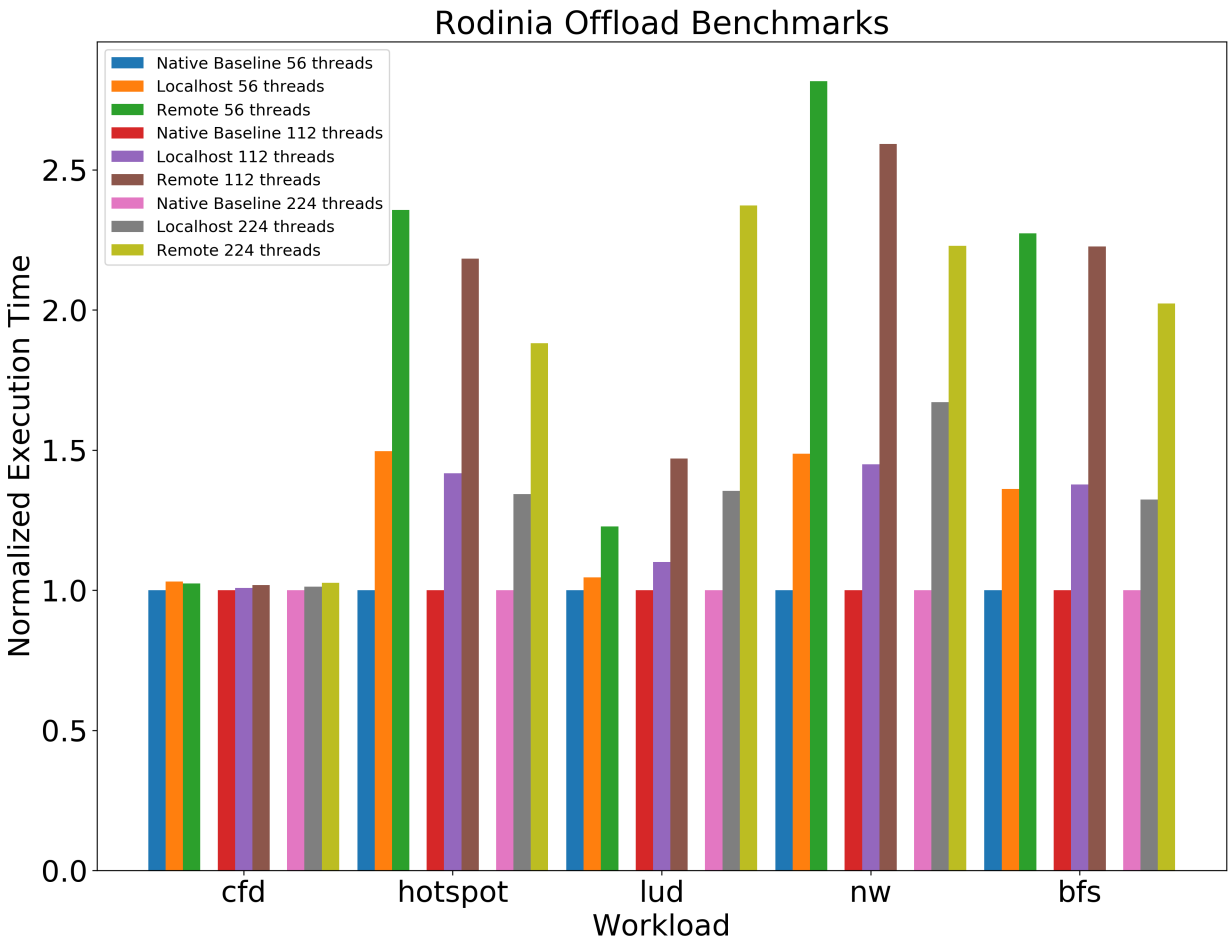


Figure 4.7: Rodinia Benchmarks Offload Execution Mode

διαμορφώσεις νήματος και πολλά υποσχόμενα αποτελέσματα από το φόρτο εργασίας LUD. Το φόρτο εργασίας LUD παρουσιάζει near native απόδοση για τη διαμόρφωση του απομακρυσμένου δικτύου για 56 νήματα και ένα overhead απόδοσης που συσσωρεύεται καθώς πάμε σε 112 και 224 νήματα. Αυτό μπορεί να εξηγηθεί αν εξετάσουμε προσεχώς τον χρόνο εκτέλεσης της φυσικής εκτέλεσης του φόρτου εργασίας για τις τρεις διαμορφώσεις νήματος. Όταν εκτελούμε το φορτίο LUD με 56 νήματα, ο χρόνος εκτέλεσης είναι περίπου 24 δευτερόλεπτα ενώ για τα 112 νήματα ο χρόνος εκτέλεσης πέφτει στα 11 δευτερόλεπτα και ακόμη περισσότερο για τα 224 θέματα, όπου εκτελεί λίγο λιγότερο από 4 δευτερόλεπτα. Λαμβάνοντας υπόψη τα γενικά χαρακτηριστικά που παρουσιάζει το πλαίσιο μας για τις παραμέτρους του απομακρυσμένου δικτύου στο φόρτο εργασίας LUD, το οποίο παραμένει σταθερό σε περίπου 5 δευτερόλεπτα, είναι προφανές ότι για την αρχική απόδοση με 56 νήματα, η γενική επιβάρυνση του πλαισίου μας είναι αμελητέα σε σχέση με τη συνολική εκτέλεση. Ωστόσο, καθώς ο χρόνος εκτέλεσης της φυσικής εκτέλεσης μειώνεται με την αύξηση του αριθμού των νημάτων, το overhead του πλαισίου μας καθίστανται όλο και πιο σημαντικά. Επομένως, η προκύπτουσα συμπεριφορά που απεικονίζεται στην Εικόνα **??** για το φόρτο εργασίας LUD με 224 θέματα είναι αιτιολογημένη. Επιπλέον, οι φόρτοι εργασίας όπως το HotSpot, το Needleman-Wunsch και το BFS έχουν χρόνο εκτέλεσης της γυσικής εκτέλεσης που είναι αρκετά χαμηλός ώστε η καθυστέρηση μεταφοράς δικτύου

