UNICORN: A BULK SYNCHRONOUS PROGRAMMING MODEL, FRAMEWORK AND RUNTIME FOR HYBRID CPU-GPU CLUSTERS

TARUN BERI

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING
INDIAN INSTITUTE OF TECHNOLOGY DELHI
MARCH 2016

UNICORN: A BULK SYNCHRONOUS PROGRAMMING MODEL, FRAMEWORK AND RUNTIME FOR HYBRID CPU-GPU CLUSTERS

by

TARUN BERI

Department of Computer Science and Engineering

Submitted
in fulfillment of the requirements of the degree of
Doctor of Philosophy
to the

INDIAN INSTITUTE OF TECHNOLOGY DELHI
MARCH 2016

Dedicated to my Family, Teachers and Friends ...

Certificate

This is to certify that the thesis titled **Unicorn: A Bulk Synchronous Programming Model, Framework and Runtime for Hybrid CPU-GPU Clusters** being submitted by **Mr. Tarun Beri** to the **Indian Institute of Technology Delhi** for the award of the degree of **Doctor of Philosophy** in Computer Science and Engineering is a record of original bonafide research work carried out by him under our supervision. The work presented in this thesis has reached the requisite standard and has not been submitted elsewhere either in part or full for the award of any other degree or diploma.

Dr. Subodh Kumar
Associate Professor
Computer Science and Engineering
Indian Institute of Technology Delhi

Dr. Sorav Bansal
Assistant Professor
Computer Science and Engineering
Indian Institute of Technology Delhi

Acknowledgements

I always knew doing a Ph.D. from India's premier institute is tough and along with a full time corporate job it was going to be even more so. I am so grateful to my guide *Prof. Subodh Kumar* and co-guide *Prof. Sorav Bansal*, who have understood my dual responsibilities at all times and made this journey so smooth that today when I look back, it has actually been an amazing experience. Today, I hold the highest regard for both of them and I do not even have the slightest hesitation in recommending them as Ph.D. guides to anyone. I am actually feeling uncomfortable of not being able to put their invaluable contributions into appropriate words. Throughout the duration of my Ph.D., their prudent advice and insightful knowledge have been a constant source of inspiration and encouragement. Prof. Subodh's surprise visit to our paper presentation at IPDPS 2015 was quite reassuring as well. I still remember the time when our paper faced a second consecutive reject in a Tier 1 conference and both of them helped me sail through that tough moment. To quote them "Many good papers get rejected 3 to 4 times before they really rise to the top". I am still learning a lot of things from them and hope to replicate their formal writing skills one day.

My sincere thanks also goes to *Prof. Kolin Paul*, *Prof. Preeti Panda* and *Dr. Yogish Sabharwal* who have been on the review committee for 5+ years and have constantly provided invaluable guidance and feedback.

Working extremely hard and not stumbling to any challenge or work pressure was infused in me right from my childhood. Being born to an influential and highly educated joint family in a small town was a big boon. Somehow it

taught me to learn from each and every incident and person I came across. My grandparents, parents, paternal and maternal uncles, cousins, in-laws and their families were all instrumental in defining this success. When I was a kid, everyone in my family especially my grand parents *Mr. Nitrajan Pal Beri* and *Mrs. Shakuntla Beri*, my father *Mr. Arun Beri* and my uncles *Mr. Anil Beri* and *Mr. Raman Beri* taught me for long hours and helped me build my understanding on several basic concepts. My mother, *Prof. Chander Beri*, who herself is a Ph.D. in mathematics, is the biggest motivation that has driven me this far. It was her wish that I join this Ph.D. program and its her confidence in me that I am now seeing its completion. Getting married during the Ph.D. could have complicated things but my understanding and supporting partner *Tanvi* and her parents *Dr. Rajinder Mehta* and *Mrs. Mala Mehta* turned it into a delightful journey. Birth of my son, *Taanish*, during this tenure proved to be the much needed rejuvenation. My brother *Ankur* and his wife *Ria* also deserve a very special mention. Several of my responsibilities often got offloaded to them while I was busy with my dual work. Without any grief, they happily accepted this. God has also been very kind to me.

Today, while writing this acknowledgement, I also recall all my school and college teachers who have time and again instilled confidence in me and made me believe in myself. I would also like to take this opportunity to thank my long time friends *Vineet Jindal*, *Jitesh Saghotra*, *Gurpreet Bedi*, *Amandeep Bawa*, *Vikas Sarmal*, *Ajay Paul Singh*, *Amit Goyal*, *Gagan Bansal*, *Prateek Prashar*, *Bikramjit Singh*, *Anand Sinha*, *Vijay Sharma*, *Asif Iqbal*, *Anish Singla*, *Gaurav Gupta*, *Dilraj Singh*, *Thalinder Singh*, *Gagandeep Singh* and

Narayan who have selflessly stayed along throughout.

My colleagues and friends at work *Vineet Batra*, *Avinandan Sengupta*, *Ravi Ahuja*, *Inderpal Bawa*, *Jatin Sasan*, *Harish Kumar*, *Tariq Rafiq*, *Sachin Patidar*, *Rohil Sinha*, *Vaibhav Tyagi*, *Vikas Marda*, *Gordon Dow*, *Ahsan Jafri* and *Pushp Agarwal* also deserve a very special recognition for supporting me helping me out directly or indirectly. While a few spared time to discuss my project and share their feedback and new ideas, others helped me grasp new technological advancements and learnings. I would also like to thank the management of Adobe Systems India Pvt. Ltd. (*Pankaj Mathur*, *Rajesh Budhiraja*, *Lekhraj Sharma*, *Gaurav Jain* and *Viraj Chatterjee*), Cadence Design Systems India Pvt. Ltd. (*Utpal Bhattacharya* and *Parag Chaudhary*) and STMicroelectronics India Pvt. Ltd. (*Anand Singh* and *Vivek Sharma*) who have supported me and let me pursue my studies along with the office work.

I would also like to thank M.Tech. students *Harmeet Singh*, *Vigya Sharma*, *Abhishek Raj*, *Abhishek Kumar* and *Ameen Mohammed* who have worked with me and helped develop the ideas in this work. In the end, my sincere thanks goes to my mother's friend and my teacher *Mrs. Savita Sharma* and her husband *Mr. Sunil Sharma* who have always encouraged and supported me since my childhood. I hope I shall make everyone proud in the time to come and work even harder to take this work and knowledge to the next level.

Tarun Beri

Abstract

Rapid evolution of graphics processing units (GPUs) into general purpose computing devices has made them vital to high performance computing clusters. These computing environments consist of multiple nodes connected by a high speed network such as Infiniband, with each node comprising several multi-core processors and several many-core accelerators. The difficulty of programming hybrid CPU-GPU clusters often limits software's exploitation of full computational power. This thesis addresses this difficulty and presents ***Unicorn*** – a novel parallel programming model for hybrid CPU-GPU clusters and the design and implementation of its runtime.

In particular, this thesis proves that efficient distributed shared memory style programing is possible. We also prove that the simplicity of shared memory style programming can be retained across CPUs and GPUs in a cluster, minus the frustration of dealing with race conditions. And this can be done with a unified abstraction, avoiding much of the complication of dealing with hybrid architectures. This is achieved with the help of transactional semantics, deferred bulk data synchronization, subtask pipelining and various communication and computation scheduling optimizations.

Unicorn provides a bulk synchronous programming model with a global address space. It schedules concurrent tasks of a program in an architecture and topology oblivious manner. It hides the network and exposes CPUs and accelerators loosely as bulk synchronous computing units with logical phases, respectively, of local computation and communication. Each task is further decomposed into coarse-grained concurrently executable subtasks that Unicorn schedules transparently on to available CPU and GPU devices in the cluster. Subtasks employ transactional memory semantics to access and synchronize data, i.e., they check out a private view of the global shared memory

before their local computation phase and check in to the global shared memory afterwards, optionally resolving conflicting writes in a reduction step.

Unicorn's main design goals are easy programmability and a deterministic parallel execution environment. Device, node and cluster management are completely handled by the runtime and no such API is exposed to the application programmer. Load balancing, scheduling and scalability are also fully transparent to the application code. Application programs do not change from cluster to cluster to maintain efficiency. Rather, Unicorn adapts the execution to the set of present devices, the network and their dynamic load. Application code is oblivious to data placement within the cluster as well as to changes in network interfaces and data availability pattern. *Unicorn's* programming model, being deterministic, eliminates data races and deadlocks.

To provide efficiency, *Unicorn's* runtime employs several optimizations. These include prefetching task data and pipelining subtasks in order to overlap their communication with computations. *Unicorn* employs pipelining at two levels – firstly to hide data transfer costs among cluster nodes and secondly to hide DMA communication costs between CPUs and GPUs on all nodes. Among other optimizations, *Unicorn's* work-stealing based scheduler employs a two-level victim selection technique to reduce the overhead of steal operations. Further, it employs special proactive and aggressive stealing mechanism to prevent the said pipelines from stalling (during a steal operation). To prevent a subtask (running on a slow device or on a device behind a slow network or I/O link) from becoming a bottleneck for the entire task, *Unicorn* reassesses its scheduling decisions at runtime and schedules a duplicate instance of a straggling subtask on a potentially faster device. *Unicorn* also employs a software LRU cache at every GPU in the cluster to prevent the shared data between subtasks getting DMA'ed more than once. To further boost GPU performance, *Unicorn* makes aggressive use of CUDA streams and schedules multiple subtasks for simultaneous execution.

To evaluate the design and implementation of *Unicorn*, we parallelize several coarse-grained scientific workloads using Unicorn. We study the scalability

and performance of these benchmarks and also the response of *Unicorn's* runtime by putting it under stress tests like changing the input data availability of these experiments. We also study the load balancing achieved in these experiments and the amount of time the runtime spends in communications.

We find that parallelization of coarse-grained applications like matrix multiplication or 2D FFT using our system requires only about 30 lines of C code to set up the runtime. The rest of the application code is regular single CPU/GPU implementation. This indicates the ease of extending sequential code to a parallel environment. The execution is efficient as well. Using GPUs only, when multiplying two square matrices of size $65536 * 65536$, *Unicorn* achieves a peak performance of 7.81 TFlop/s when run over 28 Tesla M2070 GPUs (1.03 TFlop/s theoretical peak) of our 14-node cluster (with subtasks of size $4096 * 4096$). On the other hand, CUPLAPACK [28], a linear algebra package specifically coded and optimized from scratch, reports 8 TFlop/s while multiplying two square matrices of size $62000 * 62000$ using 32 Quadro FX 5800 GPUs (0.624 TFlop/s theoretical peak) of a 16 node cluster connected via QDR InfiniBand.

Fine-grained applications, however, may not fit into our system as efficiently. Such applications often require frequent communication of small data. This is inherently against our bulk synchronous design and more advanced optimizations may be needed to make these applications profitable.

Contents

1 Introduction **1**

1.1 Application Model 3

1.2 Global Address Space 4

1.3 Scheduling 5

1.3.1 Reducing Steal Overhead 7

1.3.2 Handling CPU-GPU Performance Disparity 7

1.4 Performance Optimizations 9

1.4.1 Data Transfer Optimizations 10

1.4.1.1 Locality Aware Scheduling 10

1.4.1.2 Pipelining 11

1.4.1.3 Grouping Communications 11

1.4.1.4 Software GPU Caches 12

1.4.2 Scheduling Optimizations 12

1.4.3 Address Space Optimizations 13

1.5 High Level Abstractions 13

1.6 Epilogue 14

2 Programming Model **17**

2.1 Data Subscriptions and Lazy Memory 23

3 Runtime System **25**

3.1 Device and Node Management 25

3.1.1 Runtime Internals 27

3.1.2 List of Threads 31

3.2 Shared Address Spaces 32

3.3 Network Subsystem 41

3.4 Pipelining 44

3.5 Scheduling Subsystem 45

3.5.1 Work Stealing in Unicorn 46

3.5.1.1 ProSteal 49

3.5.1.2 Locality-aware Scheduling 52

3.5.1.2.1 Greedy Scheduling 54

3.5.1.2.2 Locality-aware work stealing . . . 55

3.5.2 Work-Group Calibration 55

3.5.3 Multi-Assign 57

3.5.3.1 Subtask Cancellation 58

3.5.4 Scheduling Across Task Barriers 59

3.5.5 Scheduling Concurrent Tasks 60

3.6 Software GPU Cache 60

3.7 Conflict Resolution 61

4 Pseudo Code Samples **65**

5 Experimental Evaluation 75

5.1 Unicorn Parallelization of Benchmarks 78

5.1.1 Characteristics of Benchmarks 81

5.2 Performance Scaling 82

5.2.1 CPU versus GPU versus CPU+GPU 85

5.2.2 PageRank 88

5.3 Scheduling 89

5.3.1 Work Stealing 90

5.3.1.1 One-level vs. Two-level 91

5.3.1.2 ProSteal 94

5.3.2 Locality-aware Scheduling 94

5.4 Load Balancing 100

5.5 Stress Tests 102

5.5.1 Heterogeneous Subtasks 103

5.5.2 Input Data Distributions 103

5.5.3 Varying Subtask Size 104

5.6 Unicorn Optimizations 105

5.6.1 Multi-Assign 106

5.6.2 Pipelining 108

5.6.3 Software cache for GPUs 109

5.6.4 Data Compression 110

5.7 Overhead Analysis 111

5.7.1 Varying CPU cores 112

5.7.2 Unicorn Time versus Application Time 114

5.7.3 Data Transfer Frequency 115

5.8 Unicorn versus others 116

6 Application Profiling 119

7 Public API 125

8 Related Work 145

9 Conclusions and Future Work 151

Bibliography 153

Appendix 163

10.1 Unicorn's MapReduce Extension 163

10.2 Scratch Buffers 164

10.3 Matrix Multiplication Source Code 165

Unicorn Publications 175

Biography 177

List of Illustrations

Figures

2.1 Application program in Unicorn 18

2.2 Mapping a BSP superstep to a Unicorn task 19

2.3 Execution Stages of a Subtask 20

2.4 Hierarchical Reduction - leaf nodes are subtasks, others are reduced subtasks; dotted lines are inter-node data transfer . . 21

3.1 The design of the Unicorn runtime – Light orange region represents the instance of the runtime on each node in the cluster 27

3.2 Address Space Ownerships – PD represents Address Space Ownership Directory, TD represents Temporary Ownership Directory, x represents non-existent directory, red text represents changes from last state and cells in light blue background represent directory changes via explicit ownership update messages; Node 1 is the address space master node. 37

3.3 Loss in victim's pipeline due to work stealing 50

5.1 Characteristics of various benchmarks 82

5.2 Performance analysis of various benchmarks 83

5.3 Scaling with increasing problem size 85

5.4 Image Convolution – GPU vs. CPU+GPU 86

5.5 Matrix Multiplication – GPU vs. CPU+GPU 86

5.6 2D FFT – GPU vs. CPU+GPU 87

5.7 Experiments with matrices of size 32768 × 32768 (lower is better) 88

5.8 Page Rank 89

5.9 Centralized scheduling versus Unicorn – 14 nodes 90

5.10 One-level versus two-level work stealing 92

5.11 Work Stealing – with and without ProSteal 93

5.12 Locality aware scheduling 96

5.13 Locality aware scheduling (Contd.) 97

5.14 Image Convolution: Locality aware work-stealing (*Derived Affinity*) 99

5.15 Matrix Multiplication: Locality aware work-stealing (*Derived Affinity*) 99

5.16 Load Balancing (Image Convolution) – W denotes a CPU work group and G denotes a GPU device – Block random data distribution 101

5.17 Load Balancing (Page Rank) – W denotes a CPU work group and G denotes a GPU device 102

5.18 Load Balancing (Matrix Multiplication) – 32768 × 32768 matrices – Centralized data distribution 102

5.19 Load Balancing (Heterogeneous Subtasks) – W denotes a CPU work group and G denotes a GPU device 103

5.20 Impact of initial data distribution pattern 104

5.21 Subtask size ($N \times N$) – experiments executed on 14 nodes . . 105

5.22 Multi-Assign (no external load) 106

5.23 Multi-assign under external load (Image Convolution) – 4 nodes106

5.24 Pipelining (Image Convolution) 108

5.25 Matrix Multiplication – GPU Cache Eviction Strategies 109

5.26 PageRank data compression (250 million web pages) 111

5.27 Varying CPU cores used in subtask computation 112

5.28 Matrix Multiplication – Varying CPU core affinity 113

5.29 Library time versus application time 114

5.30 Data Transfer Frequency 115

5.31 Unicorn versus StarPU 117

5.32 Unicorn versus SUMMA 117

6.1 Sample Unicorn Logs (part 1) 119

6.2 Sample Unicorn Logs (part 2) 120

6.3 Sample Unicorn Logs (part 3) 121

6.4 Performance 123

6.5 Load Balance 123

6.6 Compute Communication Overlap 124

6.7 Event Timeline 124

Codes

4.1 Unicorn program for square matrix multiplication 66

4.2 Unicorn callbacks for square matrix multiplication 68

4.3 Unicorn program for 2D-FFT 70

4.4 Unicorn callbacks for 2D-FFT 72

7.1 Unicorn Header: pmPublicDefinitions.h 125

7.2 Unicorn Header: pmPublicUtilities.h 142

10.1 File matmul.h 165

10.2 File matmul.cpp 166

10.3 File matmul.cu 172

Chapter 1

Introduction

High performance computing environments consist of multiple nodes connected by a network. Each node may comprise multi-core processors (CPUs) and possibly many-core accelerators like graphics processing units (GPUs). Attractive performance per-$ and per-watt of such accelerators have rendered them mainstream in scientific and other domains. Nonetheless, writing efficient programs employing both CPUs and GPUs across a network remains challenging.

Traditionally, two major parallel programming paradigms have been proposed – the first is a shared memory approach while the second is based upon message passing. Shared memory programming [19] is considered intuitive and familiar, but it quickly becomes inefficient as the shared memory gets distributed (DSM) across a network [45, 3, 49, 52]. This is because DSM systems generally employ complex memory consistency protocols (often requiring application specific knowledge) resulting in high coherence overheads. On the other hand, message passing alternatives like MPI [36] generally require transfer of not only data but also some control and program state information. With a large number of small data transfers, the latency quickly becomes a bottleneck. Also, maintaining the additional baggage of control and state information complicates MPI programs.

Our system uses the best of these two worlds by exposing a DSM style model

to the programmer but behind the scenes employs shared memory (within a node) and message passing (across nodes) for data transfers. Deferred data exchanges in bulk amortize message passing overheads. Bulk synchronization also eliminates race conditions and provides determinism, significantly simplifying programming effort. Our work concludes that efficiency can indeed be achieved with bulk-synchronous distributed shared memory. In particular, we demonstrate that the traditional inefficiency of the shared memory approach can be offset by hiding communication latency behind coarse-grained computation and batching communication using ideas from transactional memory: the application operates on local *views* of the global shared memory and inter-view conflict is resolved lazily.

GPUs are discrete off-chip devices with exclusive memory and user must often explicitly copy the data from the host CPU. Once data is copied to the device, the GPU works independently and after finishing the computation, the user synchronizes the results back into the host memory. The explicit placement and retrieval of data in GPU memory and the SIMD execution of thousands of hardware threads in lock-step makes GPU programming a lot more complex than traditional CPU programming. As such, it is desirable not to further complicate this when designing programming models for GPU clusters. Our choice of abstracting the simplicity of distributed shared memory style programming is a step in that direction.

Our programming model is based on the theoretical Bulk Synchronous Parallel computing model (BSP) [57] and thus complete. This thesis belies the conventional wisdom that BSP is only a bridging model and too inefficient to act as a real programming model. The thesis shows that with balanced load

and other well targetted optimizations, coarse-grained parallel applications can indeed be efficient in this model on a CPU/GPU cluster. A motivating factor for choosing BSP is that the GPU architecture is the most efficient with bulk-synchronous computation. Extending this bulk-synchronization allows application programs to naturally generalize from one GPU to multiple and from one node to a cluster of GPU nodes. Our system extends this generalization to CPU cores as well, which are also individually viewed as bulk synchronous devices. This uniformity allows us to simplify application programs and much of the complexity in parallelism management, load distribution, communication and scalability remains confined within the runtime. As a result, most application programmers are left to deal only with the core logic of the application. We now introduce the important components of our programming environment.

1.1 Application Model

The application in our framework consists of a set of interdependent *tasks* and can be thought of logically as BSP super-steps. It may create address spaces (section 1.2) as well as pass them from task to task. Tasks may also hierarchically spawn other tasks. A task is ready to be scheduled at the completion of all tasks it depends on. The task hierarchy is abstract and a program-time decision. It is independent of the cluster topology and is dynamically mapped to and executed on any given cluster by our runtime. A task may request any number of concurrent work-sharing *subtasks*, which is a data-parallel work-sharing construct of a task, and is individually scheduled by our runtime on any available CPU or GPU in the cluster. Each subtask

executes an application-provided "kernel function," which must determine its share of work based on its subtask id and any task-wide parameters. Unicorn schedules subtasks on devices (CPU or GPU), in a load-balanced manner while also accounting for the location of their data.

Unlike many other distributed programming models (section 8), we do not expect our applications to provide a dependency graph comprising all schedulable entities (subtasks in our case). Rather, Unicorn based applications specify dependencies only among tasks. These dependencies are implicit and inferred by Unicorn from associated address spaces and their specified usage (read-only, write-only, read-write). Subtasks of a task are concurrent with no inter-dependencies. This has two advantages. First, it is natural to think of an application as a graph of tasks (as opposed to a graph of subtasks). Second, this reduces the size and processing time of dependency graph and lets us perform several optimizations among subtasks. These optimizations include out-of-order subtask execution, arbitrary grouping of subtasks, freely migrating subtasks across cluster nodes, etc.

1.2 Global Address Space

Our runtime supports allocation of global shared memory regions called *address spaces*. Usually a task's input and output are stored here. While an address space is logically shared, it may be physically distributed across multiple machines and devices by our runtime. The task registers callbacks to indicate input data distribution and access patterns, the subtask logic and the logic to combine, i.e., *reduce*, subtask-local output into the shared space.

The subtask code operates only on its local copy of the data. Its memory writes are visible only to its dependent tasks, implying that any computation requiring this output must either be in the same subtask or in a subsequent task. We have chosen to omit any 'flush' or global read/write primitive and the address space is updated only at the end of each task. The nature of programming simplifies significantly because of that choice. We demonstrate later that this style of programming is still efficient and powerful enough for many coarse-grained scientific applications.

Our address spaces are inspired by the idea of transactional memory. Each subtask logically checks-out its local view from global shared address space and after operating on it, the subtask checks-in its private view back to the shared address space. These private views with deferred synchronization lead to sequential consistency trivially. This also avoids several data hazards and deadlocks among subtasks, which again simplifies the application code. Thirdly, this helps *Unicorn* perform a special scheduling optimization, called *multi-assign* (section 3.5.3), where several independent instances of a straggling subtask are started in the cluster. Because of the transactional design of our address spaces all private views of one but all subtasks get trivially discarded. Finally, the transactional design of our address spaces helps minimize coherence messages in the cluster (section 3.2).

1.3 Scheduling

In our model, the number of subtasks of a task are known at the time the task is submitted, but their workload is not. Scheduling these subtasks across

the cluster with balanced load is an important ingredient to scalable computation. Broadly, two dynamic load balancing strategies have been proposed in the past – *Push* and *Pull*. In the former, overloaded computing devices send work to others while in the latter, underloaded devices ask for work from others. *Push* schemes are generally used for small centralized systems as they do not scale well on larger ones. For de-centralized systems, *Pull* schemes offer better scalability and fault tolerance. Work stealing is one of the most effective examples of *Pull* based dynamic load balancing and has been employed in many language and library based systems like Mul-T [46], Cilk [14], OpenMP [19], Intel's Thread Building Blocks (TBB) [56] and Microsoft's Parallel Patterns Library (PPL) [16].

Although work stealing (with random victim selection) has proven to be quite effective in several parallel systems, it poses several challenges in our context of hybrid CPU-GPU clusters. First, care is needed to limit the overhead of stealing. This is particularly true in the context of task pipelines, where a given subtask flows through many stages like input preparation stage, data fetch stage, execution stage, etc. Sometimes it is useful to steal a subtask before a computing device exhausts its pipeline, but it is also more complicated. Secondly, the effectiveness of work-stealing tends to reduce as the heterogeneity and computational disparity between devices grows. CPUs allow threads to be scheduled on a single core, but GPUs do not allow per-core scheduling and kernels can occupy the entire GPU. Subtasks large enough to effectively use the GPU can be too slow on the CPU. Shorter ones may improve CPU performance, but GPUs remain under-utilized. This disparity between CPUs and GPUs poses scheduling challenges.

1.3.1 Reducing Steal Overhead

Unicorn uses two techniques to limit the overhead of stealing. First, it employs a steal-agent per node in the cluster. Due to shared memory usage, the agent is able to easily monitor local load and track the ability of each local device to service an incoming steal request. This guided victim selection helps making more targeted steal requests, thereby reducing the number of unsuccessful steal attempts in the cluster. The selection of a victim node for a steal request remains random.

Secondly, *Unicorn* employs a proactive stealing technique called *ProSteal*. When a device runs out of work (i.e., subtasks), its task pipeline stalls. It can no longer overlap communication of subtasks with computation of others. It needs to prime its task pipeline afresh after it steals a new subtask. Unicorn prevents this performance loss in the system by stealing early (especially for GPU devices) with one or more subtasks still pending with the device. The exact number of subtasks pending with a device at the time of steal is computed dynamically based on the device's rate of subtask execution and the observed latency to prime the pipeline after a stall (section 3.5.1.1).

1.3.2 Handling CPU-GPU Performance Disparity

Task decomposition in our system remains in user's control. However, this can sometimes be tedious. Unicorn simplifies application's subtask sizing by adjusting it to suit the current execution environment. However, choosing sizes for heterogenous devices in the presence of high diversity is tricky.

One alternative, to address the CPU-GPU performance disparity, is to let

the application programmer design subtasks differently for different devices. But this would compromise abstraction and simplicity of programming. In our framework, applications logically partition tasks and remain unaware of where each subtask is scheduled. In any case, dynamically adapting subtask sizes to the current execution environment can be a tedious exercise for the application programmer.

Another alternative is to group, say, multiple CPU cores together into a single device, but this can become complicated with a large number of device types with diverse capabilities. Additionally, this would also compromise the simplicity of the programming model as each kernel must execute on a "set of devices." This also takes away the liberty to use existing sequential CPU functions as subtask kernels, an important design goal for us. Another alternative is to envision a subtask as a set of work-items and schedule a single work-item per CPU core. This is analogous to OpenCL's [55] work-group and work-item. This feature is implemented in our system but again tasks may opt to disregard it in the interest of simplicity. However, in that case all CPU cores and GPUs uniformly execute the subtasks of the same size.

Unicorn supports diversity between devices' computation powers by resizing subtasks at runtime. For example, CPU subtasks may be further split into smaller ones. On a node with, say, two octa-core processors, a CPU subtask may be decomposed into up to 16 smaller units, each scheduled on one core. On the other hand, GPU subtasks may be logically grouped into a bigger one, as multiple GPU subtasks can run concurrently using CUDA [48] streams. On Fermi generation devices, it is possible to launch up to 16

subtasks simultaneously. This example allows a skew of 1:256 in the size of the subtasks executed on CPUs versus those executed on GPUs. Results in section 5.5.3 show the performance improvement obtained by creating such skew on a variety of subtasks.

In general, an application should empirically determine an appropriate subtask size. The goal here is not to select a subtask size that causes too many CPU cache misses or keeps the GPU largely under utilized. Experimenting with a small data set exclusively on one GPU and exclusively on one CPU should help in making this decision.

1.4 Performance Optimizations

Besides the two work-stealing optimizations mentioned above, *Unicorn* employs several other optimizations for efficiency. Some of these performance optimizations target scheduling while others focus on data transfers and minimizing control messages within the runtime. The optimizations include:

1. Data affinity based subtask scheduling to reduce data transfers within the cluster
2. Pipelining subtask data transfers and computation in order to hide communication latency
3. Grouping communications among a pair of nodes to reduce network latency
4. Software cache in GPUs to boost re-use of shared read-only data among subtasks

5. Scheduling multiple instances of the same subtask in the cluster to prevent stragglers

6. Spatially optimized address space accesses

7. Lazy address space coherence within the runtime to reduce management messages

We briefly introduce the main optimizations next.

1.4.1 Data Transfer Optimizations

Some of data transfer optimizations explicitly focus on reducing the amount of data transferred in the cluster, while others attempt to reduce the data transfer time by hiding it behind other ongoing useful computation.

1.4.1.1 Locality Aware Scheduling

Often a coarse-grained computation is decomposed such that adjacent subtasks exhibit spatial locality and access adjacent regions of input address spaces. However, this is sometimes infeasible or it is overly complicated to write programs in this fashion. To help such application programs, our runtime maintains a distributed map of data resident on various nodes and uses it to estimate the affinity of work to different nodes to guide scheduling. Traditionally, locality-aware scheduling has mostly focussed on maximizing reuse of resident data. This approach tends to schedule computation (or subtasks) on nodes having the largest amount of input data resident. We, however, observe that it is equally important to focus on minimizing the cost

of fetching the non-resident data because significant time can be lost in fetching remote data and different devices are able to consume data at different rates. Thus, our locality-aware scheduler takes into account the time spent in fetching remote data and strives to minimize it.

1.4.1.2 Pipelining

To further reduce the cost of fetching remote data, *Unicorn* anticipates subtasks a device may run in the future and fetches its data, hiding communication latency behind ongoing computation. Our scheduler makes subtask assignments in groups. Subtasks in the group are sequentially executed starting with the first one. While a device is executing the first subtask in the assigned group, the next subtask overlaps its communication with the ongoing computation of the first subtask. Similarly, the third subtask overlaps its communication with the second subtask's computation. This continues for all subsequent subtasks in the group.

This mechanism is especially useful for GPUs capable of compute-communication overlap and multiple kernel launches, where we create a pipeline of subtasks. At any given time, one subtask may be transferring its data to the GPU, one or more subtasks may be executing and one subtask may be copying its data out of the GPU.

1.4.1.3 Grouping Communications

Often subtasks access data in patterns. This is especially true of regular coarse-grained experiments where a subtask may access multiple contiguous ranges of data with uniform separation. In such cases, our runtime detects the

access pattern and instead of issuing multiple remote data transfer requests for each subtask, one unified request is issued. At the remote end, the data is packed before being sent to the requestor where it is unpacked and mapped into the requesting subtask's local view. This optimization helps scalability as it prevents flooding of the network with too many small data transfer requests. This also simplifies application programs, which do not need to consider data and communication granularity.

1.4.1.4 Software GPU Caches

In addition to on-CPU caches of memory fetched from remote nodes, our runtime system maintains an on-GPU software LRU cache for portions of the address spaces currently loaded on the GPU device. This greatly helps eliminate unnecessary data transfers when the same read-only data is requested by multiple subtasks scheduled on that GPU, or in cases where data written by a subtask is later read by another subtask of a subsequent task.

1.4.2 Scheduling Optimizations

Among the scheduling optimizations, *Unicorn* employs special mechanism to prevent a few straggling subtasks from becoming a bottleneck for the entire application. If a subtask executing on a device in the cluster takes long to finish, while other devices have become idle, *Unicorn's* scheduler may *multi-assign*, i.e., assign the same subtask to multiple devices. We never migrate away the subtask from the "slow" unit, allowing the multi-assigned units to compete. Migration overheads are high and unnecessary: multi-assign happens only near the end of a task. The results of the first finishing device

are employed and the unused subtasks are aborted and their local writes are discarded. This optimization also helps tackle situations where a subtask straggles as a result of a slow network link or excessive load on one or more nodes in the cluster.

1.4.3 Address Space Optimizations

Unicorn's address spaces are specifically designed for efficient transactional semantics and to minimize explicit cache coherence messages. We do not employ the usual MSI coherence protocol as it has the potential to generate too many cache invalidations. The transactional semantics mean that a subtask sees the global data at the beginning of the task, and then only its own updates to that data. After the task finishes, the writes of the subtasks become visible to subsequent tasks. Thus, we need no explicit coherence messages during the task execution. Only at the end of the task, a few coherence messages may be exchanged to invalidate outdated copies of data.

Address spaces are also optimized for linear and block accesses. Subtasks needing one or more contiguous data ranges (like elements of an array) may request for optimized linear accesses while subtasks needing one or more uniformly separated contiguous data ranges (like sub-matrix of a bigger matrix) may request for optimized block accesses. Internally, *Unicorn* organizes address spaces to adapt to the kind of requested access.

1.5 High Level Abstractions

Unicorn is an extensible framework and it is possible to optimally orchestrate

its functionality into other well known programming models like Map-Reduce [20]. Chapter 2 describes the set of supported callbacks in *Unicorn*. The *Subtask Execution* callback can serve as the Map stage whereas the *Data Synchronization* callback can be programmed to synchronize subtasks reducing two a time. Chapter 5 evaluates this approach by implementing page-rank [51] computation for a collection of web pages. The map stage of the experiment computes contributions of PageRank for all outlinks in the web. The reduce stage accumulates individual contributions on all inlinks for each webpage. Results demonstrate the efficiency of *Unicorn's* implementation and its Map-Reduce suitability in general.

1.6 Epilogue

In addition to demonstrating the efficacy of the proposed system, this thesis explores various optimization parameters. For pipelining, we explore *when to allow steal* and study *the impact of pipeline-flush*. We explore various *GPU cache policies* and *the impact of application-provided hints*. We study *when to multi-assign* and *how often the later assigned device finishes first*. We analyze the impact of *variance in application controlled subtask size*. We also study *the impact of changing input data availability patterns* in the cluster. The primary contributions of this work are:

1. We present a novel shared-memory based parallel programming model that transparently maps and autonomously schedules computation on any cluster of CPUs and GPUs in a load-balanced fashion.

2. We develop a runtime framework supporting the proposed model and investigate the optimizations necessary for such a framework to be practical. Our runtime (*Unicorn*) batches and combines data transfer for efficient communication. It performs prefetching and pipelining allowing compute-communication overlap. It also supports multi-scheduling in response to dynamically changing load.

3. We present a concrete study of scheduling in hybrid CPU-GPU clusters. Specifically, we propose *ProSteal* which is a proactive and aggressive stealing approach that avoids GPU pipeline stalls. We also explore locality aware scheduling for subtasks written in no particular order of spatial locality. We also present a technique to dynamically group subtasks on GPU and split subtasks among CPU cores, effectively supporting a large variation in subtask sizes for efficient execution on devices of variable computing power.

The rest of the thesis is organized as follows. Chapter 2 provides more insights into the Unicorn's programming model. Chapter 3 describes Unicorn's runtime and internal implementation details at length. Chapter 4 discusses the pseudo-code implementation of matrix multiplication and two dimensional Fast Fourier Transform experiments in our system. Chapter 5 discusses the parallelizations of a few scientific benchmarks on *Unicorn*. The chapter also evaluates the performance of these benchmarks along with an overhead and stress analysis of our runtime. Chapter 6 discusses *Unicorn's* application profiling tool. Chapter 7 provide a reference to Unicorn's public header files. Chapter 8 presents related work and chapter 9 concludes the thesis.