που διαθέτει το πλαίσιο μας να έχει ένα σημαντικό ρόλο στο συνολικό χρόνο εκτέλεσης του φόρτου εργασίας όπως απεικονίζεται στο παρουσιαζόμενο Σχήμα **??**.

Οι κυμαινόμενες επιδόσεις που παρατηρούνται μεταξύ των διαφορετικών φόρτων εργασίας, με ορισμένο φόρτο εργασίας που να έχει near native απόδοση και άλλοι με σημαντική επιβάρυνση, μπορεί επίσης εν μέρει να ανιχνευθεί στον συνολικό αριθμό εργασιών εκφόρτωσης που περιλαμβάνει κάθε φόρτο εργασίας. Οι φόρτοι εργασίας με πολλαπλές λειτουργίες εκφόρτωσης επιβαρύνονται με την επιβάρυνση των πολλαπλών επιβαρυνσεων δικτύου, οι οποίες, όπως διερευνήσαμε στο τμήμα αξιολόγησης microbenchmarks, είναι κυρίως υπεύθυνες για την επιβάρυνση του πλαισίου μας. Συνεπώς, οι φόρτοι εργασίας με λιγότερες λειτουργίες εκφόρτωσης αντιμετωπίζουν λιγότερες καθυστερήσεις δικτύου και συνολική καθυστέρηση. Για τις προηγούμενες παρατηρήσεις θεωρούμε ένα σενάριο όπου το μέγεθος των δεδομένων που εκφορτώνονται στον συνεπεξεργαστή παραμένει σταθερό. Έτσι, μπορούμε να συμπεράνουμε ότι λόγω της φύσης του μέσου μετάδοσης δικτύου το οποίο χρησιμοποιεί το πλαίσιο μας και της εγγενούς καθυστέρησης που παρουσιάζει, το RACEX είναι κατάλληλο και λειτουργεί καλύτερα για φόρτους εργασίας που περιλαμβάνουν λιγότερες λειτουργίες εκφόρτωσης. Καθώς ο αριθμός των διαφορετικών λειτουργιών εκφόρτωσης αυξάνεται, το ίδιο ισχύει και για τα γενικά έξοδα του πλαισίου μας, λόγω των πολλαπλών καθυστερήσεων δικτύου.

4.5 Αξιολόγηση Κλιμακωσιμότητας

Στη συνέχεια, αξιολογούμε την απόδοση του RACEX κατά την ταυτόχρονη πρόσβαση σε πόρους του επιταχυντή από πολλούς πελάτες ενώ εκτελούνε μια εφαρμογή υψηλού επιπέδου. Έχουμε επιλέξει ένα υποσύνολο των φόρτων εργασίας που περιλαμβάνονται στην Rodinia Heterogeneous Benchmark Suite [36] που έχουμε ήδη εισαγάγει στις προηγούμενες ενότητες αξιολόγησης. Εκτελούμε τα διαφορετικά φορτία πρώτα σε ένα εγγενές περιβάλλον στον κεντρικό υπολογιστή στον οποίο είναι άμεσα συνδεδεμένος ο συνεπεξεργαστής, προκειμένου να αποκτήσει την απόδοση κλιμάκωσης σε φυσική εκτέλεση του Intel Xeon Phi και στη συνέχεια στο τοπικό και το απομακρυσμένο δίκτυο, από το οποίο η πραγματική απόδοση του RACEX λαμβάνεται. Επιπλέον, αξιολογούμε τον τρόπο με τον οποίο διαφοροποιείται η απόδοση για διαφορετικούς αριθμούς δημιουργούμενων νημάτων στον συνεπεξεργαστή (56, 112, 224) σε συνδυασμό με διαφορετική διαμόρφωση συγγένειας νήματος. Συγκεκριμένα, διερευνάμε τον τρόπο με τον οποίο οι φόρτοι εργασίας εκτελούνται όταν οι πελάτες έχουν εκχωρηθεί συγκεκριμένα νήματα στο συνεπεξεργαστή , τα οποία συνδέονται με τις απομακρυσμένες διεργασίες που έχει εκκινήσει κάθε αντίστοιχος πελάτης στον συνεπεξεργαστή. Τα αποτελέσματα της αξιολόγησης της κλιμακωσιμότητας μας απεικονίζονται στο σχήμα **??**.

Κάνουμε αρκετές παρατηρήσεις βάσει των αποτελεσμάτων της αξιολόγησης της κλιμακωσιμότητας. Πρώτα απ 'όλα, όπως αναφέρθηκε στις προηγούμενες ενότητες, λόγω της έκτασης του συνολικού χρόνου εκτέλεσης του φόρτου εργασίας CFD, δεν παρατηρούμε αποδυνάμωση των επιδόσεων στην κλιμάκωση του πλαισίου RACEX σε σύγκριση με τη φυσική κλιμάκωση. Λαμβάνοντας υπόψη την επεκτασιμότητα του πλαισίου σε συνδυασμό με τις διαφορετικές διαμορφώσεις νήματος, κάνουμε τις ακόλουθες παρατηρήσεις, λαμβάνοντας ως παράδειγμα το φόρτο εργασίας CFD. Όπως έχουμε διευκρινίσει, ο συνεπεξεργαστής που έχουμε εκτελέσει τα σενάρια αξιολόγησης έχει 224 νήματα υλικού (1 πυρήνα αφιερωμένο στο λειτουργικό σύστημα). Από το σχήμα 4.8 για το φόρτο εργασίας του CFD όπου βιώνουμε την εγγενή απόδοση για το πλαίσιο μας, είναι προφανές ότι οι γραμμές ομαδοποιούνται σε τρεις ομάδες, που αντιστοιχούν στις τρεις διαφορετικές διαμορφώσεις νημάτων. Κατά τη διάρκεια των σεναρίων αξιολόγησης της κλιμακωσιμότητας μας, αναθέσαμε τα νήματα υλικού στις διεργασίες που ξεκίνησαν στον συνεπεξεργαστή από τους αντίστοιχους απο-

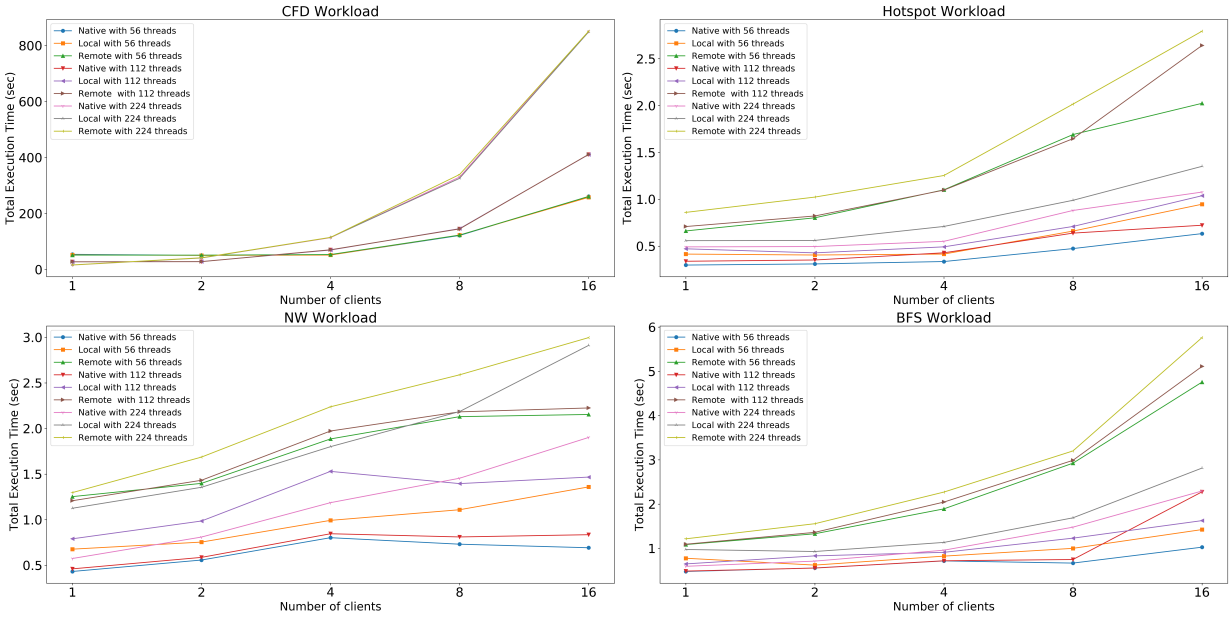


Figure 4.8: RACEX Framework Scalability Evaluation

μακρυσμένους πελάτες, με κυκλικό τρόπο έως ότου έχουν ανατεθεί όλα τα νήματα και στη συνέχεια, σε περίπτωση πρόσθετων διεργασιών πελάτη, θα πραγματοποιούνταν επικαλυπτόμενες αναθέσεις. Για παράδειγμα, όταν κάθε διεργασία καταλαμβάνει 56 νήματα, προσαρμόζουμε τα πρώτα 56 νήματα του συνεπεξεργαστή στον πρώτο πελάτη και τα επόμενα 56 νήματα στην επόμενη διεργασία πελάτη. Στη συνέχεια, για τέσσερις πελάτες, εξαπλώναμε τη ζήτηση για τα νήματα του συνεπξεραστή, προκειμένου να αξιοποιήσουμε πλήρως την ικανότητα του υλικού. Επομένως, όπως παρατηρούμε από το σχήμα **??** για το φόρτο εργασίας CFD, δε υπάρχει διαφοροποίηση από 1 έως 4 πελάτες για 56 διαμορφώσεις νήματος και στη συνέχεια, καθώς προχωρούμε σε 8 και 16 ταυτόχρονους πελάτες, παρατηρούμε υποβάθμιση των επιδόσεων, καθώς έχουμε επικαλύψεις ανάθεσης των νημάτων σε διεργασίες. Συνεπώς, παρόμοια συμπεριφορά παρατηρήτε διαμόρφωση των νημάτων 112 και 224 όπου η απόδοση υποβαθμίζεται καθώς αυξάνεται ο αριθμός των διεργασιών πελάτη που έχουν εκχωρηθεί στο ίδιο υποσύνολο των νημάτων.