Chapter 2

Programming Model

The *Unicorn* programming model [2] is a practical implementation of the theoretical BSP model of parallel computation. It is designed especially to simplify application programs for heterogenous clusters. More than other distributed programming models and frameworks proposed (section 8), *Unicorn* limits application's burden like task decomposition and data management. In our model, applications neither bother about data placement in the cluster nor do they handle how and when the data is transferred. Our model abstracts all types of computational devices uniformly as bulk synchronous devices. The model retains the simplicity of traditional distributed shared memory (DSM) style programming and at the same time it improves efficiency and eliminates non-determinism because the bulk synchronous semantics require only deferred synchronizations and data exchanges. The transactional address space design, along with the scheme of local views, makes multi-assign like optimizations quite efficient. Finally, the deterministic nature of our programming model omits several data hazards and deadlocks from the application code.

An application in *Unicorn* is written as a graph of interdependent tasks (Figure 2.1). Tasks communicate through and operate upon one or more address spaces. A task is eligible to execute when all its precedent tasks complete. While task graphs impart the generality to our model, the majority

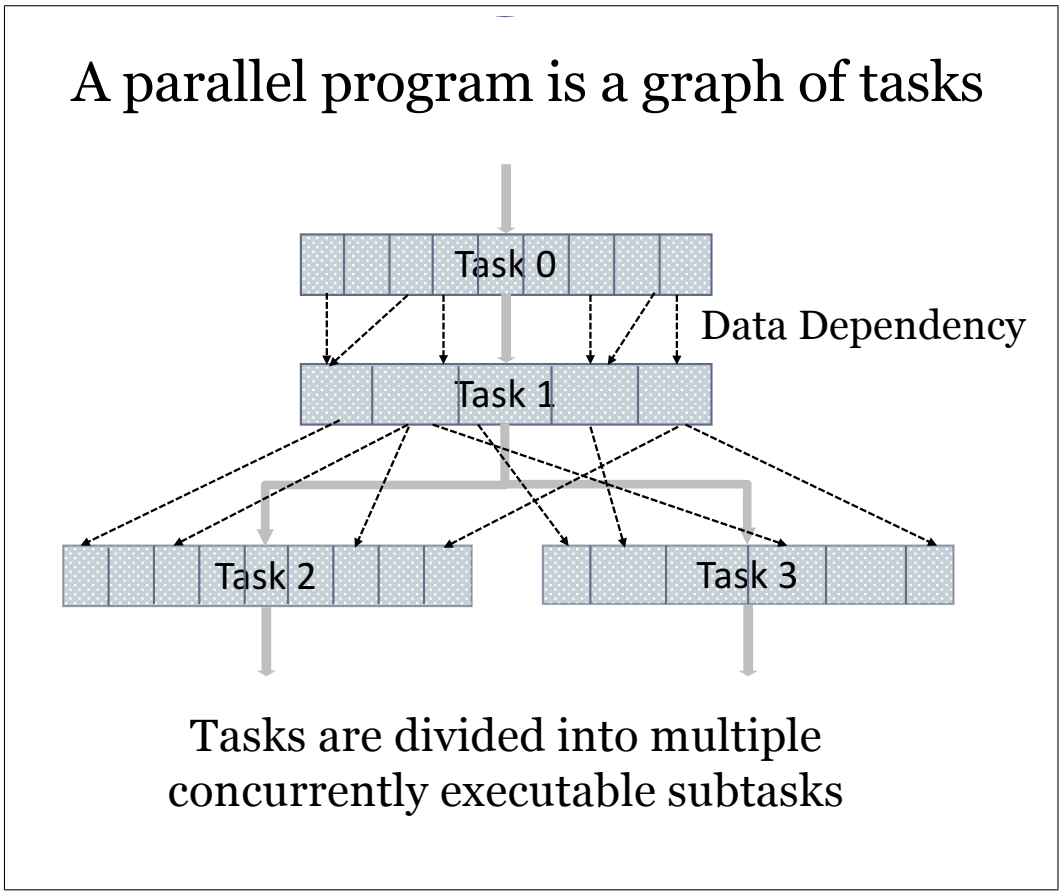


Figure 2.1: Application program in Unicorn

of the efficiency and simplicity is built within a task. It is expected that tasks are highly coarse and can be decomposed into many concurrent subtasks, which can be executed in parallel. A task is decomposed into independent subtasks that are scheduled and concurrently executed on various devices in the cluster. A task is equivalent to a BSP superstep. Similar, to the BSP model where a device operates in three phases - input phase, local computation phase and output phase, the lifetime of a *Unicorn* subtask also consists of three similar phases (Figure 2.2), each backed by an application implemented callback. A subtask is specified by the following three callback functions:

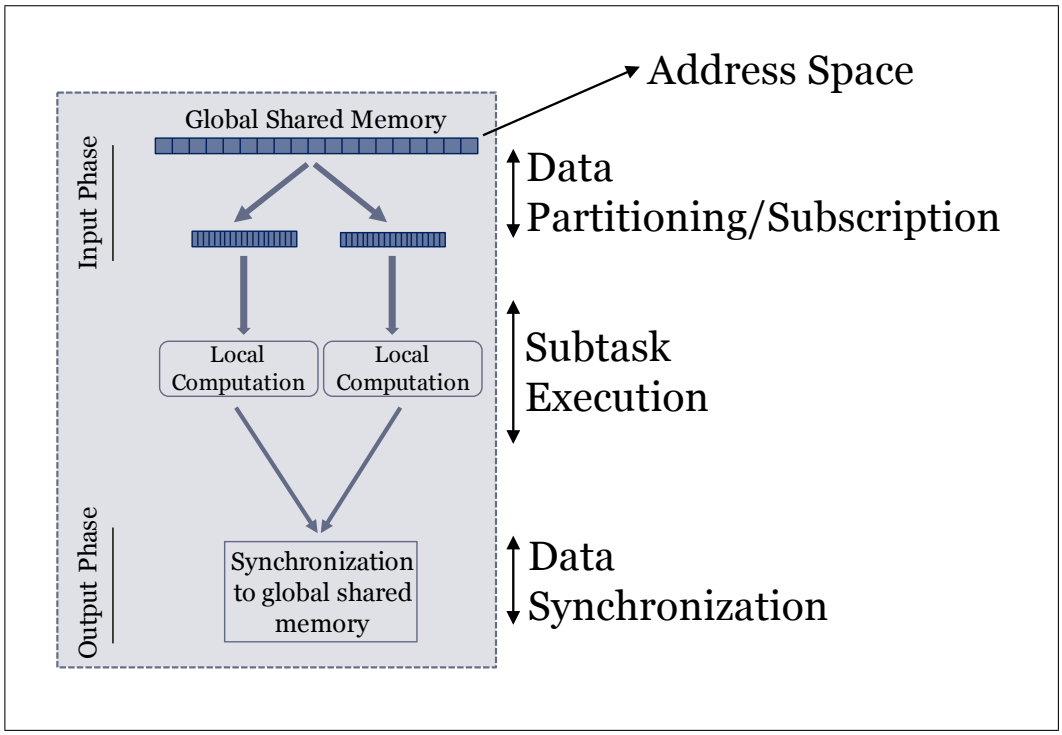


Figure 2.2: Mapping a BSP superstep to a Unicorn task

1. A *subscription* function that may explicitly specify the regions of one or more address spaces the subtask accesses. Subtask subscriptions may overlap. A trivial function may simply subscribe to entire shared spaces.

2. A *kernel execution* function that specifies the execution logic of subtasks, in SPMD fashion.

3. A *data synchronization* function that manages the output of subtasks (i.e., reduction or redistribution).

Figure 2.3 shows these three stages of the subtask and the pseudo-code of the *Subscription* and *Execution* callbacks (for matrix multiplication task) is listed in chapter 4. The subtasks of this task do not have conflicting writes and no explicit *Data Synchronization* callback is required.

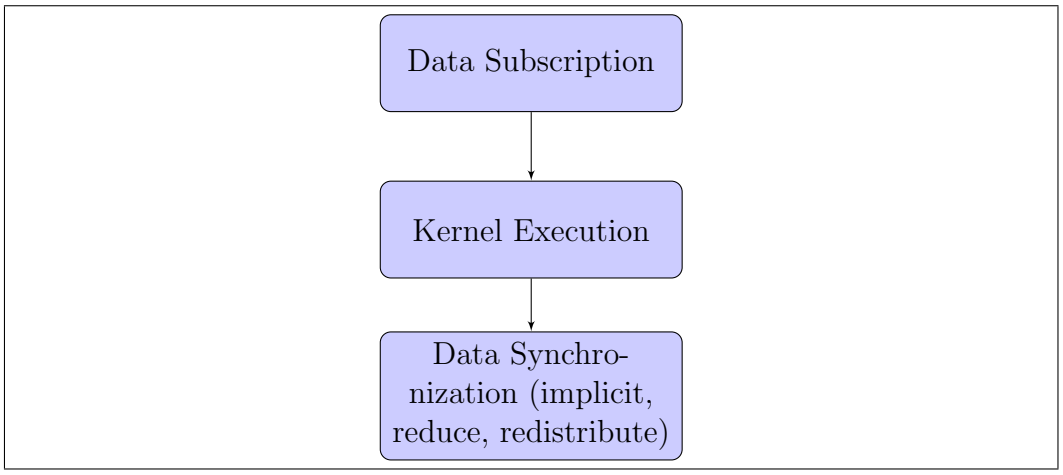


Figure 2.3: Execution Stages of a Subtask

Note that explicitly specifying subtask subscriptions is not a requirement of our model. However, due to a lack of virtual paging on GPUs one cannot automatically deduce this information at runtime without modifying subtask code. Static code analysis is also not an attractive option as its scope is limited and it restricts how kernels should be written. On CPUs, however, automatic subscription inference is performed dynamically using POSIX's [32] *mprotect* feature to protect shared address spaces. On access by a subtask, that virtual page along with a few contiguous pages, is prefetched (if necessary). With a pre-fetch of 5 pages (while multiplying two square matrices with 4K elements each on an 8-node cluster with 12 CPU cores per node), we measured this scheme to have a low overhead (10% - 15%) compared to one with explicit subscriptions. A detailed analysis of this scheme, however, is beyond the scope of this thesis.

A subtask may subscribe to multiple shared address spaces and multiple discontiguous regions within each address space. This causes memory fragmentation and leads to poor cache performance. To alleviate this problem, our model supports a notion of subscription views. A subtask kernel may

use the original global addresses (*natural view*) or a packed remapping of addresses so the subscribed regions appear contiguous (*compact view*). The compact view can save space and also yields better memory hierarchy usage on GPUs, where memory is scarce (and virtual paging is not available). For both views, a private copy of the subscribed regions of shared address space is created for each potential writer, where it accumulates writes locally. On completion of kernel execution conflicting spaces are combined. Read-only regions are also fetched and cached locally on nodes but are shared among all subscribing subtasks scheduled on that node. Note that using compact views requires the subtask kernel to be modified to use re-mapped addresses.

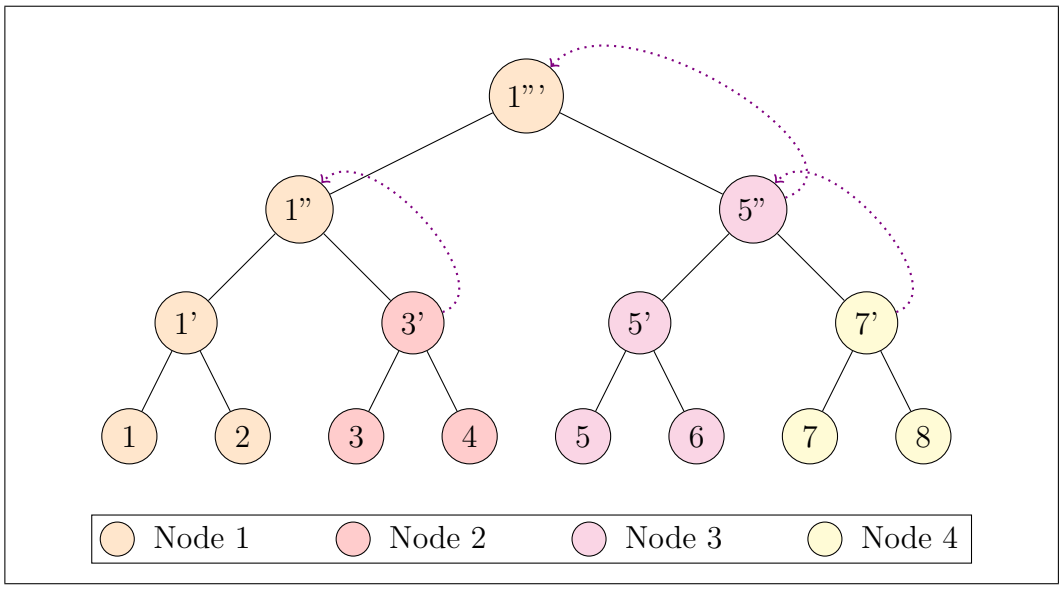


Figure 2.4: Hierarchical Reduction - leaf nodes are subtasks, others are reduced subtasks; dotted lines are inter-node data transfer

Although we support unified OpenCL subtask kernels for simplicity, better performance may be achieved using specialized kernels for each architecture type in the cluster (e.g. C++ for CPUs and CUDA for nVIDIA GPUs). When writing specialized kernels, the programmer need not obsess about the

memory hierarchy of devices any more than one might with standard CPU or CUDA implementations. This means that most existing kernel implementations can be effectively used as-is and can be debugged in sequential or single GPU setup before being used.

We allow two primitives for synchronizing the output computed by different subtasks: `reduce` and `redistribute`. The `reduce` operator allows an application-provided callback function to combine output of two subtasks; output that have been written to the same global address. Like standard reduction, we assume it is commutative and associative. Our runtime schedules reductions of the subtasks of a task in a hierarchical manner (Figure 2.4) greedily as subtasks complete. The runtime first reduces memory of subtasks executed on the same node. Once a node completes its local reductions, it is reduced with another node that also has completed its local reductions. The runtime executes inter-node reductions in a binary-tree fashion to improve parallelism and reduce data transfer.

Reduction semantics require the local copies of the same address to be combined to produce the final (or intermediate) copies. Sometimes the output of the subtasks simply needs to be collated or reordered, e.g., to scatter-gather. Performing this through reduction can be inefficient. We instead provide the `redistribute` operator. Using this operator, the application can associate a *rank* with different regions of its output address spaces. Our runtime then ensures that all memory regions with the same rank are inserted consecutively, in the order of subtask ID. Thus the regions are ordered in shared address space by rank and within a rank by the writing subtask ID. This operator can also be used to demand all-to-all broadcast or scatter-gather,

more efficiently than through individual subscription.

2.1 Data Subscriptions and Lazy Memory

We acknowledge the programming overhead in explicitly specifying data subscriptions before subtasks can access it. However, this is not a requirement of our model. Rather, limitations of virtual paging prevent automatic inference of data subscriptions. GPUs altogether lack virtual paging while CPUs are limited to the granularity of a virtual memory page.

On CPUs, automatic subscription inference is performed using POSIX's [32] *mprotect* feature to protect shared address spaces. On access by a subtask, that virtual page along with a few contiguous pages, is prefetched (if necessary). With a pre-fetch of 5 pages, we measured this scheme to be only 10% slower than explicit subscriptions while multiplying two square matrices with 4K elements each on an 8-node cluster with 12 CPU cores per node. We call this delayed on-demand loading of subscription as *lazy memory*. Since the *mprotect* feature can not protect a partial page, there could be conflicts in case two subtasks write to different portions of the same page. However, there are three ways to circumvent this limitation – the first involves padding the address spaces such that every page is exclusively used by one subtask only, the second employs page initialization by a sentinel value and a post-processing step to take final value from the subtask that has changed the sentinel and the third method is to resolve the conflict by implementing a *data reduction* callback, which we do.

Due to a lack of virtual paging on GPUs, one cannot employ a similar tech-

nique to deduce subscription information at runtime. One possible alternative is force the programmer to access GPU's global memory through some wrapper that sets a flag (on GPU) for a polling CPU thread. The wrapper makes the CPU thread fetch the required data and then DMA to GPU and in the meantime the GPU kernel is made to sleep. However, this approach is severely performance limiting. Another alternative is to infer subscription using static code analysis of CUDA (or OpenCL or PTX assembly) code. But we find its scope limited and it also restricts how kernels should be written. Due to these operational limitations with GPUs and to maintain programming uniformity, we focus this thesis on explicit data subscription by application programs. Future improvements in GPU technology may provide better solutions to this problem.

Chapter 3

Runtime System

This chapter describes the design of *Unicorn's* runtime along with the motivation for several design choices and trade-offs. The chapter also describes several optimizations that enable our programming model. *Unicorn's* runtime system can be broadly decomposed into the following components. The next few sections discuss these in more detail.

1. Device and Node Management
2. Shared Address Spaces
3. Network Subsystem
4. Pipelining
5. Scheduling
6. Software GPU Cache
7. Conflict Resolution

3.1 Device and Node Management

Unicorn consists of two major subsystems – *network* and *scheduling*. The two comprise threads that manage cluster-wide operations like subtask scheduling

and data exchange between nodes. The *scheduling subsystem* is a two-level hierarchy of threads with "scheduler thread" employing a "compute thread" for every device (i.e., CPU core and GPU) on the node. The *scheduler thread* issues commands to all *compute threads* and ensures that load among those is balanced. The *network subsystem* is a collection of three threads. All these threads serve similar purpose which is transfer of control messages and data among nodes. However, different kinds of messages are handled by different threads. One of these threads is designed for transfer of subtask data (and accompanying data compression, if any) while the other two handle the remaining messages (like task creation, address space ownership update, etc.)

All threads in *Unicorn* serve a private priority queue that contains the commands queued for it. These priority queues are revocable and commands in queue can be removed before execution. Prioritization is required for task ordering and issuing prefetch requests at lower importance than regular data fetch. Revocation is required for features like *work-stealing* (section 3.5.3) where subtask cancellation needs removal of queued commands.