Παρόμοιες παρατηρήσεις σχετικά με την υποβάθμιση της απόδοσης σε σύγκριση με τη συνάφεια των νημάτων στον συνεπεξεργαστή μπορούν επίσης να αντληθούν από το υπόλοιπο φορτίο εργασίας. Ωστόσο, όπως έχουμε δηλώσει προηγουμένως, ο χρόνος εκτέλεσης του υπόλοιπου φόρτου εργασίας που απεικονίζεται στο Σχήμα 4.8 έχει δευτερεύοντα χρόνο εκτέλεσης και ταυτόχρονα έχει πολλαπλές ανεξάρτητες λειτουργίες εκφόρτωσης που πολλαπλασιάζουν τη συνολική καθυστέρηση του δικτύου, συνεπώς, η υπάρχουσα καθυστέρηση δικτύου έχει σημαντικό αντίκτυπο στη συνολική απόδοση.

Chapter 5

Μελλοντική Δουλειά

Η επικοινωνία υψηλής ταχύτητας μεταξύ των κόμβων ενός συμπλέγματος υπολογιστών είναι ζωτικής σημασίας για τους σύγχρονους υπερυπολογιστές και τα κέντρα δεδομένων, προκειμένου να επιτευχθεί η απόδοση που απαιτούν οι τρέχουσες εφαρμογές υψηλής απόδοσης υπολογιστών. Είναι επίσης ένας από τους λόγους που στο παρελθόν η εκτέλεση των εφαρμογών HPC στο σύννεφο είχε αμφισβητηθεί, καθώς υπήρχε έλλειψη υπηρεσιών υψηλής ταχύτητας συνδεσιμότητας που προσφέρονται από τους παρόχους μεταξύ των διαφόρων VM. Παρόλο που το πρόβλημα συνεχίζει να περιορίζει την υιοθέτηση των υπηρεσιών Cloud Computing από την κοινότητα HPC, έχουν γίνει βελτιώσεις, ενώ οι πάροχοι προσφέρουν τώρα δυνατότητες δικτύωσης υψηλού εύρους ζώνης, χαμηλής καθυστέρησης για τους πελάτες τους, χρησιμοποιώντας διασυνδέσεις υψηλής ταχύτητας και συγκεκριμένη τοπολογική τοποθέτηση των VMs [37, 38].

Σε αυτή την εργασία προτείνουμε το RACEX ως πλαίσιο απομακρυσμένης εκτέλεσης επιταχυντή για υπολογιστικά συμπλέγματα σε περιβάλλοντα cloud computing που επιτρέπει την κοινή χρήση επιταχυνόμενων πόρων σε επίπεδο συμπλέγματος, είτε παρέχοντας πρόσβαση σε εικονικές μηχανές στον ίδιο διακομιστή με τον επιταχυντή σε ή σε απομακρυσμένους κόμβους συμπλέγματος. Έχουμε υλοποιήσει ένα πρωτότυπο της ιδέας μας για την αξιολόγηση του προτεινόμενου πλαισίου, στοχεύοντας στον συν-επεξεργαστή Intel Xeon Phi, χρησιμοποιώντας BSD Sockets και τυποποιημένη Ethernet συνδεσιμότητα δικτύου μεταξύ του κεντρικού διακομιστή στον οποίο είναι συνδεδεμένος ο συνεπεξεργαστής και του κόμβου του πελάτη χρησιμοποιούνται για την αξιολόγησή μας. Όπως έχουμε δει στην ενότητα αξιολόγησης, το μεγαλύτερο μερίδιο του overhead που παρουσιάζει το πλαίσιο μας αποδίδεται στο συνολικό χρόνο καθυστέρησης μεταφοράς δικτύου που περιλαμβάνει ο Χρόνος μετάδοσης και το Stack TCP/IP. Υποστηρίζουμε ότι με την απασχόληση διασυνδεδεμένων δικτύων υψηλής απόδοσης όπως το InfiniBand που χαρακτηρίζεται από υψηλή απόδοση και χαμηλή καθυστέρηση, το πλαίσιο μας μπορεί να μετριάσει την καθυστερημένη επιβάρυνση που παρουσιάζει και να παρουσιάσει near native επιδόσεις.