Our commands are carefully designed to minimize the load on these queues. For example, one of our commands allows execution of a set of subtasks (specified as a range of subtask IDs) in one go. Our scheduler chooses the device and the set of subtasks that should run on it. If the device is remote, the scheduler requests the network subsystem to deliver the command to the concerned queue.

For inter-node communications, the commands passed to the network subsystem undergo a series of optimizations minimizing the number of MPI requests

and the volume of data transferred. These include buffering and grouping disjoint requests into larger chunks when possible, filtering out duplicate requests made by devices, and combining multiple outgoing messages to the same node into one.

3.1.1 Runtime Internals

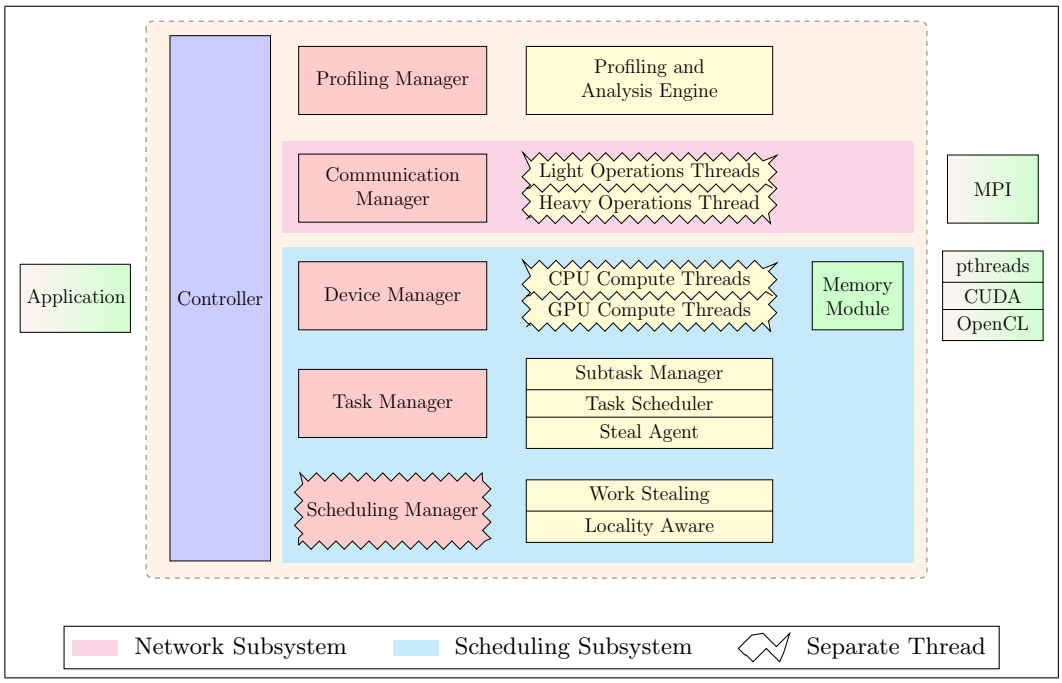


Figure 3.1: The design of the Unicorn runtime – Light orange region represents the instance of the runtime on each node in the cluster

On initialization, *Unicorn* starts one MPI process per node in the cluster. This process begins by creating an instance of *controller*. The controller manages the creation, destruction and lifetime of all other runtime components. These include *Profiling Manager*, *Communication Manager*, *Device Manager*, *Task Manager* and *Scheduling Manager*. The last three modules are part of the *scheduling subsystem* and the *Communication Manager* comprises the *network subsystem*. The controller is also the interface between the

application code and the *Unicorn* runtime. All application requests like creation and submission of tasks, creation and destruction of address spaces, etc. pass through the controller. The runtime also executes user space callbacks under the sandbox of the controller.

Once the controller initializes other runtime components, they directly talk to each other without the controller's involvement. All modules store diagnostic information with the *Profiling Manager*. The information collected by the *Profiling Manager* on all nodes is accumulated on MPI master node at application shutdown. This data may be dumped to stdout/stderr or may be sent to *Profiling and Analysis Engine* for further processing. More information on the kinds of analysis performed by the engine is present in chapter 6. The engine also converts the data into readily consumable graphical and tabular formats. This information is mostly used for performance debugging and diagnosis.

Task and subtask execution and all communication in *Unicorn* are carried out asynchronously. For this reason, separate communication, scheduling and device management components exist in the runtime. To support asynchrony, these components accept commands with priority levels from other components and execute them in dedicated threads. Being asynchronous is also essential for optimizations like pre-fetching and for task pipelines that overlap computation of subtasks with communication of others.

The *Communication Manager* carries out all inter-node communications over MPI. All control messages (generated by the runtime) and all data transfers (requested by executing subtasks) destined for remote nodes are routed through the *Communication Manager*, which filters duplicate requests made

by various subtasks and also combines multiple requests targeted for same destination node, whenever possible. Requests that involve subtask data transfer are classified as *heavy operations* and are handled by a separate thread. All other requests are *light operations* and are categorized into two types – fixed size requests and variable size requests. Fixed size requests are the ones whose data size is pre-known and special MPI optimizations that re-use the same data buffers repeatedly are employed. Variable size data requests have varying lengths and buffers are not pre-allocated for those. Rather they are handled using *MPI_Probe* in a separate thread. Segregating heavy and light operations also allows critical control messages in the runtime to be transferred quickly. This indirectly helps in keeping most runtime threads active rather than waiting on a few commands.

The *Device Manager* handles all CPU cores and GPUs on a node. For each device, it creates a dedicated *compute thread*, each of which is backed by an exclusive priority queue. All commands that are targeted for execution on a device are enqueued in the corresponding priority queue. Each *compute thread* has a helper *Memory Module*. For the CPU compute threads, the *Memory Module* manages sharing of read-only address space data and creation/destruction of virtual memory for local working copies of the address space subscriptions of the subtasks. On the other hand, the *Memory Module* for the GPU *compute threads* employ a software LRU cache for efficient sharing of scarce GPU memory between subtasks. It also manages the creation and destruction of pinned memory buffers used for bidirectional DMA transfers between CPU and GPU. CPU *compute threads* use pthreads underneath while GPU ones use the CUDA runtime. Both may optionally use

OpenCL. We also explored using different processes (instead of threads) for managing CPU and GPU devices. This helps sandboxing application code but the performance implications of the approach render it impractical.

The *Task Manager* enqueues each task submitted by the application and submits them for execution (to the *Scheduling Manager*) when all its dependencies are fulfilled. This is accompanied by creation of a *Subtask Manager*, *Task Scheduler* and *Steal Agent*. The *Subtask Manager* keeps track of the subscriptions and data transfers of each subtask executing on the node. It also tracks the subtask execution times, which helps estimate the relative execution rates at different cluster devices. This also helps in determining if a subtask is straggling and if it should be multi-assigned to some other device. The *Task Scheduler* co-operates with the *Subtask Manager* and the *Scheduling Manager* for executing a subtask and collecting its acknowledgement. The *Steal Agent's* role is limited to directing an incoming steal request to the device with the highest load. The purpose of *Steal Agent* is to help reduce the number of steal attempts in the cluster.

The *Scheduling Manager* handles scheduling of all tasks submitted by the application. Like *compute threads*, the *scheduling Manager* has a dedicated thread backed by an exclusive priority queue. The *Task Scheduler* of every task enqueues subtasks in the *Scheduling Manager's* queue and it schedules them for local or remote devices (using the *Communication Manager*). When a device finishes execution of its subtasks, an acknowledgement is sent to the corresponding *Task Scheduler* through the *Scheduling Managers* on various nodes communicating via their respective *Communication Managers*.

The default *Unicorn* scheduler is locality-oblivious and based upon two-level

work stealing assisted by *Steal Agents*. However, a task may opt for locality aware scheduling, in which case data locality (on cluster nodes) is incorporated into scheduling decisions. Having a different system-wide *Scheduling Manager* and a *Task scheduler* per task allows *Unicorn* to employ different scheduling policies for different tasks running at the same time. However, the common functionality is abstracted out into the system-wide *Scheduling Manager*.

The next section lists the threads created by *Unicorn's* runtime on every node in the cluster. Subsequent sections of this chapter talk about the runtime in greater detail and also explain the functionalities and design of the *Network subsystem* and the *Scheduling subsystem*.

3.1.2 List of Threads

Unicorn runtime creates the following threads on each node in the cluster:

1. Fixed size network operations thread
2. Variable size network operations thread
3. Heavy network operations thread
4. Scheduler thread
5. One thread per CPU core or GPU device on the node

All our threads are light-weight and we do not dedicate any CPU core to any of these. Rather, we let the management threads co-operate with device

threads running application code. The observed overhead of these management threads is small. For example, the image convolution experiment (chapter 5) has a measured overhead of 0.2% of its total execution time.

3.2 Shared Address Spaces

Our runtime provides distributed shared address space to the application. *Unicorn* tasks bind to one or more address spaces for input and output. This is specified with access flags like *read-only*, *write-only* or *read-write*. Address spaces can be used exclusively by tasks or they may be shared. Two examples of address space sharing among tasks are – an output address space of a task may serve as input to the next task, or an address space may contain constant data serving as input to a series of tasks.

Address spaces can be thought of as an equal (and contiguous) allocation of virtual memory on all intended nodes in the cluster. This allocation on all nodes is equivalent and exists to provide a logical view of distributed memory. The address spaces are distributed in the sense that generally their entire data is not resident on any particular node in the cluster but distributed over several nodes.

Subtasks (of tasks) operate upon the bound address spaces by means of subscriptions. *Read* subscriptions define address space bytes required for computations done by the subtask and *write* subscriptions define address space bytes produced by the subtask (as output). These bytes can be a single contiguous region of memory within the address space or these can be a set of uniformly or randomly distributed contiguous memory regions within the

address space. For this reason, *Unicorn* allows a subtask to register multiple subscriptions with the runtime (section 2). Note that each of these subscriptions can be concisely specified using the quad tuple (offset, length, step, size). This format enables applications to specify generally used contiguous, block based and strided subscriptions in a few calls, which also directly reduces the number of subscription calls the runtime processes per subtask.

As tasks execute on address spaces and their subtask subscriptions are processed, address spaces are logically fragmented into memory regions defined by these subscriptions. Internally, address spaces store these logical fragmentations (called *data regions*) in the quad tuple format (offset, length, step, size). Each of these *data regions* have an associated *owner node* in the cluster and this *owner node* notionally contains the entire data corresponding to the *data region*. The mapping of *data regions* to corresponding *owner nodes* is stored in a directory called the *address space ownership directory*, or simply the directory.

An application may create any number of address spaces from any node in the cluster. The node on which an address space is incarnated is called its *creator node*. Internally, *Unicorn* also assigns a *master node* to every address space created by the application. As explained below, the *master node* serves as an intermediary to enable efficient management of the directory. *Master nodes* are chosen in round robin fashion to balance the load of all address space routing requests among all cluster nodes. The master node contains the master copy of directory for a given address space. Other nodes using that address space generally contain a partial map: a subset of *master ownership directory*. If these other nodes require a region that exists in their partial

map, they directly send region fetch requests to the corresponding owner. On the other hand, if a region is not present in the partial map, they route address space fetches through the master node. The master then consults its ownership directory and forwards the request to the actual owner containing the data region in question.

When an address space is created, it contains a single region. The ownership directory on all nodes is updated to map that single region to the selected master. As a subtask executes and commits its writes (originally in its private view) to the address space, the node on which the subtask was executed becomes the new owner of the data regions write-subscribed by the subtask. (For multiple writers, the final owner is where the final reduction occurs). At the end of a task, the new owner node records this ownership in its directory and also sends this ownership update to the master. If the master observes a new owner node, it updates its own map and forwards the ownership update message to the previous owner(s), which also updates its map. All other nodes always initialize their maps to point to the master at the end of the task. Thus, at the beginning of a task all nodes map to themselves data regions they own and point to master for all other data regions. The master, however, always knows the true locations of all data regions in the address space.

Unicorn tasks guarantee transactional semantics. This means that address space ownership updates are reflected only at task boundaries and all subtasks of a task see the same address space data during task execution. In other words, even for an address space marked *read-write*, where a few subtasks are updating the address space while a few are reading it, the ones that

are reading must not see the new data but read the state at the inception of the task. Our address spaces achieve transactional semantics by deferring the address space ownership updates till the end of the task. Since ownerships are not modified during task execution, all subtasks continue to refer to the original data location until task boundary. The following sequence of steps define our delayed ownership update semantics –

1. All nodes hold information about write subscriptions of all locally committed subtasks in a local data structure. This information is a set *'S'* of quad tuples (offset, length, step, size).

2. At the end of the task, all nodes commit *'S'* into their *address space ownership directories* and also send an ownership update message to the *master node*. This message also carries the set *'S'*.

3. The master node commits the received set *'S'* from every other node. While commiting, it records the *data regions* that have changed owners from last time and sends them another ownership update message.

4. All non-master nodes process the ownership update message from the master and record master as the new owner of the *data regions* in the message.

Note that we maintain a separate set *'S'* for every address space master node. In other words, if a task employs three address spaces with two having the same master node and the third having a different master node, we maintain two such sets, one for each master. In case all address spaces used in a task share the same master, only one set *'S'* is maintained and transferred.

Often, read subscriptions of subtasks (of a task) overlap. In such cases, it is prudent that data once fetched (by a subtask) must not be re-fetched upon request by another subtask. To accomplish this, our runtime records all data fetched on a node in a separate address space directory. This directory is temporary as its lifetime is bound to that of the ongoing task and is thus called *temporary ownership directory.* At inception, the temporary directory is a logical replica of the *address space ownership directory* on every node and as data is fetched, it records the updates. For every read subscription request from a subtask, the temporary directory is the one consulted and not the main directory. This ensures that no data is transferred again during the lifetime of the task. Note that this mechanism also conforms to the transactional semantics guaranteed by our runtime. It further allows several nodes to be simultaneously designated temporary owners of a data region without the need of an explicit handshake (or ownership update message) for data sharing.

At the end of the task, the data fetched for reading (and recorded in temporary directory) may become stale as data ownerships could have changed and the new data owner could have written new data in that region. Thus, at task boundaries, the temporary directories are simply discarded. However, if a task does not write to an address space (and subscribes to it for reading only), as an optimization its temporary directory entries are retained for the subsequent task.

Figure 3.2 shows an address space and states of its *address space ownership directories* and *temporary ownership directories* as it flows through two *Unicorn* tasks on a sample three node cluster. In the initial state, the address

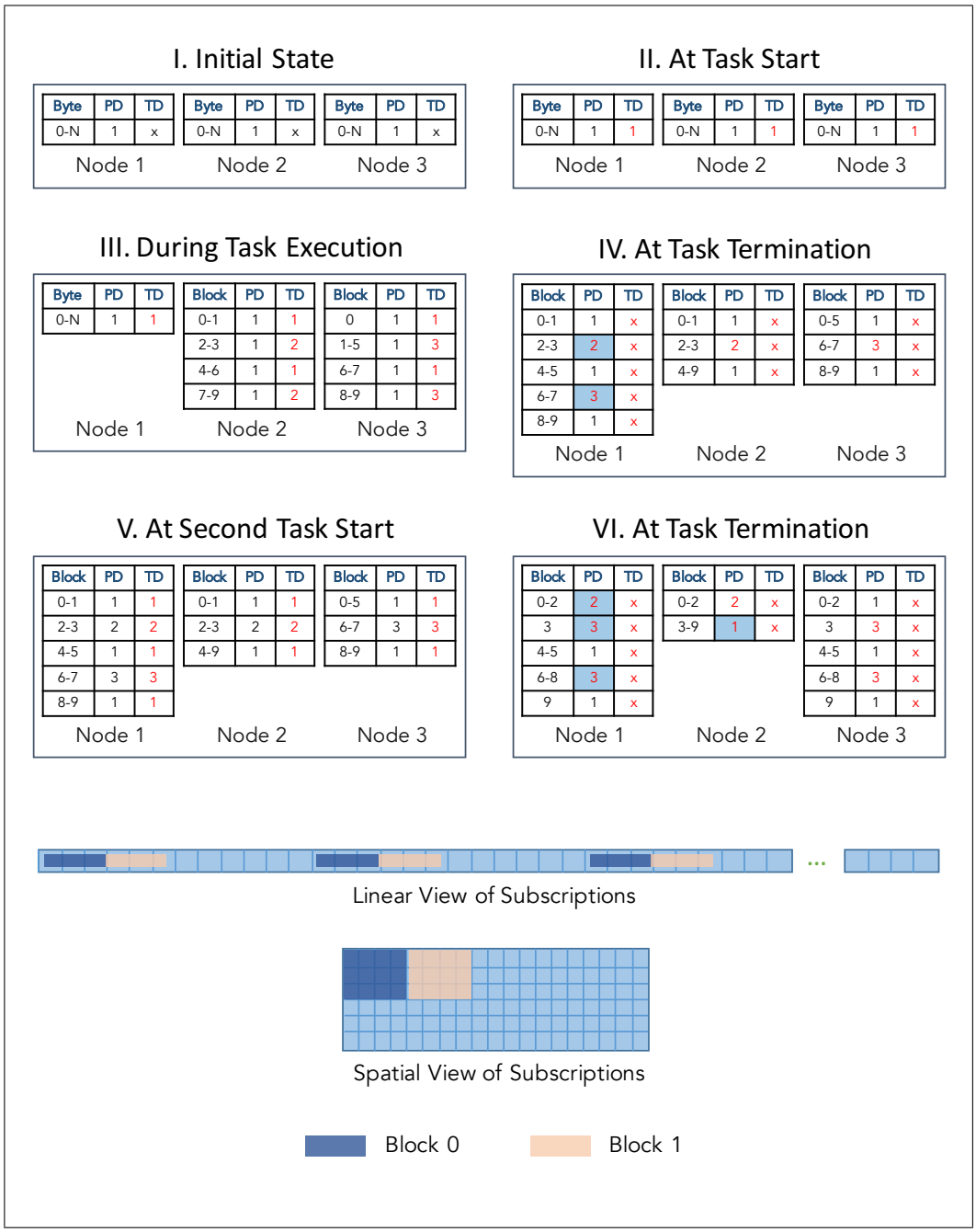


Figure 3.2: Address Space Ownerships – PD represents Address Space Ownership Directory, TD represents Temporary Ownership Directory, x represents non-existent directory, red text represents changes from last state and cells in light blue background represent directory changes via explicit ownership update messages; Node 1 is the address space master node.

space *master node* (Node 1) owns the entire data. All directories point to Node 1; temporary directories are null. In state II, a task starts and the temporary directories on all nodes are logical replicas of the corresponding main directories. In state III, Node 2 and Node 3 fetch data from Node 1 (for task execution) and the data brought in is recorded in their temporary directories. The task terminates in state IV and subtasks that have executed on each of the nodes commit their updates into the corresponding directories. Noticing a change from previous state, Node 2 and Node 3 send ownership update messages to Node 1 ands delete the temporary entries. In state V, another task starts and a new temporary directory is instantiated on each node. This task executes and upon termination commits writes to directories in state VI. Again, Node 2 and Node 3 send ownership update messages to Node 1. Upon receiving this message for block (or *data region*) 3, Node 1 notices a change from previous ownership (Node 2) and accordingly sends another ownership update message to Node 2. Note that *Unicorn* consolidates all ownership updates to be sent from a node to another and sends a single unified message.

To theoretically analyze the overhead of our ownership update protocol, assume a task executing on an N node cluster and using K address spaces $K > N$. Our master selection mechanism ensures that every node is master of at least one address space. Lets also assume that subtasks on node j make W_{ij} write subscriptions for address space i. This means the total number of write subscriptions in the cluster is equal to

$$M = \sum_{i=1}^{K} \sum_{j=1}^{N} W_{ij}$$

Updating the ownerships at the first instance would mean sending (or per-

haps broadcasting) at least M ownership update messages in the cluster. However, by delaying these messages to task boundaries and by sending only one ownership update message for all address spaces having the same *master node*, we significantly reduce the number of ownership update messages generated in the cluster. In this example, every node being master of at least one address space receives *N-1* ownership update messages and every master sends out a similar number of ownership update messages (in the worst case) to non-masters. Thus, the maximum number of ownership update messages possible with our protocol is *2N(N-1)*, which is significantly fewer than M.

Recall that *Unicorn* subtasks are also allowed to have overlapping write subscriptions. In this case, subtasks are reduced by an application defined reduction operator. This reduction is a commutative and associative operation and our runtime internally decides the node which becomes the *final owner* of the overlapping *data region* post reduction. In this case, we do not send any explicit ownership update at task boundary. Rather all nodes (except master and the final owner) silently reset the new owner to the *master node* whereas the *master node* and the *final node* record the *final node* as the new owner.

Besides transactional semantics, our address spaces are designed for efficiency. Our design ensures that no explicit ownership update messages are required during task execution. Only at the end of the task, a few such messages may be exchanged when the ownership of any data region changes. To further limit the number of these messages, we aggressively reduce address space fragmentation by combining adjacent directory records whenever possible. Specifying subscriptions using block regions, further helps in mini-

mizing the number of directory entries. *Unicorn's* runtime also ensures that all ownership updates of an address space (at a task boundary) have finalized before a subsequent task starts on any particular cluster node.

Our runtime implements *address space ownership directory* using an r-tree [37] based map shared by all subtasks. Its key is the said quad tuple and the value is the *owner node*. On a read query for a particular *data region*, the corresponding quad tuple is looked up in the r-tree, which returns the set of all overlapping quad tuples (along with their *owner nodes*). Each quad tuple in this set is intersected with the *data region* in question and the data fetch request of the intersecting regions is sent to the corresponding *owner nodes*. If the *owner node* is the master node and it does not have the required data, it re-routes the request to the node having the data, which now directly sends data back to the requestor without involving the *master node*.

On the other hand, if an update needs to be recorded in the r-tree before the new *data region* is inserted in it, all its overlapping quad tuples are first deleted, the data region is subtracted from their union and the result and the new region, both inserted. Before the insertion of any quad tuple, a query is made on its four boundaries for the existing quad tuples. with the same owner nodes as for the one being inserted. Any adjacent quad with the same owner node (as the value to be inserted) is removed from the r-tree and a unified quad is inserted.

3.3 Network Subsystem

Most distributed systems are bottlenecked by communication. Optimally transferring data over the network is critical for sustained performance in a cluster computing environment. Our network subsystem is optimized for performance and works closely with *compute threads* and *address spaces*, merging duplicate and overlapping requests, reducing both request and returned data message counts and sizes. Besides, subtask data fetch requests are treated at high priority while the prefetch requests for subtasks anticipated to run in future are executed at a lower priority. Our network subsystem is also capable of compressing outgoing data and loss-lessly decompressing the same at the recipient. This is especially important for large data transfers that are otherwise too slow.

Recall that *Unicorn* runtime decomposes a task into multiple concurrent subtasks. These subtasks may share part of their input data and thus may subscribe to overlapping regions of input address spaces. *Unicorn* allows subtasks to specify subscriptions as (offset, length) pairs or using quad tuples (offset, length, step, count). The latter form is easy to use for strided or block subscriptions. For each subtask subscription, our network subsystem queries the address space directory for the corresponding data location in the cluster. The query response may point to a local data location if the data is resident locally on the node. In case the data is not locally resident, it may already be enroute due to an earlier request or a new request must be initiated. In either case the memory module returns a unique handle (internally implemented like pthread signal-wait) on which the calling *compute thread* can wait till

the data arrives.

The network subsystem initiates a remote data fetch against this handle if the request does not fully overlap with previous requests. The r-tree is again used for overlap detection. When the remote data fetch completes, the handled is notified and the waiting thread signaled. Note that the address space query and handle creation happen atomically. This is done to ensure that any other *compute thread* requiring the same data is returned the same handle to wait on. In a typical scenario, a *compute thread* may wait on several such handles and a handle may be waited on by several *compute threads*. In our implementation, a subtask first issues all subscription requests and collects all the remote handles it depends on and then an aggregate handle is created which it actually waits on. The aggregate handle is unique to a *compute thread* while the internal handles may be shared by several *compute threads*.

Handles created for remote data fetch for a subtask may be destined for one or more nodes. Our network subsystem reduces latency of these network requests by grouping all handles destined for the same remote node into one underlying request on the network. The response, however, is separate for different handles as this helps faster awakening of *compute threads* waiting upon a subset of handles.

In addition to this, system latency is also reduced by piggy-backing several control messages over other messages. For example, address space ownership update messages (section 3.2) are often combined with subtask completion acknowledgements. This prevents network congestion and makes way for other high priority messages.

Unicorn tasks with conflicting writes (to output address spaces) are required

to define a *data reduction callback* (chapter 2). Subtask subscriptions in such tasks are often large and may be entire address spaces altogether. Transferring such large address spaces (for reduction) over the network is generally detrimental to system performance. In such cases, our network subsystem compresses the address spaces before sending them across to other nodes. Refer to section 3.7 for more details on this.

Our network subsystem internally employs three threads – one for *fixed size transfers*, the second for *variable size transfers* and the third for *heavy operations*. The need for separate thread for fixed and variable sized data transfers arises from MPI (our network subsystem's backbone). While the former ones are optimized by allocating the recipient buffer (of known size) in advance and using persistent MPI commands repeatedly, the latter ones are received using MPI_Probe and post allocation of receiving buffer. We find that a separate thread for MPI_Probe works better. An example of variable size data transfers in our runtime are *ownership transfer messages* which may have a variable number of entries every time they are sent. These threads are typically used to handle runtime's control messages and many other task execution messages. Hence we need to maintain a fast throughput on receiving and handling such messages. As a result, messages which are expected to take longer are delegated from these threads to a separate *heavy operations* thread. These messages include subtask memory transfers, reduction memory transfers (and associated compression/decompression), etc.

3.4 Pipelining

Overlapping computation with communication is critical to hiding heavy data transfer latencies for subtasks. Data is transferred at two stages in our runtime – the first is the transfer of data among nodes and the second is the transfer of data from CPU to GPU. We intend to hide both these data transfer costs by deploying subtask pipelines.

Our scheduler (section 3.5) assigns a set of subtasks to a device. These subtasks are usually executed in the given order. Thus, while executing any particular subtask, a device knows the next subtask it is likely to execute (modulo steal). The device, thus, issues a communication request for the next subtask when it starts executing any subtask. This behavior ensures that while the devices are busy executing subtasks, the network does not become idle and fetches data for future work.

Similarly, every GPU in the cluster is kept busy by employing a pipeline of subtasks. The data is moved in and out of these GPUs along with the execution of one or more GPU kernels. Since data transfer to/from GPUs requires DMA (and pinned memory), our runtime allocates and re-uses pinned buffers for fast movement of data in and out of GPUs.

Besides pipelining, the bulk synchronous nature of our programming model (that leads to the transactional design of our address spaces) plays a significant role in hiding data transfer latency. The transactional design guarantees that all subtasks (of a task) see the same input data (at any given location) for the lifetime of the task. This enables us to use simple coherence protocol where no address space ownership update messages are generated during task

execution. Whatever data is pre-fetched to a node, remains valid throughout the task, precluding the need for any further on-demand transfers for any subtask that may need that data. This no-invalidation system helps increase the percentage of time that network carries data messages (as opposed to control messages for system correctness). The ability to have somewhat coarse-grained tasks also helps reduce communication overhead.

3.5 Scheduling Subsystem

Unicorn's scheduling subsystem is at the core of its performance. It is both a proactive and reactive system. A number of optimizations are built into it to ensure that appropriate scheduling decisions are taken in the first place. However, device/network load as well as application characteristics are variable. Our dynamic scheduler adapts to the situation efficiently. Results in chapter 5 highlight the importance of such a dynamic scheme.

Our scheduler is work-stealing [44, 23] based and load is balanced among devices by stealing unexecuted subtasks. We observe that the usual design generally suffices when devices with uniform computing capability are involved in a computation. However, modern cluster environments have heterogeneities in memory hierarchy and computing power of devices, network throughput of nodes, device architecture (x86_64, Tesla, Fermi) and programming styles (OpenCL/CUDA/C++), etc. Due to these variations, devices tend to differ a lot in the effective throughput delivered for a particular computation. In our experiments, we measured GPU performance higher than CPU performance by more than an order of magnitude sometimes. In our tests, a BLAS

[24] based sequential implementation of multiplication of two dense square matrices with 8K elements each (on Intel Xeon X5650) was outperformed by a CUBLAS [17] based implementation (on Tesla M2070) by a factor of 33x+. Similarly, a BLAS based block LU-Decomposition on the GPU reported 10x+ speedup over the sequential implementation. This disparity is further aggravated in a cluster environment if the data is placed near faster GPUs while the slower CPUs are on a remote node away from the data.

Scheduling subtasks across devices of widely different capabilities presents its own challenges. *Unicorn* normally uses one cluster device as a computing unit. However, to reduce disparity, a set of slow devices (usually CPU cores) are made one computing unit (or work-group). Equivalently, a subtask is now envisioned as a set of work-items and a single work-item is scheduled per CPU core. The number of devices in a computing unit are dynamically calibrated to yield the maximum performance. This work-group calibration is performed locally on all nodes and all CPU cores on a node participate in the formation of work-groups.

We also extend work stealing to account for subtask pipelining and underlying device topology. To summarize, *Unicorn* employs a pipeline aware work stealing scheduler accompanied by dynamic calibration of CPU work-groups. The next few sub-sections describe these in detail, along with a scheduling reconsideration technique, called *multi-assign*, for handling stragglers.

3.5.1 Work Stealing in Unicorn

The benefit of dynamic scheduling is that it responds not only to varying subtask load and compute unit capacity but also to varying network throughput.

Unicorn scheduler begins by assigning an equal number of subtasks to each device. Once a device nears completion, it steals subtasks from another device. We employ two-level random stealing, where a compute unit first attempts a local steal. On failure, it requests the steal-agent on a randomly selected remote node, which in turn attempts to steal from its local units. Although one might extend shared-memory algorithms to stealing over shared address spaces or one sided MPI communication [23], we find the overheads too high for that. On the other hand our runtime is already based on multiple asynchronous threads on each node and it is easier to use that runtime infrastructure for load stealing without incurring loss in compute threads.

Recall that our pipelined schedule implies that the subtasks may be

1. idle,
2. waiting for data,
3. ready for execution, or
4. executing.

Our lock-free steal algorithm allows steal from any stage from idle to ready for execution. (Steals from stage 4 is allowed as a special case and is discussed later in section 3.5.3). The steal-agent on each node helps accomplish this as it maintains the count of outstanding subtasks in each stage for all local devices on its node and also monitors their execution rates. (Since we do not model subtask heterogeneity, we simplify by using the past rate as a predictor for future rate.) Thus, our two-level victim selection technique proceeds as follows for device d_{ij} on node i:

1. In first attempt set victim $v = i$, otherwise, choose random victim v, different from victims selected in this round. (Round resets when a victim is found or no potential agents remain.)

2. Send steal request to (the steal agent of) node v. The steal request includes the measured subtask execution rate, R_{ij} of d_{ij} and its aggression level a_{ij}, a value in [1:3]. In our setup, we keep GPUs at maximum aggression level (i.e. 3) while the aggression of CPUs increases with the number of consecutively failed steal attempts. Initially, CPUs start with aggression 1 and reach the maximum level at $N/2$ consecutive steal failures, where N is the number of nodes executing the task. Note that all devices in the cluster record the number of steal requests issued by them and their outcome (success or failure).

3. Agent v, selects device d_{vw} that is expected to complete its queue the last among all devices of node v. (i.e., the highest value of $\frac{Q_{vw}}{R_{vw}}$, where Q stands for the length of the queue at the given device). Agent v considers subtasks in stage k only if $a_{ij} >= k$.

To further reduce the number of steal attempts in the cluster, a range of subtasks is stolen. We compare the devices' execution rates to decide the quantum of the final steal: $Q_{vw} \times \frac{R_{ij}}{R_{ij}+R_{vw}}$. A task terminates when all its device queues are empty.

In contrast, a one level stealing algorithm, where a device directly steals from another randomly selected device, generates too many steal requests and is not scalable. For example, if we assume N nodes and D devices per node, and that at a time t, every device has a probability p_t of having an

empty queue. The probability of a one level random steal attempt being successful is $\frac{1-p_t}{N \times D}$, while that of steal from a random steal agent is $\frac{1-p_t^D}{N}$. Our experiments described in chapter 5 also demonstrate this improvement. We do not see the benefit of a deeper hierarchy (e.g. multiple steal-groups on a node [40]) due to its management overheads. However, for larger clusters it is possible that organizing nodes into hierarchical steal-groups will be useful.

3.5.1.1 ProSteal

As described earlier, *Unicorn* has all devices in the cluster aggressively overlap their current subtask computation with data transfer of the next potential subtask. Various subtasks incur different data transfer latencies as the data may already be locally available for some, while others may have to fetch from one or more remote nodes. When a device turns idle and starts a steal operation, it suspends its pipeline as it has no subtask whose data can be fetched. After the stolen subtask arrives, the device restarts its pipeline by initiating the stolen subtask's data transfer. Had this subtask not been stolen, its communication might have overlapped another subtask's execution at the victim, which could have potentially completed the work faster. In addition to this, GPU stealers suffer another pipeline hazard as they lose the opportunity to overlap bi-directional data transfers between the device and host CPU while they steal.

Figure 3.3 shows an example of a victim's pipeline. The first three subtasks execute normally – the computation of the first subtask gets overlapped with the communication of the second and the computation of the second gets overlapped with the communication of the third. However, subtask

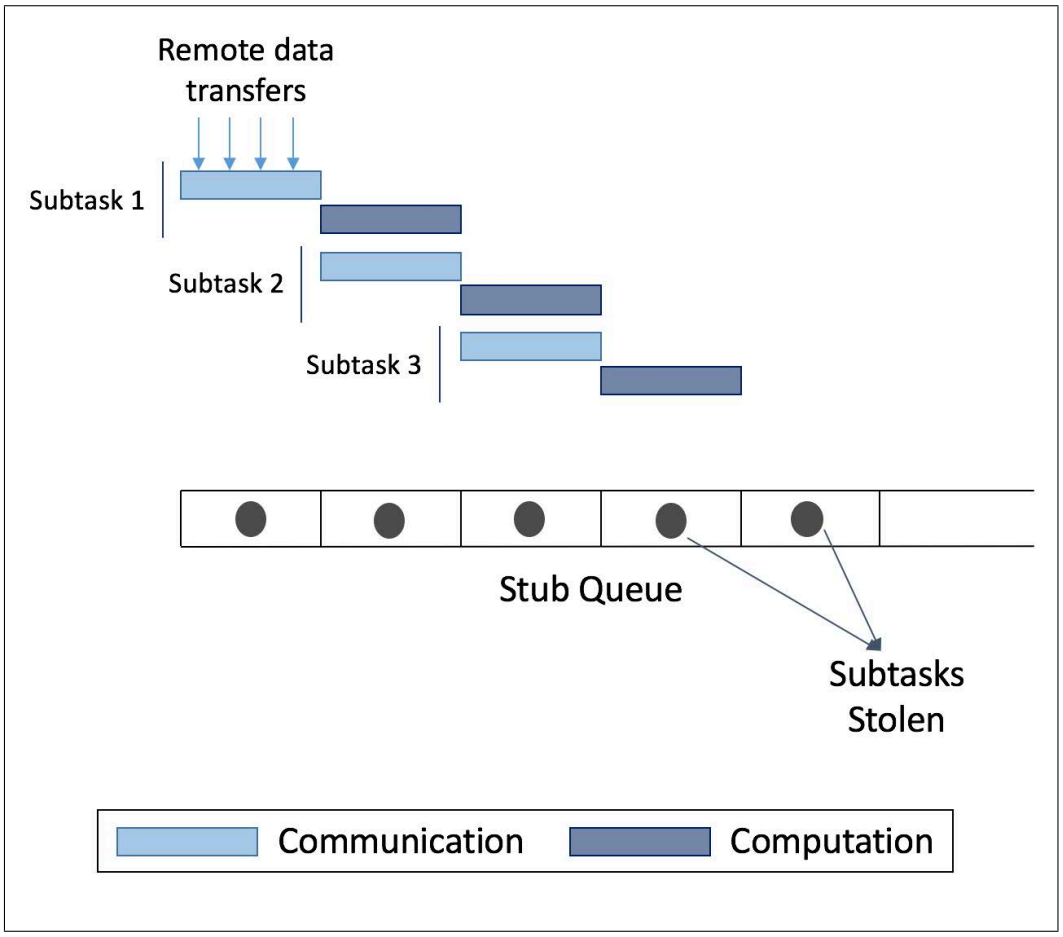


Figure 3.3: Loss in victim's pipeline due to work stealing

four and five get stolen and there is no subtask left in the *compute thread's* queue, whose communication could overlap with the computation of the third subtask. This causes a stall and subsequent performance loss in the victim's pipeline. Note that the stealer has initiated the steal operation as its pipeline has already stalled. Thus, a steal operation may cause a stall in both stealer's and victim's pipelines.