Επιπλέον, είναι θεμελιώδες για ένα πλαίσιο κοινής χρήσης επιταχυντή για περιβάλλοντα υπολογιστικού νέφους να είναι σε θέση να παρέχει στις διάφορες εικονικές μηχανές που φιλοξενούνται στον ίδιο κόμβο συμπλέγματος στο οποίο προσαρτάται ο επιταχυντής, πρόσβαση στους πόρους του επιταχυντή. Στο πλαίσιο μας υποστηρίζουμε την προαναφερθείσα λειτουργικότητα μέσω τακτικών BSD Socket, της στοίβας TCP/IP και ενός εικονικού δικτύου μεταξύ των VM και του κεντρικού λειτουργικού συστήματος. Ωστόσο, όπως είδαμε, η χρήση του Network Stack για επικοινωνία μεταξύ πελατών και διακομιστή δαίμονα δεν είναι ιδανική για την προβλεπόμενη χρήση λόγω της καθυστερημένης επιβάρυνσης που παρουσιάζει. Αντίθετα, υποστηρίζουμε ότι μπορούμε να ενσωματώσουμε στο έργο

μας ένα πλαίσιο επικοινωνίας χαμηλού κόστους, υψηλής απόδοσης μέσα στο ;ίδιο κόμβο για τα VM που βρίσκονται πάνω στο κόμβο, όπως το V4VSockets [39], το οποίο βελτιώνει την ανταλλαγή δεδομένων μεταξύ των κόμβων από άποψη καθυστέρησης με συντελεστή 4,5, επιτρέποντας έτσι στο RACEX να παρέχει ένα αποτελεσματικό πλαίσιο διαμοιρασμού επιταχυντή υψηλής απόδοσης και κλιμάκωσης για τα VM που βρίσκονται στο ίδιο μηχάνημα με τον επιταχυντή.

Chapter 6

Σχετική Δουλειά

Η παροχή απομακρυσμένης πρόσβασης στους επιταχυντές υπήρξε ένα πρόβλημα που απασχολεί τον ακαδημαϊκό κόσμο εδώ και αρκετό καιρό. Οι πιο σημαντικές προσεγγίσεις, οι οποίες έχουν καθιερωθεί τα τελευταία χρόνια στην κατασκευή επιταχυντών από απόσταση, είναι αυτές που στοχεύουν μονάδες GPU.

6.1 rCUDA

Ίσως το πιο δημοφιλές μεταξύ των προτεινόμενων λύσεων και εκείνο που έχει περάσει πέρα από το πεδίο της ακαδημαϊκής κοινότητας και έχει γίνει εμπορικά διαθέσιμο είναι το rCUDA [17]. Το rCUDA επικεντρώνεται στο πλαίσιο CUDA της NVIDIA, την παράλληλη υπολογιστική πλατφόρμα και το μοντέλο προγραμματισμού που προσφέρει η NVIDIA για την αξιοποίηση της υπολογιστικής ισχύος των GPU. Το πλαίσιο επιτρέπει την εξ αποστάσεως πρόσβαση σε GPU συμβατές με CUDA. Προκειμένου να παρέχει απομακρυσμένες υπηρεσίες CUDA, το rCUDA δημιουργεί εικονικές συμβατές συσκευές CUDA στις συσκευές που δεν διαθέτουν GPU συμβατή με CUDA και ζητούν πρόσβαση στους πόρους μιας απομακρυσμένης συσκευής συμβατής με CUDA. Το rCUDA virtualizes τις GPU που υπάρχουν στο υπολογιστικό κέντρο και παρέχει πρόσβαση σε πολλούς υπολογιστές-πελάτες χρησιμοποιώντας μια αρχιτεκτονική Client-Server. Σε αυτό το έργο, οι κλήσεις χρόνου εκτέλεσης CUDA παρεμποδίζονται και προωθούνται σε έναν απομακρυσμένο διακομιστή που τους εκτελεί και επιστρέφει τα αποτελέσματα. Στη δουλειά μας, σε σύγκριση με το rCUDA, στοχεύουμε την επικοινωνία χαμηλού επιπέδου μεταφοράς μεταξύ του κεντρικού υπολογιστή και του επιταχυντή που λειτουργεί με το πρωτόκολλο SCIF. Αποκρύπτουμε το Μεταφορικό Στρώμα μεταξύ του κεντρικού υπολογιστή και του επιταχυντή, κι έτσι προχωρούμε προς ένα κεντρικό σχήμα διαχείρισης επιταχυντή για υπολογιστικά clusters. Επιπλέον, έχουμε δημοσιεύσει το RACEX ως λογισμικό ανοιχτού κώδικα.

6.2 vCUDA

vCUDA [40] είναι μια λύση υπολογιστικής μονάδας επεξεργασίας γραφικών γενικής χρήσης (GPGPU) για εικονικές μηχανές (VMs). Το vCUDA επιτρέπει σε εφαρμογές που εκτελούν εντός VM να επιταχύνουν την επιτάχυνση υλικού από τις υποκείμενες μονάδες επεξεργασίας γραφικών.

Ο σχεδιασμός του πλαισίου περιλαμβάνει υποκλοπή κλήσεων API και ανακατεύθυνση και ειδικό σύστημα RPC για VM. Το vCuda χρησιμοποιεί μια αρχιτεκτονική πελάτη-διακομιστή που αποτελείται από τρία στοιχεία χώρου χρήστη: μια βιβλιοθήκη σε επίπεδο χρήστη, μια δομή δεδομένων που χρησιμοποιείται από τη βιβλιοθήκη, που αντιπροσωπεύει μια εικονική GPU και ένα στοιχείο διακομιστή. Η βιβλιοθήκη είναι υπεύθυνη για την παρακολούθηση και την ανακατεύθυνση κλήσεων API από τον πελάτη στον κεντρικό υπολογιστή, όπου ο διακομιστής τις εκτελεί και επιστρέφει τα αποτελέσματα. Η επικοινωνία μεταξύ πελάτη και διακομιστή υλοποιείται χρησιμοποιώντας το πρωτόκολλο XML-RPC. Εκτός από την εικονικοποίηση, το vCUDA επιτρέπει τη πολυπλεξία των συσκευών μεταξύ πολλαπλών ταυτόχρονων εκτελούμενων OSes, δημιουργώντας ένα νήμα για κάθε πελάτη. Το πλαίσιο παρέχει υποστήριξη για αναστολή και συνέχιση, επιτρέποντας τη διακοπή ή μετακίνηση των περιόδων σύνδεσης μεταξύ υπολογιστών. Το σύστημα εφαρμόζεται στο Xen αλλά είναι φορητό σε διάφορες πλατφόρμες virtualization λόγω του μηχανισμού μετάδοσης δικτύου. Τα πειράματα στο [40] δείχνουν ότι ο χρόνος που δαπανάται στα βήματα κωδικοποίησης-αποκωδικοποίησης του πρωτοκόλλου επικοινωνίας προκαλεί σημαντική αρνητική επίπτωση στη συνολική απόδοση της λύσης.