Unicorn addresses this problem using a novel aggressive stealing approach, called *ProSteal* where the stealer does not wait for its pipeline to exhaust. Rather it steals with one or more subtasks still in the queue. Stealing early

helps the stealer to overlap the stolen subtask's communication with the ongoing computations at its end. It is important to initiate the steal request at the appropriate time as stealing too early may increase the number of steal attempts and harm application performance. The optimal number of outstanding subtasks when a device steals is computed as $R_s \times (L_s + L_d)$ where R_s, L_s and L_d, respectively, are rate of subtask execution, steal latency and data fetch latency of the stealer. In other words, the longer it takes to prime the pipeline after flush, the more aggressive the stealer is. For CPUs, which generally have a slow rate of subtask execution in comparison to GPUs, the expression results in relatively less aggressiveness. Note that this aggressiveness is different from the one described in section 3.5.1, where the aggressiveness is actually a measure of desperation (and increases with the number of consecutive steal failures) and affects the response of the requested steal agent. On the other hand, *ProSteal's* aggressiveness is a measure of performance and determines how soon the stealer should initiate the steal request.

A pipeline is perfect if it spends the same time in all its stages. Assuming that our two stage pipeline (with communication being the first stage and computation being the second stage) is perfect, we can obtain a theoretical lower bound on the time lost in pipeline stall as $L_s + T_p$ where L_s is the steal latency and T_p is the time spent in any pipeline stage. However, in case both stages are not uniform the time lost in pipeline stall will have an upper bound of $L_s + max(T_{s1}, T_{s2})$ and a lower bound of $L_s + min(T_{s1}, T_{s2})$ where T_{s1} is the time spent in the first stage of the pipeline and T_{s2} is the time spent in the second stage of the pipeline. Using a value greater than $max(T_{s1}, T_{s2})$

for L_d may lead to stealing too early and bringing in newer subtasks with a few already in the stealer's pipeline. This may cause throttle and ping pong of subtasks (as the newly brought in subtasks, being pending, may be stolen again). On the other hand, using a value smaller than $min(T_{s1}, T_{s2})$ for L_d may be too small to prevent a pipeline stall. Thus, we use the average of both T_{s1} and T_{s2} as the value of L_d. This yields good performance in our experiments.

With *ProSteal*, we attempt to minimize the time lost in pipeline stall when a GPU device empties its queue. It is difficult to predict L_s, T_{s1} and T_{s2} values for any particular subtask. Thus, for computing these numbers, every *Unicorn* device uses the running average of the subtasks it has previously executed. Secondly, we issue steal requests at subtask boundaries and not in the middle of the pipeline. For this reason, we multiply our expression $L_s + L_d$ by the rate of subtask execution R_s. [3] presents a more comprehensive analysis of this scheme. Results show that despite our approximations to the theoretical bound, we avert GPU pipeline stalls by 80% (when using ProSteal versus when not using it) on an average computed over the experiments presented in section 5.3.1.2.

3.5.1.2 Locality-aware Scheduling

To reduce data fetch bottlenecks, *Unicorn* requires all subtasks of a task be implemented with one dimensional or two dimensional spatial locality. This means that consecutive subtasks access consecutive memory locations. However, if a *Unicorn* task is implemented with little or no spatial locality among subtasks, the application programmer can enable *Unicorn's* locality-

aware scheduling. When used, our locality-aware scheduler enriches scheduling decisions by maximizing the re-use of locally resident data on nodes or by minimizing the cost of fetching non-local data.

To maximize reuse of locally resident data on nodes, we compute an affinity score of every subtask to every cluster node and possibly schedule subtasks on nodes with high affinity towards them. To build such an affinity table (subtasks versus nodes), we run an internal *Unicorn* task that computes this information by analyzing address space directories on all cluster nodes. We explore affinity score based on different parameters like maximize re-use of locally resident data, minimize cost of transferring remote data, and minimize estimated subtask execution time.

In *Unicorn*, a node's address space directory is guaranteed to contain true locations only for addresses in its local view. For all other addresses, it points to address space's *master node* (section 3.2), which points to the true location. Hence, a node can only compute affinity of subtasks to itself but not to other nodes. Thus, finding affinity of all subtasks to all nodes requires a distributed computation in which each node finds its own affinity to all subtasks. Another way to accomplish this is to compute this subtask-node affinity table centrally. But a task may use multiple address spaces. Centralized computation would require all master node's directories to be consolidated: leading to computation bottleneck in addition to large data transfer.

For these reasons, our locality-aware scheduler works with only partial affinity information on every node [4]. We let each node only compute every subtask's affinity to itself. This is a much smaller list with size equal to the number

of subtasks in one task. Only this list is centrally gathered and analyzed for task scheduling: at one node for each task.

Unicorn's default locality-oblivious scheduler initially computes the number of subtasks to be assigned to devices on each node, under the premise that all subtasks are likely to have equal load. We retain this as the first step even for locality-aware scheduling. Starting from this number of subtasks to be assigned to each node, we resort to a greedy approach (section 3.5.1.2.1) to assign specific subtasks to specific nodes.

3.5.1.2.1 Greedy Scheduling

In our greedy scheduling approach, subtasks pick nodes round-robin. Each time a subtask picks a node, the estimated completion time of that node is incremented by a time-estimate based upon the subtask's affinity score to the node. Accordingly, a global task completion estimate is also updated. Assume the computed affinity of subtask j to node v is A_{vj} and $\mathcal{T}$ is a user controlled function mapping A_{vj} to time t_{vj}. The estimate for node v, $T_v = \sum t_{vj}$ $\forall$ subtasks j assigned to node v. The estimate of the task, $T = \max T_v$ $\forall$ node v. Now, $\forall$ subtask j:

- assign j to node v such that the increase in T is the smallest.
- if two nodes v and w lead to the same increase (or no increase), assign v if $t_{vj} < t_{wj}$ and w otherwise.

We discuss various mapping functions $\mathcal{T}$ in section 5.3.2. [4] theoretically evaluates the greedy assignment algorithm and proves that greedily assigning

every work-item to a node that will complete it with the lowest resulting execution time is O($\log n$) competitive, with n being the number of nodes.

3.5.1.2.2 Locality-aware work stealing

The affinity information is also used during steal attempts where the victim agent chooses the subtasks with the highest difference in affinity towards the stealer. The victim computes a number – call it s – of subtasks to be sent to the stealer on the basis of their relative rates of subtask execution (i.e., the number of subtasks executed per second before the steal operation). The victim, then, chooses the s subtasks with the highest difference between their affinity to the stealer versus to the victim. As an aside, the stealer's affinity scores are not computed by the victim. We also do no include it with every steal request. Rather, we piggyback nodes' affinity scores on other data transfer. Since stealing happens near the end of the task, a stealer's affinity array is highly likely to reach all potential victims with negligible overhead. Nevertheless, if the affinity scores have not reached earlier, it comes with the request. In section 5.3.2, we report performance improvements with this scheme as compared to Unicorn's locality oblivious work stealer, which may allow a subtask with entire data on the victim's node to be stolen by a device on some other node with potentially no data, resulting in sub-optimal performance.

3.5.2 Work-Group Calibration

An application may opt for work-groups to be created from multiple CPU cores (on a node) in an attempt to assign them less work than intended in

a subtask. Otherwise, all the subtasks assigned to these cores may be stolen by faster GPUs. In this case, a subtask is logically subdivided and multiple work-items are created from it, with each work-item intended to run on one CPU core of the work-group. The maximum size of work-group created for a subtask is equal to the number of CPU cores on a node. The optimal size of work-group (for subtasks of a task) is, however, computed using binary search over the range [1, number of CPU cores].

Our algorithm requires at least two subtasks to bootstrap the binary search. For the first subtask, the sizeof work-group is set to the number of CPU cores on the node and for the second subtask the size of work-group is set to half the number of CPU cores. Next, we compare the execution times of both these subtasks. If the former one's execution time is shorter, the size of work-group for the third subtask is set to three-fourth the number of CPU cores. On the other hand, if the latter execution time is longer, then the size of work-group created for the third subtask is one-fourth the total number of CPU cores. This dynamic adjustment continues until the end of the task.

This dynamic calibration ensures that changing system load is optimally dealt with and CPU cores do not over-execute subtasks. Over-executing can be detrimental not only to the subtasks executed by CPUs but to the ones executed by GPUs as well. First, because several CPU cycles are required to complete GPU's work. In case, CPUs remain busy with other work, they won't be able to sufficiently keep the GPU pipeline busy by moving data to/from pinned memory (and GPUs) and launching GPU kernels. Further, large work on these slower cores also unnecessarily overload the memory and network throughputs at these nodes.

3.5.3 Multi-Assign

Work-stealing in *Unicorn* is non-preemptive. Hence, it is not possible to migrate a running subtask from one node (or device) to another. If a subtask takes a long time to finish, while other devices have become idle, *Unicorn* may *multi-assign* that subtask, i.e., assign the same subtask to multiple devices. At this stage, multiple instances of the same subtask may be running in the cluster. However, the one that finishes first commits its results to the global address space. All other running instances of the subtask are cancelled and their output (in their private views) is discarded.

We employ a simple protocol to determine the subtask instances to be cancelled. When a subtask is multi-assigned to a cluster device, it also memorizes the device where the subtask's original instance is running. The original instance also records the device executing the multi-assigned instance. Further multi-assignments are only allowed from the initial instance of the subtask and a multi-assigned instance can not multi-assign further. Thus, at any given time, the original instance knows where all the multi-assigned instances are running and all the multi-assigned instances know the location of the original instance. Whichever device finishes first, it informs the original instance, which in turn sends cancellation messages to all others.

Multi-assign is implemented as a part of stealing. In particular, we allow a subtask in 'executing' stage to also be stolen, while it is allowed to continue executing. Such steals are only allowed when the steal request specifically includes a multi-assign flag (i.e., when the stealing device is at maximum aggression level (section 3.5.1)).

We multi-assign only if the stealing device's estimated completion time (based upon its observed execution rate) is less than the victim's remaining estimated completion time. We use the remaining time for victim because it has already executed a part of the subtask at the time of *multi-assign*. Thus, the time it has already spent in execution of the subtask is subtracted from its total estimated time. As an additional heuristic, a subtask is only multi-assigned to a device on a different node or to a different device type on the same node (CPU subtasks are multi-assigned to GPU and vice versa). This reduces the chance of slow execution on the re-assigned device as well. Experiments show significantly faster completion times due to multi-assignment. This is especially useful in networked environments, where the effective data transfer bandwidths may be variable and a device on a node with slow link may not receive its input fast enough, delaying the entire task.

3.5.3.1 Subtask Cancellation

Post *multi-assign*, only the earliest completing instance of subtask is allowed to commit its output. All others are sent cancellation messages. In our implementation, subtask cancellations are done gracefully if they happen to be executing *Unicorn's* code at the time of cancellation. However, cancellation is complicated in case subtasks are executing application code at the time of cancellation.

We cancel a CPU subtask by performing a *longjmp* out of the application code. However, this has the potential to leak resources or leave them in an inconsistent state. For this reason, we require application kernels to be self-confined and not access global state that requires explicit cleanup. Ap-

plications that cannot conform to this requirement may opt to disable the *multi-assign* feature.

Further, the current state of technology on GPUs does not allow clean cancellation of an executing kernel. The entire GPU context may be destroyed but that has a high overhead and may also leave the GPU in an inconsistent state temporarily. For this reason, we currently do not multi-assign GPU subtasks in case they have copied data from CPU to GPU (and started kernel execution). However, if GPU technology matures and provides a low overhead kernel termination in future, we can enable GPU multi-assignment freely.

3.5.4 Scheduling Across Task Barriers

The *Unicorn* programming model introduces a task barrier at the end of every task. This existence of the barrier simplifies application programs and helps programmers logically think through the parallel program's control flow. However, barriers do incur overhead on the overall application performance, even with good load balance. To circumvent this overhead, we maintain only a logical barrier but dilute its physical presence by allowing subtasks (of a subsequent task) to be scheduled before the barrier imposed by the earlier task, as soon as their dependencies are satisfied. To support this, we pre-process subtask subscriptions of all submitted tasks and build a subtask data dependency graph. Any subtask which has all its dependencies satisfied can be allowed to start without waiting for the barrier. A detailed analysis of this aggressive scheduling scheme and the types of applications where they accrue benefit, however, is beyond the scope of this thesis.

3.5.5 Scheduling Concurrent Tasks

The current implementation of Unicorn allows multiple simultaneous tasks only if their subscribed address spaces are disjoint or if they are read-only. When a task is ready for execution, all its subtasks are added to execution queues of various compute devices. Thereafter, they are scheduled, stolen and multi-assigned as usual. An application can also specify a priority with a task to control its scheduling.

3.6 Software GPU Cache

Moving data between CPUs and GPUs is a relatively expensive operation. For optimal performance, it is important to reduce the frequency and volume of such data copies. Subtasks of a *Unicorn* task are allowed to have overlapping subscriptions. In such situation, it is beneficial to transfer data from CPU to GPU only once and use it for as many subtasks as possible. Similarly, output produced by a *Unicorn* task may serve as input to a subsequent task. In this case, data can be retained on GPUs at the end of the task in anticipation that a subtask (of a future task) may be scheduled on the same GPU.

Our runtime maintains an on-GPU software cache to optimize DMA transfers to the device. The cache allows a GPU to skip a data transfer in case the data is already resident. This scheme is suitable for both the scenarios described above, i.e., multiple subtasks of a task share read-only data and data written by a subtask of a task is later read by a subtask of another task. An exclusive

instance of GPU cache is created for every GPU in the cluster.

Unicorn's runtime supports four GPU cache eviction policies – LRU (least recently used), MRU (most recently used), LFU (least frequently used) and MFU (most frequently used). The eviction policies come into play when GPU is low on global memory space and memory needs to be freed up by evicting one or more cache entries. LRU scheme evicts the least recently used subscription, MRU evicts the most recently used subscription, LFU evicts the least frequently used subscription while MFU evicts the most frequently used subscription. By default, our current implementation employs LRU cache eviction policy. Depending upon the nature of the application, other cache eviction policies may be requested. A study of performance implications of these policies is presented in section 5.6.3.

Unlike hardware caches, we do not have a concept of cache lines that get invalidated. Rather, we invalidate entire subtask subscription. A study on a true cache implementation versus ours may provide interesting pointers for a better cache design. But this is beyond the scope of this thesis.

3.7 Conflict Resolution

Unicorn subtasks may subscribe to overlapping regions in an address space. The runtime treats this as a write conflict among subtasks and the application must explicitly specify a reduction mechanism to resolve the conflict. The output, otherwise, is not defined. *Unicorn* runtime provides a range of inbuilt reduction operators which an application may use. Applications are also free to define custom reductions.

Unicorn treats reduction as a first-class citizen and the output of subtasks are aggressively reduced as soon as they are ready. Thus, at any given time, a few subtasks may be in the execution stage while a few others may be in reduction stage. Aggressively performing reductions help in reducing overall execution time as these are executed during the time subtasks await their remote subscriptions to be fetched. Secondly, this reduces memory pressure on the system. As soon as two subtasks reduce, the memory of one of these is freed. Steal attempts are made only if there is no reduction to perform.

Note that subtasks may subscribe to an entire address space. This is the worst case for reduction as all subtasks conflict with each other. To efficiently perform reduction, *Unicorn* employs the following three optimizations:

1. Despite subscribing to entire address spaces, subtasks may only write to a few disjoint portions of the address space. We uses POSIX's *mprotect* feature to determine the VM pages touched a subtask and invoke conflict resolution on those pages only.

2. GPU kernels are by design multi-threaded. On CPUs, we use OpenMP's *parallel for* construct to speedup execution of inbuilt reduction operators.

3. Before transmitting data to remote nodes for reduction, it is compressed by our network subsystem (section 3.3). To optimize reduction, we also compress data before transmitting from GPUs to CPUs. *Unicorn* provides a few in-built compression algorithms but applications may also provide their own.

Chapter 5 describes the PageRank experiment implemented over *Unicorn*,

where entire address spaces are subscribed. The results of the experiment show that *Unicorn* achieves good scalability and reduction is performed quite efficiently.

Chapter 4

Pseudo Code Samples

Unicorn programs are written as a graph of tasks communicating through address spaces. A task reads from its input address spaces and writes to its output ones that may serve as input to one or more subsequent tasks. During execution, subtasks of a task are required to subscribe to address space regions they want to operate upon. Once the subscriptions are specified, Unicorn runtime optimally transfers the subscribed data to the device where subtask is executing and creates private views for subtask's writes to be accumulated.