6.3 Distributed-Shared CUDA

Το DS-CUDA [18], είναι ένα middleware λογισμικό που επιτρέπει τη χρήση πολλών GPU σε ένα περιβάλλον cloud. Εικονικοποιεί τις GPU σε ένα νέφος έτσι ώστε να φαίνονται να είναι τοπικά εγκατεστημένες GPU σε μια μηχανή πελάτη. Οι συγγραφείς παρουσιάζουν ένα middleware λογισμικό με στόχο την αντιμετώπιση προβλημάτων στον προγραμματισμό ετερογενών υπολογιστικών κόμβων. Το σύστημα εφαρμόζει εικονικοποίηση ενός συνόλου υπολογιστών εξοπλισμένων με μονάδες GPU έτσι ώστε να εμφανίζονται σαν να συνδέονται με έναν μόνο κόμβο, προκειμένου να απλοποιηθεί ο προγραμματισμός των εφαρμογών πολλαπλών GPU. Η αρχιτεκτονική του συστήματος αποτελείται από ένα μεμονωμένο κόμβο πελάτη και πολλούς κόμβους διακομιστή, στον οποίο έχουν εγκατασταθεί μία ή περισσότερες συσκευές CUDA. Η επικοινωνία μεταξύ των απομακρυσμένων υπολογιστών και του διακομιστή υλοποιείται μέσω των διεπαφων InfiniBand και μπορεί επίσης να χρησιμοποιεί υποδοχές TCP σε περίπτωση που η υποδομή δικτύου δεν υποστηρίζει το InfiniBand.

6.4 Άλλη σχετική δουλειά

Επιπλέον, στον τομέα του Cloud Computing, οι πάροχοι έχουν ήδη δημιουργήσει μια αγορά όπου παρέχουν ελαστική πρόσβαση στον επιταχυντή, προκειμένου να ικανοποιήσουν τις αυξανόμενες ανάγκες των απαιτητικών πελατών που θέλουν να εκμεταλλευτούν το μαζικά παράλληλο. Στην κοινότητα του IoT, εκτελέστηκε πληθώρα ερευνητικών εργασιών που υποδηλώνουν την ανάγκη παροχής δυνατοτήτων εκφόρτωσης στο περιβάλλον Cloud Computing για συσκευές χαμηλής κατανάλωσης, συνδεδεμένες στο διαδίκτυο [9, 10, 41].

Chapter 7

Σύνοψη

Στην εργασία μας, προτείνουμε το RACEX ως ένα αποτελεσματικό πλαίσιο για την εξ αποστάσεως πρόσβαση σε πόρους επιταχυντών. Εφαρμόζουμε το RACEX ως απόδειξη της ιδέας που στοχεύει στον συν-επεξεργαστή Intel Xeon Phi. Το πλαίσιο RACEX ακολουθεί μια κατανεμημένη αρχιτεκτονική διακομιστή-πελάτη, όπου ένας δαίμονας διακομιστή μπορεί να υποστηρίξει την ταυτόχρονη πρόσβαση στους πόρους Intel Xeon Phi από πολλούς απομακρυσμένους πελάτες. Το RACEX χρησιμοποιεί έναν μηχανισμό προώθησης κλήσεων API, χρησιμοποιώντας ένα προσαρμοσμένο πρωτόκολλο επικοινωνίας για να επιτρέψει την απομακρυσμένη εκτέλεση των κλήσεων API της SCIF της Intel. Το πρωτόκολλο επικοινωνίας SCIF βρίσκεται στην κορυφή του Transport Layer, κάνοντας το RACEX, το οποίο αναδιπλώνει και καθιστά τη λειτουργικότητα του SCIF εξ αποστάσεως διαθέσιμη, ευέλικτη, αποδοτική και συμβατή με υψηλότερες εφαρμογές στοίβας λογισμικού χωρίς την ανάγκη recompile. Η προκαταρκτική αξιολόγηση των επιδόσεων έχει δείξει μια σχεδόν σταθερή γενική επιβάρυνση για το RACEX σε σύγκριση με τις φυσικές επιδόσεις αναφοράς, κυρίως λόγω της καθυστέρησης μετάδοσης του δικτύου. Εντούτοις, η αξιολόγηση εφαρμογών υψηλότερου επιπέδου, μέρος του Rodinia Heterogeneous Computing Benchmark Suite, έδωσε πολλά υποσχόμενα αποτελέσματα, με το πλαίσιο RACEX να εκτελείται σε χαμηλά επίπεδα επιβάρυνσης σε σύγκριση με την εγγενή απόδοση και μπορεί να εξυπηρετήσει πολλούς ταυτόχρονους πελάτες με την ίδια συμπεριφορά.

Εφαρμόζουμε και αξιολογούμε το RACEX τόσο στους τρόπους εκτέλεσης του *native* όσο και του *offload* για τον Coprocessor Intel Xeon Phi. Ως μελλοντικό έργο, σχεδιάζουμε να επεκτείνουμε την εφαρμογή μας για να υποστηρίξουμε τον τρόπο εκτέλεσης *symmetric* και να επιτρέψουμε τη χρήση του πλαισίου MPI. Αυτή η επέκταση εφαρμόζεται εύκολα καθώς η υποκείμενη υποδομή του πλαισίου μας έχει σχεδιαστεί για να υποστηρίζει τη συμβατότητα του τρόπου εκτέλεσης στο μέλλον. Επιπλέον, σχεδιάζουμε να χρησιμοποιήσουμε την τεχνολογία Infiniband ως High Performing Network Interconnect προκειμένου να ελαχιστοποιήσουμε την λανθάνουσα διάρκεια του δικτύου που είναι εγγενής στην αρχική μας εφαρμογή και να μπορέσουμε να επιτύχουμε εγγενείς επιδόσεις. Έχουμε κάνει το έργο μας διαθέσιμο ως λογισμικό ανοιχτού κώδικα, διαθέσιμο στο διαδίκτυο στη διεύθυνση `https://github.com/fertakis/racex`.

Appendix A

Συντομογραφίες

RACEX	Remote ACcelerator EXecution
SCIF	Symmetric Communication Interface
COI	Coprocessor Offload Infrastructure
MIC	Many Integrated Core
MPSS	ManyCore Platform Software Stack
RMA	Remote Memory Access
DMA	Direct Memory Access
BSD	Berkeley Software Distribution
API	Application Programming Interface

Bibliography

[1] S. Crago, K. Dunn, P. Eads, L. Hochstein, D. I. Kang, M. Kang, D. Modium, K. Singh, J. Suh, and J. P. Walters, "Heterogeneous cloud computing," in *2011 IEEE International Conference on Cluster Computing*, pp. 378–385, Sept 2011.

[2] E. Deelman, G. Singh, M. Livny, B. Berriman, and J. Good, "The cost of doing science on the cloud: The montage example," in *2008 SC - International Conference for High Performance Computing, Networking, Storage and Analysis*, pp. 1–12, Nov 2008.

[3] L. Wang, J. Tao, M. Kunze, A. C. Castellanos, D. Kramer, and W. Karl, "Scientific cloud computing: Early definition and experience," in *2008 10th IEEE International Conference on High Performance Computing and Communications*, pp. 825–830, Sept 2008.

[4] A. M. Caulfield, E. S. Chung, A. Putnam, H. Angepat, J. Fowers, M. Haselman, S. Heil, M. Humphrey, P. Kaur, J. Y. Kim, D. Lo, T. Massengill, K. Ovtcharov, M. Papamichael, L. Woods, S. Lanka, D. Chiou, and D. Burger, "A cloud-scale acceleration architecture," in *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pp. 1–13, Oct 2016.

[5] Google, "Accelerated cloud computing." `https://cloud.google.com/gpu/`, 2018.

[6] Nvidia, "Gpu cloud computing." `https://www.nvidia.com/en-us/data-center/gpu-cloud-computing/`, 2018.

[7] Amazon, "High performance computing on aws."

[8] C. Evangelinos and C. N. Hill, "Cloud computing for parallel scientific hpc applications: Feasibility of running coupled atmosphere-ocean climate models on amazon's ec2," vol. 2, 01 2008.

[9] R. M. Shukla and A. Munir, "An efficient computation offloading architecture for the internet of things (iot) devices," in *2017 14th IEEE Annual Consumer Communications Networking Conference (CCNC)*, pp. 728–731, Jan 2017.

[10] F. Samie, V. Tsoutsouras, L. Bauer, S. Xydis, D. Soudris, and J. Henkel, "Computation offloading and resource allocation for low-power iot edge devices," in *2016 IEEE 3rd World Forum on Internet of Things (WF-IoT)*, pp. 7–12, Dec 2016.

[11] V. T. Ravi, M. Becchi, G. Agrawal, and S. Chakradhar, "Supporting gpu sharing in cloud environments with a transparent runtime consolidation framework," in *Proceedings of the*

20th International Symposium on High Performance Distributed Computing, HPDC '11, (New York, NY, USA), pp. 217–228, ACM, 2011.

[12] N. Haydel, S. Gesing, I. Taylor, G. Madey, A. Dakkak, S. G. d. Gonzalo, and W. M. W. Hwu, "Enhancing the usability and utilization of accelerated architectures via docker," in *2015 IEEE/ACM 8th International Conference on Utility and Cloud Computing (UCC)*, pp. 361–367, Dec 2015.

[13] Y. Shen, M. Ferdman, and P. Milder, "Overcoming resource underutilization in spatial cnn accelerators," in *2016 26th International Conference on Field Programmable Logic and Applications (FPL)*, pp. 1–4, Aug 2016.