Subtasks communicate with our framework through a series of callback functions - the *data subscription callback* defines the regions of the global shared memory a subtask would read-from or write-to (collectively called subtask subscription), the *kernel execution callback* is the actual computation logic of the subtask written either in hardware's native language like C/C++ for CPUs and CUDA for GPUs, or in a high level language like OpenCL [55] which caters to both CPUs and GPUs, and the *data synchronization callback* which defines how conflicting writes from different subtasks be merged before committing to the global shared address space.

This section presents pseudo code skeletons of two application programs: one computes Matrix Multiplication and the other computes two dimensional Fast Fourier Transform (2D-FFT) using our runtime. The former example

is implemented using one Unicorn task whereas the latter one employs two Unicorn tasks - the first one computes 1D-FFT on matrix rows while the second one computes 1D-FFT on matrix columns.

Listing 4.1 shows the pseudo code skeleton of a Unicorn program that multiplies two square matrices to produce the result matrix. The *matrix_multiply* function takes *matrix_dimension* and *block_dimension* as input. The former is the number of elements in rows/columns of the square matrices while the latter is the number of elements in rows/columns of each subtask. For simplicity, the code sample assumes that *matrix_dimension* is divisible by *block_dimension*. Line 5 of the code registers the *data subscription* callback and line 6 registers an OpenCL based *subtask execution* callback. This callback is executed on both CPU and GPU devices in the cluster. Lines 11 to 13 of the program create three address spaces - one for each matrix involved in the computation. The output (or result) matrix is logically divided into blocks of size *block_dimension* $*$ *block_dimension*. The program creates a task and a subtask for each of these logical blocks of the output matrix (line 20). The line also creates an instance of the task configuration struct *matmul_conf*. Task configuration is generally used for small task-wide read-only that all subtasks should see. *Unicorn* ensures that the configuration is automatically transported to all CPU and GPU devices in the cluster. Thereafter, lines 21 to 23 specify access modifiers for each of the three address spaces. Both input address spaces are accessed in read-only fashion while the output address space is write-only. Finally, line 24 submits the matrix multiplication task to the *Unicorn* runtime for asynchronous execution. The program waits for the asynchronous task to finish on line 25.

```
1  struct matmul_conf { size_t matrix_dimension,
       block_dimension; };

2  matrix_multiply(matrix_dimension, block_dimension)
3  {
4   key = "MATMUL";

5   register_callback(key,SUBSCRIPTION,
       matmul_subscription);
6   register_callback(key, OPENCL, "matmul_ocl",
       "prog.ocl");

7   if(get_host() == 0)  // Submit task from single host
8   {
9    // create address spaces
10   size = matrix_dimension * matrix_dimension *
        sizeof(float);

11   input1 = malloc_shared(size);
12   input2 = malloc_shared(size);
13   output = malloc_shared(size);

14   initialize_input(input1); // application code
15   initialize_input(input2); // application code

16   // find blocks per dimension (assumes divisibility)
17   block_count = matrix_dimension / block_dimension;

18   // create task with one subtask for every block of
        the output matrix
19   subtasks = block_count * block_count;
20   task = create_task(key, subtasks,
        matmul_conf(matrix_dimension, block_dimension));

21   bind_address_space(task, input1, READ_ONLY);
22   bind_address_space(task, input2, READ_ONLY);
23   bind_address_space(task, output, WRITE_ONLY);

24   submit_task(task);
25   wait_for_task_completion(task);
26  }
```

```
27 }
```

Listing 4.1: Unicorn program for square matrix multiplication

Unicorn callbacks for the matrix multiplication application are shown in listing 4.2. Each subtask computes its part in the result matrix. For that, it subscribes to corresponding rows of the first input matrix and corresponding columns of the second input matrix (lines 23-25) in the *data subscription* callback. The OpenCL kernel launch configuration is specified on line 27 and the OpenCL callback is implemented from line 30 onwards. This callback reads from the subtask's subscriptions in both input matrices and computes the subtask's output. For higher performance, Unicorn also allows the OpenCL subtask kernel to be substituted by two native kernels - one for CPU (in C/C++) and the other for GPU (in CUDA).

```
1  matmul_subscription(task, device, subtask)
2  {
3   matmul_conf* conf = (matmul_conf*)(task.conf);
4   block_count = conf→matrix_dimension /
       conf→block_dimension;

5   block_row = (subtask.id / block_count);
6   block_column = (subtask.id % block_count);

7   matrix_row_size = conf→matrix_dimension *
       sizeof(float);
8   block_row_size = conf→block_dimension *
       sizeof(float);

9   block_row_offset = block_row *
       conf→block_dimension * matrix_row_size;
10  block_column_offset = block_column *
       conf→block_dimension * sizeof(float);
11  block_offset = block_row_offset +
       block_column_offset;
```

```
12  /*
13   * Specify subscriptions
14   * A block subscription is defined by offset, size,
       step and count
15   * First input matrix subscribes to all blocks in
       the row block_row
16   * Second input matrix subscribes to all blocks in
       the column block_column
17   * Output matrix subscribes to the block at row
       block_row and column block_column
18   */

19  block_subscription_info bsinfo0(block_row_offset,
       matrix_row_size, matrix_row_size,
       conf→block_dimension);

20  block_subscription_info
       bsinfo1(block_column_offset, block_row_size,
       matrix_row_size, conf→matrix_dimension);

21  block_subscription_info bsinfo2(block_offset,
       block_row_size, matrix_row_size,
       conf→block_dimension);

22  // Bind subscriptions to address spaces by index
23  subscribe(task.id, device.id, subtask.id, 0,
       bsinfo0);
24  subscribe(task.id, device.id, subtask.id, 1,
       bsinfo1);
25  subscribe(task.id, device.id, subtask.id, 2,
       bsinfo2);

26  // OpenCL kernel launch configuration
27  set_launch_conf(task.id, device.id, subtask.id,
       ...);
28 }


29 // OpenCL Kernel (should go into prog.ocl file)
30 matmul_ocl(task, device, subtask)
31 {
32  matmul_conf* conf = (matmul_conf*)(task.conf);
```

```
33  float* input1 = (float*)subtask.subscription[0];
34  float* input2 = (float*)subtask.subscription[1];
35  float* output = (float*)subtask.subscription[2];

36  ... OpenCL Matrix Multiplication Code ...
37 }
```

Listing 4.2: Unicorn callbacks for square matrix multiplication

Listing 4.3 shows the pseudo code for 2D-FFT. The experiment firstly performs a 1D-FFT along the input matrix rows (lines 22-29) and then performs 1D-FFT along the columns of the input matrix (lines 30-39). Different callbacks are registered for both kinds of FFTs. Subscriptions for the row FFT task are specified on lines 9-10 while the ones for the column FFT task are registered on lines 11-12. Line 35-37 ensure that the column FFT uses the output address space of the row FFT as its input. The input address space of the row FFT contains the final output after execution of the column FFT task.

```
1 struct complex { float real, imag; };
2 struct fft_conf { size_t rows, cols; };

3 ROWS_PER_SUBTASK = 128;

4 // Assumes matrix_rows and matrix_cols are divisible
    by ROWS_PER_SUBTASK
5 fft_2d(matrix_rows, matrix_cols)
6 {
7  row_fft_key = "FFT_ROW";
8  col_fft_key = "FFT_COL";

9  register_callback(row_fft_key, SUBSCRIPTION,
     fft_row_subscription);
10  register_callback(row_fft_key, OPENCL,
     "fft_row_ocl", "prog_row.ocl");
```

```
11  register_callback(col_fft_key, SUBSCRIPTION,
       fft_col_subscription);
12  register_callback(col_fft_key, OPENCL,
       "fft_col_ocl", "prog_col.ocl");

13  if(get_host() == 0)  // Submit task from single host
14  {
15   // create address spaces
16   size = matrix_rows * matrix_cols * sizeof(complex);
17   input = malloc_shared(size);
18   output = malloc_shared(size);

19   initialize_input(input); // application data

20   // create task configuration
21   fft_conf fft_task_conf(matrix_rows, matrix_cols);

22   /* Row FFT Task */

23   // create task with one subtask per row
24   nsubtasks = matrix_rows / ROWS_PER_SUBTASK;
25   task = create_task(row_fft_key, nsubtasks,
        fft_task_conf);

26   bind_address_space(task, input, READ_ONLY);
27   bind_address_space(task, output, WRITE_ONLY);

28   submit_task(task);
29   wait_for_task_completion(task);

30   /* Column FFT Task */

31   // create task with one subtask per column
32   nsubtasks = matrix_cols / ROWS_PER_SUBTASK;
33   task = create_task(col_fft_key, nsubtasks,
        fft_task_conf);

34   // Swap both the address spaces
35   swap(input, output);

36   bind_address_space(task, input, READ_ONLY);
```

```
37    bind_address_space(task, output, WRITE_ONLY);

38    submit_task(task);
39    wait_for_task_completion(task);
40   }
41 }
```

Listing 4.3: Unicorn program for 2D-FFT

Listing 4.4 describes callbacks of both 1D-FFTs. Each subtask of the row FFT task computes 1D-FFT on a contiguous set of 128 rows while each subtask of the column FFT task computes 1D-FFT on a contiguous set of 128 columns of the input matrix. The subscription for row FFT task is a contiguous chunk of 128 rows (lines 4-9) while the subscription for column FFT task is a scattered set of rows of 128 elements (lines 24-31).

```
1  fft_row_subscription(task, device, subtask)
2  {
3   fft_conf* conf = (fft_conf*)(task.conf);
4   rows_size = ROWS_PER_SUBTASK * conf→cols *
       sizeof(complex);

5   // subscribe using offset and length
6   subscription_info sinfo(subtask.id * row_size,
       rows_size);

7   // subscribe to input and output memory by index
8   subscribe(task.id, device.id, subtask.id, 0, sinfo);
9   subscribe(task.id, device.id, subtask.id, 1, sinfo);

10  // OpenCL kernel launch configuration
11  set_launch_conf(task.id, device.id, subtask.id,
       ...);
12 }

13 // OpenCL Kernel (should go into prog_row.ocl file)
14 fft_row_ocl(task, device, subtask)
15 {
```

```
16  fft_conf* conf = (fft_conf*)(task.conf);
17  complex* input = (complex*)subtask.subscription[0];
18  complex* output = (complex*)subtask.subscription[1];

19  ... OpenCL 1D FFT Code ...
20 }

21 fft_col_subscription(task, device, subtask)
22 {
23  fft_conf* conf = (fft_conf*)(task.conf);
24  offset = ROWS_PER_SUBTASK * subtask.id *
       sizeof(complex);
25  cols_size = ROWS_PER_SUBTASK * sizeof(complex);
26  step_size = conf->cols * sizeof(complex);

27  // subscribe using offset and length
28  block_subscription_info bsinfo(offset, cols_size,
       step_size, conf->rows);

29  // subscribe to input and output memory by index
30  subscribe(task.id, device.id, subtask.id, 0,
       bsinfo);
31  subscribe(task.id, device.id, subtask.id, 1,
       bsinfo);

32  // OpenCL kernel launch configuration
33  set_launch_conf(task.id, device.id, subtask.id,
       ...);
34 }

35 // OpenCL Kernel (should go into prog_col.ocl file)
36 fft_col_ocl(task, device, subtask)
37 {
38  fft_conf* conf = (fft_conf*)(task.conf);
39  complex* input = (complex*)subtask.subscription[0];
40  complex* output = (complex*)subtask.subscription[1];

41  ... OpenCL 1D FFT Code ...
42 // Successive elements are conf->cols apart
43 }
```

Listing 4.4: Unicorn callbacks for 2D-FFT

Chapter 5

Experimental Evaluation

We have implemented several coarse-grained scientific computation benchmarks over Unicorn. These include image convolution, matrix multiplication, LU matrix decomposition, two-dimensional fast Fourier transform (2D-FFT) and Page Rank. These benchmarks have well known parallelizations and are diverse enough for studying different kinds of complexities. Image convolution, although embarrassingly parallel, has a unique requirement of fringe around the image. Matrix multiplication is computationally intensive with heavy data transfer requirements. LU decomposition has a nested task hierarchy, 2D-FFT involves a parallelization-unfriendly matrix transpose operation and finally Page Rank is a map-reduce based computation. The goal of these experiments is to assess conditions under which our runtime responds well.

Our experimental cluster consists of fourteen nodes, each equipped with two 6-core Intel Xeon X5650 2.67 GHz processors and two Tesla M2070 GPUs. The nodes run CentOS 6.2 with CUDA 5.5. For communication, we use Open MPI [31] 1.4.5 (over SSH) over an InfiniBand [1] network with 32Gbps peak bandwidth. Unless stated otherwise, the input address spaces are randomly distributed (as 2048×2048 blocks) over the cluster nodes and our runtime transparently moves data on-demand to other nodes, as the program executes.

Our runtime allows restricting applications to only certain cores, e.g., only CPU cores, only GPUs, or both, However, CPU cores, being main OS vehicles, are not used purely for computation but also other support functions like CPU-GPU data transfers, GPU kernel launches, network data transfers, etc. When subtask data usage is high (e.g., in several hundred MBs per subtask), significant CPU load is observed in CPU↔GPU data transfers. Based on our empirical analysis, we withhold upto two CPU cores (per node) from subtask execution when both CPUs and GPUs are requested by an application. In section 5.7.1, we study the performance impact of varying the number of CPU cores employed in computations. This also provides an insight into the library's runtime overheads.

Many of our experiments use subtasks of size 2048×2048. This empirically determined size works well for both CPUs and GPUs. The size is not too large for CPUs (to cause frequent cache misses) and not too small for GPUs (to cause their under-utilization). In section 5.5.3, we present the impact of varying subtask size on performance.

We first discuss the implementation of the selected benchmarks over Unicorn and then present how these scale with an increasing number of nodes. We also compare the performance of these implementations to the reference cases when only CPU cores and when only GPUs are used in the cluster.

Next, we evaluate Unicorn's work-stealing based scheduling by comparing it with a work-sharing scheduling scheme. We also compare the overheads of one-level work-stealing with a naive two-level technique and *ProSteal* (section 3.5.1.1) – a scheduling optimization which proactively steals subtasks before the devices finish their assigned subtasks. Next, we evaluate Uni-

corn's locality-aware scheduling, which is used in cases when the subtasks of our experiments are written without any particular order of spatial locality. As described in section 3.5.1.2, the affinity of subtasks to nodes is evaluated in a separate internally created *Unicorn* task. The outcome of this task is a table that gives a score to a subtask-node pair. The score can be based on several criterion. In our evaluation, we evaluate four different heuristics for computing the affinity (or locality) score. The first three heuristics respectively maximize reuse of local data on a node (i.e., score awarded to a subtask-node pair is directly proportional to the amount of subtask's input data resident on the node), minimize the number of remote data transfer events per subtask, and minimize the estimated data transfer time per subtask. The fourth one is a hybrid heuristic called *Derived Affinity*, which computes scores by minimizing both remote transfer events and estimated data transfer time per subtask.

The scheduling results are followed by a study of *load balance* achieved by our runtime. Here, we study the finishing times of various cluster devices involved in computations. The closer the devices finish to each other, the better the load is balanced. We also experiment with changing the input data location in the cluster and observe that the load still remains balanced as our runtime automatically moves more computations to the node with more resident data.

Next, we test our runtime's resilience to data organization and subtask sizes used in computations. We first change the initial placement (in the cluster) of input data for various experiments and study the response of our runtime to various commonly used patterns. Next, we change subtask sizes for various

experiments and study its performance implications.

Next, we study various optimizations presented in section 3. These include *multi-assign*, *pipelining* and *gpu caches*. This is followed by a study of *Unicorn's* overheads where we focus on time spent in runtime *versus* application code. We also vary the number of CPU cores employed in computations requiring both CPUs and GPUs in the cluster and study the response of the experiments.

All measurements are based on at least three trials. Results are presented for the sample that reported minimum execution time.

5.1 Unicorn Parallelization of Benchmarks

In our image convolution experiment all color channels of a 24-bit RGB image of size 65536×65536 are convolved with a 31×31 filter. The input image is stored in a read-only address space (initially distributed randomly across the cluster nodes), logically divided into 1024 blocks of size 2048×2048. Each block is convolved using a separate subtask. However, because convolution at boundaries requires data from adjoining blocks, the input memory subscription of a subtask overlaps with other subtasks', usually at all four boundaries. The output image is generated in a write-only address space.

In the matrix multiplication experiment two dense square matrices of size $2^{16} \times 2^{16}$ each are multiplied to produce the result matrix. Each input matrix is stored in a read-only address space and the result matrix is stored in a write-only address space of the task. The output matrix is logically divided

into 2048 × 2048 blocks and computation of each block is assigned to a different subtask (which subscribes to all blocks in the corresponding row of the first input matrix and all blocks in the corresponding column of the second input matrix). The CPU subtask callback is implemented using a single-precision BLAS [24] function and the GPU callback uses the corresponding CUBLAS [17] function.

For the in-place block LU Decomposition [21] experiment, the input matrix ($2^{16} \times 2^{16}$) is kept in a read-write address space and is logically divided again into 2048 × 2048 sized blocks. The matrix is solved top-down for each diagonal block. For a matrix divided into $n * n$ blocks, solving for each diagonal block (i, j) involves three tasks – LU decomposition of the diagonal block (i, j), propagation of its results to other blocks in its row $(i, j + 1...n)$ and column $(i + 1...n, j)$, and propagation of these results to other blocks underneath $(i + 1...n, j + 1...n)$. The first of these three tasks is executed sequentially while the other two are executed in parallel. One task is spawned per diagonal block, which in turn, executes 3 tasks within, making a total of $3n - 2$ tasks (where n is the number of diagonal blocks). The parallelism in tasks (i.e., the number of subtasks) reduces as we move down the matrix because the number of blocks to be solved in parallel decreases. The CPU subtask implementation uses single-precision BLAS functions while the GPU implementation employs the corresponding CUBLAS routines.