[14] F. Silla, J. Prades, S. Iserte, and C. Reaño, "Remote gpu virtualization: Is it useful?," in *2016 2nd IEEE International Workshop on High-Performance Interconnection Networks in the Exascale and Big-Data Era (HiPINEB)*, pp. 41–48, March 2016.

[15] S. Mislata and F. Silla, "Using remote accelerators to improve the performance of the fftw library," in *2016 IEEE 18th International Conference on High Performance Computing and Communications; IEEE 14th International Conference on Smart City; IEEE 2nd International Conference on Data Science and Systems (HPCC/SmartCity/DSS)*, pp. 913–920, Dec 2016.

[16] S. Gerangelos and N. Koziris, "vphi: Enabling xeon phi capabilities in virtual machines," in *2017 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, pp. 1333–1340, May 2017.

[17] J. Duato, A. J. Peña, F. Silla, R. Mayo, and E. S. Quintana-Ortí, "rcuda: Reducing the number of gpu-based accelerators in high performance clusters," in *2010 International Conference on High Performance Computing Simulation*, pp. 224–231, June 2010.

[18] M. Oikawa, A. Kawai, K. Nomura, K. Yasuoka, K. Yoshikawa, and T. Narumi, "Ds-cuda: A middleware to use many gpus in the cloud environment," in *2012 SC Companion: High Performance Computing, Networking Storage and Analysis*, pp. 1207–1214, Nov 2012.

[19] R. D. Lauro, F. Giannone, L. Ambrosio, and R. Montella, "Virtualizing general purpose gpus for high performance cloud computing: An application to a fluid simulator," in *2012 IEEE 10th International Symposium on Parallel and Distributed Processing with Applications*, pp. 863–864, July 2012.

[20] Wikipedia, "Digital revolution," Dec. 2017.

[21] G. E. Moore, "Cramming more components onto integrated circuits," *Electronics*, Apr. 1965.

[22] G. M. Amdahl, "Validity of the single processor approach to achieving large scale computing capabilities," in *Proceedings of the April 18-20, 1967, Spring Joint Computer Conference*, AFIPS '67 (Spring), (New York, NY, USA), pp. 483–485, ACM, 1967.

[23] M. J. Flynn, "Some computer organizations and their effectiveness," *IEEE Transactions on Computers*, vol. C-21, pp. 948–960, Sept 1972.

[24] NVIDIA, "Printer friendlynvidia launches the world's first graphics processing unit: Geforce 256."

[25] O. J. D., L. David, G. Naga, H. Mark, K. Jens, L. A. E., and P. T. J., "A survey of general-purpose computation on graphics hardware," *Computer Graphics Forum*, vol. 26, no. 1, pp. 80–113.

[26] T. Messay, C. Chen, R. Ordóñez, and T. M. Taha, "Gpgpu acceleration of a novel calibration method for industrial robots," in *Proceedings of the 2011 IEEE National Aerospace and Electronics Conference (NAECON)*, pp. 124–129, July 2011.

[27] C. L. Hung, P. C. Wu, H. H. Wang, and C. Y. Lin, "Efficient parallel muti-pattern matching using gpgpu acceleration for packet filtering," in *2015 IEEE 17th International Conference on High Performance Computing and Communications, 2015 IEEE 7th International Symposium on Cyberspace Safety and Security, and 2015 IEEE 12th International Conference on Embedded Software and Systems*, pp. 1843–1847, Aug 2015.

[28] K. L. Rice, T. M. Taha, K. M. Iftekharuddin, K. Anderson, and T. Salan, "Gpgpu acceleration of cellular simultaneous recurrent networks adapted for maze traversals," in *The 2011 International Joint Conference on Neural Networks*, pp. 2717–2724, July 2011.

[29] K. Groups, "The open standard for parallel programming of heterogeneous systems."

[30] NVIDIA, "About cuda."

[31] Intel, "Best known methods for using openmp on intel many integrated core (intel mic) architecture." https://software.intel.com/en-us/articles/best-known-methods-for-using-openmp-on-intel-many-integrated-core-intel-mic-architecture, 2013.

[32] TOP500, "The fiftieth top500 list of the fastest supercomputers in the world.." `https://www.top500.org/lists/2017/11/`, 2017.

[33] Intel, "Intel manycore platform software stack."

[34] Intel, "Scif user guide."

[35] G. Developers, "Protocol buffers."

[36] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, S. H. Lee, and K. Skadron, "Rodinia: A benchmark suite for heterogeneous computing," in *2009 IEEE International Symposium on Workload Characterization (IISWC)*, pp. 44–54, Oct 2009.

[37] A. AWS, "Placement groups."

[38] A. AWS, "Ec@ encanced networking."

[39] A. Nanos, S. Gerangelos, I. Alifieraki, and N. Koziris, "V4vsockets: Low-overhead intra-node communication in xen," in *Proceedings of the 5th International Workshop on Cloud Data and Platforms*, CloudDP '15, (New York, NY, USA), pp. 1:1–1:6, ACM, 2015.

[40] L. Shi, H. Chen, J. Sun, and K. Li, "vcuda: Gpu-accelerated high-performance computing in virtual machines," *IEEE Transactions on Computers*, vol. 61, pp. 804–816, June 2012.

[41] H. Guo, J. Liu, and H. Qin, "Collaborative mobile edge computation offloading for iot over fiber-wireless networks," *IEEE Network*, vol. 32, pp. 66–71, Jan 2018.