The 2D-FFT experiment performs two single-precision one dimensional complex-to-complex FFTs (one along matrix rows and the other along matrix columns) over a matrix with 61440 × 61440 elements. The input matrix is initially randomly distributed over the cluster nodes in blocks of 512 consecutive matrix

rows. We use two *Unicorn* tasks for the experiment (each with 120 subtasks). The first task performs 1D-FFT along matrix rows while the second performs 1D-FFT along matrix columns. Note that we do not need to perform an explicit transpose in between the two and instead rely on our network subsystem to efficiently serve data. For the first 1D-FFT, the subtask size is 512 rows while it is 512 columns for the second. The CPU subtask callback uses calls to the FFTW [30] library, whereas the GPU subtask uses calls to the CUFFT [18] library functions.

The Page Rank experiment computes the search rank of a webpage based on the web's hyperlink structure. The search rank of a webpage is the probability of a random surfer visiting it. The algorithm works by first uniformly initializing the ranks of each page to a constant value, and then iteratively transferring the ranks of all web pages to their outlinks till the ranks of all pages converge (or till a maximum number of iterations).

For our experiment, we use a randomly generated web graph of 500 million web pages and a maximum of 20 outlinks per web page. With 90% probability, a page's outlinks point to nearby pages (within an imaginary circle centered at this web page and having a diameter of 0.1% of total web pages), with 9% probability, a page's outlinks point within a diameter of 1%, and with 1% probability they are linked to farther off pages. The data for the web (along with its outlinks) is stored on disk in multiple NFS files with each file storing record of one million web pages. Data from these 500 files is read by all cluster nodes through a parallel file mapping API provided by *Unicorn*. This API memory maps file regions requested by a subtask and provides a handle to access it.

Our Page Rank implementation performs 25 iterations and each iteration executes 250 subtasks. Each subtask processes 2 million distinct web pages and transfers each web page's page rank to all its outlinks equally. Note that a web page may point to any other page in the web. Thus, the corresponding subtask may produce output page rank for any web page. For this reason, all subtasks write-subscribe to entire output address space and the output of all subtasks are summed up (using *Unicorn's* data reduction callback) to produce the final output. The output of subtasks is written to an address space in every iteration and after reduction it becomes the input for the next iteration. We allocate two address spaces and cycle them as input or output every iteration. After the last iteration, the output address space contains the final page ranks. The CUDA kernel code for this experiment uses the GPU's global memory to accumulate page ranks produced by various GPU threads (employed by a subtask) and synchronization between them is done using global memory atomic add instructions.

5.1.1 Characteristics of Benchmarks

Figure 5.1 provides a quick glance at our implementations of the said benchmarks. Note that per subtask input data requirement for *Matrix Multiplication*, *2D FFT* and *Page Rank* is quite high in comparison to other experiments. Because of these large data sizes, our runtime turns off use of pinned buffers for *CPU to GPU* data transfers (as excessive pinned memory usage degrades overall system performance). For similar reasons, *GPU to CPU* data transfers are also done without pinned buffers. As a side effect of turning off pinned buffers, multiple GPU subtasks cannot be pipelined (for data

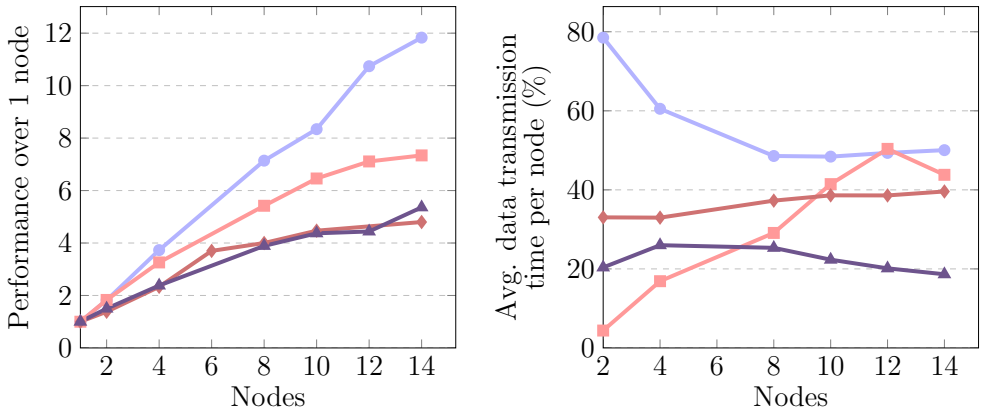
transfers and computations) and multiple simultaneous kernel executions are not possible. This also prevents our runtime from employing our aggressive stealing approach called *ProSteal* (section 3.5.1.1). For these reasons, we also present some results for smaller input matrix sizes (32768 × 32768), where usage of pinned memory buffers is not too large to throttle the memory subsystem. Our runtime is then able to employ pinned buffers and multiple kernel executions. Although our runtime turns off pinned buffers if subtask subscription is too high, but an application program may reduce subtask size to enable these.

Benchmark	Characteristic					
	Total Input (in GB)	Total Output (in GB)	Task Count	Max. Subtask Input (in MB)	Max. Subtask Output (in MB)	Requires Conflict Resolution
Image Convolution	12	12	1	12.35	12	No
Matrix Multiplication	32	16	1	1024	16	No
LU Decomposition	16	16	94	48	16	No
2D FFT	28.125	28.125	2	240	240	No
Page Rank	40.98	1.86	25	175.48	1907.35	Yes

Figure 5.1: Characteristics of various benchmarks

5.2 Performance Scaling

Results in Figure 5.2 show strong scaling achieved by our implementations (of Image convolution, Matrix multiplication, LU decomposition and 2D FFT) discussed in section 5.1. Of the four experiments, Image convolution exhibits maximum scaling and peaks at 11.83 when run on 14 nodes. Matrix multiplication, being relatively expensive on communication, achieves a maximum scaling of 7.34. In contrast to image convolution, which spends 50.05% of



Nodes	Data transferred (GB) 2	4	8	10	12	14
Image Convolution	6.06	9.30	10.61	10.69	10.96	11.40
Matrix Multiplication	17.00	61.50	140.00	186.50	224.00	248.50
LU Decomposition	40.02	79.22	142.91	172.00	197.97	222.45
2D FFT	24.66	45.67	50.33	49.46	52.94	51.22

Nodes	Data transfer events 2	4	8	10	12	14
Image Convolution	1550	3203	4462	4965	5119	5383
Matrix Multiplication	67	209	481	543	666	659
LU Decomposition	7527	15248	25428	30320	32094	36654
2D FFT	1830	3966	11180	10842	10930	14584

Image Convolution — Matrix Multiplication — LU Decomposition — 2D FFT

Figure 5.2: Performance analysis of various benchmarks

the total experimental time in communication (of 11.4 GB), matrix multiplication transfers a much larger 248.50 GB data in the cluster and this takes 43.81% of the total experimental time. Although for matrix multiplication both input matrices are 16 GB each, every input block is required by all subtasks in its row and all in its column. Thus about 250 GB data is eventually transferred in 659 pipelined events (each of the 1024 subtasks subscribe to 1 GB data from both the input matrices).

The other two experiments, block LU decomposition and 2D-FFT, exhibit relatively inferior scaling (4.8 and 5.36 respectively) as these experiments have limited parallelism. The former is an iterative experiment with 32 iterations and 3 tasks per iteration (one out of these three is sequential). The experiment has an average of 121.7 subtasks per task. The 2D-FFT experiment (with an implicit matrix transpose operation) has two tasks each with 120 subtasks.

The results in Figure 5.2 indicate a similar amount of data transfer for matrix multiplication and LU decomposition. However, the former is implemented as a single task with time complexity $O(n^3)$ and the latter has many iterations with three tasks using BLAS calls of varying time complexities ($O(n)$, $O(n^2)$ and $O(n^3)$). This increases communication latency, which is evident from fifty times more data transfer events (36654 *versus* 659). This, coupled with the fact that the first of these three tasks is sequential, leads to lower scalability for the experiment. In contrast, both image convolution and 2D FFT have lower time complexities – $O(nm)$ for the former (m being the filter size) and $O(n \log n)$ for the latter, but the latter has around 4.5x more data transfer, resulting in its lower scaling. These results collectively indicate that

applications with high compute to communication ratio should perform well with our runtime.

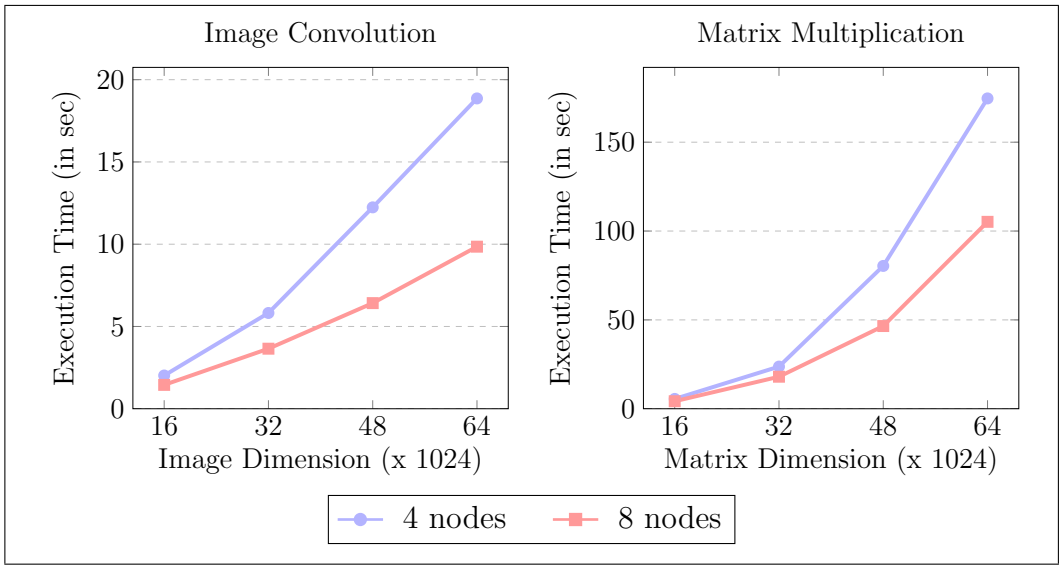


Figure 5.3: Scaling with increasing problem size

Figure 5.3 plots the execution times of image convolution and matrix multiplication benchmarks with increase in the problem size. The results are presented for two cases - in the first case only 4 cluster nodes are used for execution while 8 cluster nodes are employed in the second case. Results show that our runtime responds equivalently to various input sizes as similar scaling curves (while varying workload sizes) are observed despite increase in the number of nodes.

5.2.1 CPU versus GPU versus CPU+GPU

Figures 5.4, 5.5 and 5.6 plot GPUs-only performances for image convolution, matrix multiplication and 2D FFT respectively and compare those to the corresponding CPU+GPU performances. For the computation in these applications there is a significant performance disparity between CPUs and

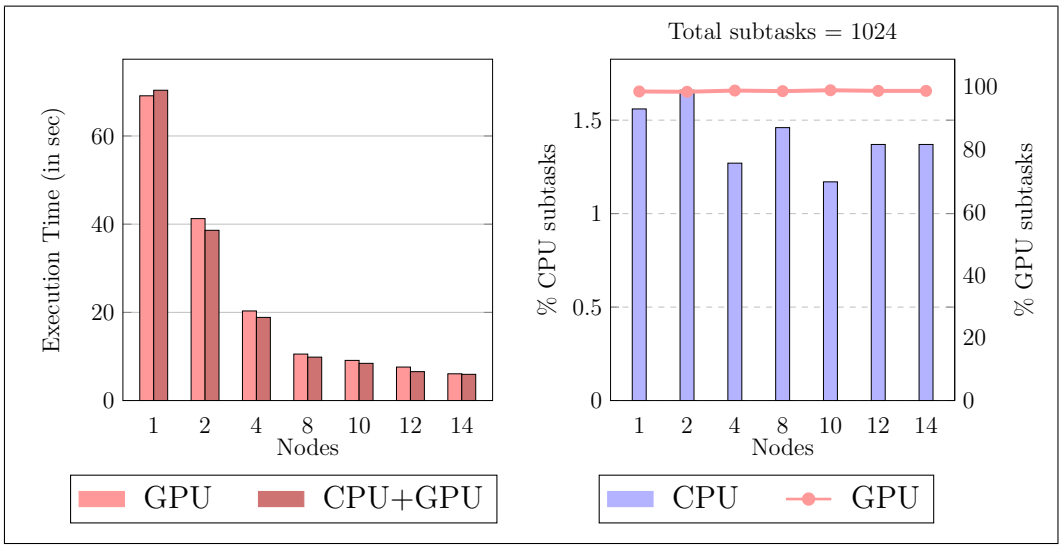


Figure 5.4: Image Convolution – GPU vs. CPU+GPU

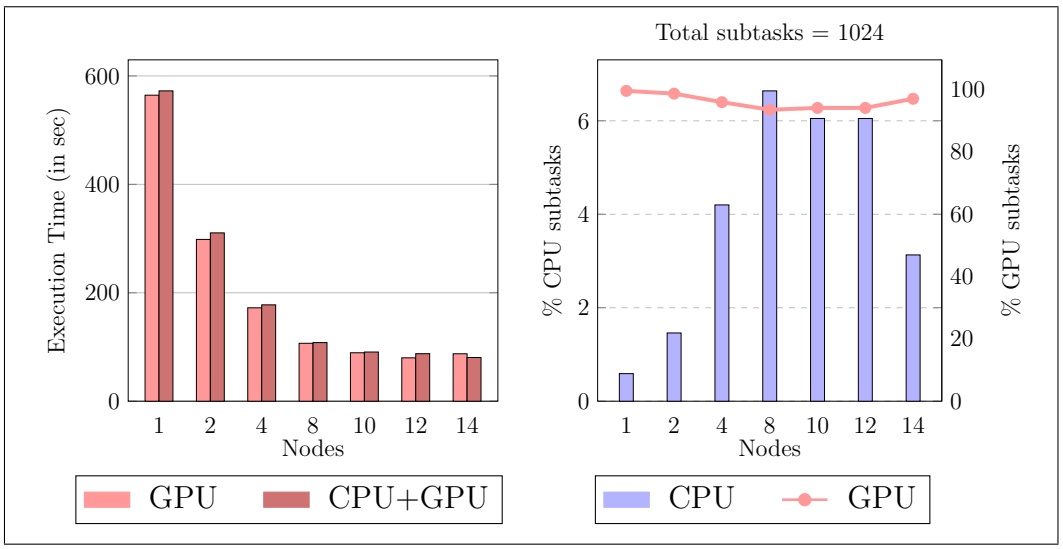


Figure 5.5: Matrix Multiplication – GPU vs. CPU+GPU

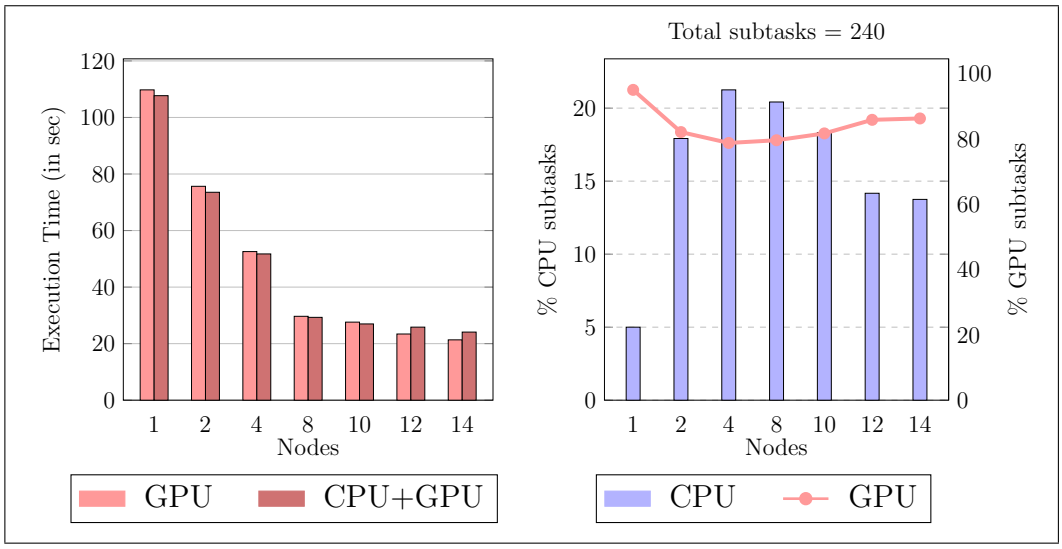


Figure 5.6: 2D FFT – GPU vs. CPU+GPU

GPUs, with GPUs being several times faster than CPUs. Due to this, employing both together generally results in nearly all CPU subtasks getting *multi-assigned* to GPUs. Also, running some subtasks on slower CPUs takes away the opportunity to pipeline those in case the GPUs had executed them. For these reasons, the experiments' performances when using CPU+GPU do not show much throughput gain over their GPUs-only versions. For image convolution and matrix multiplication, results show that nearly 98% subtasks were executed by GPUs and CPUs were only able to complete fewer than 2% of the subtasks. For 2D FFT, this ratio is a little better with around 15% subtasks completed by CPUs.

Figure 5.7 compares CPUs-only and GPUs-only performances for matrix multiplication and 2D FFT for input matrices of size 32768×32768. The wide performance disparity between CPUs and GPUs is evident in these results (especially for matrix multiplication where in the 1-node case CPUs-only performance is around 5 times slower than GPUs-only performance).

The disparity is narrower for 2D-FFT, where the best results occur when using both CPUs and GPUs together. However, this experiment uses only 32 subtasks, thus restricting parallelism and performance gains to 32 devices (i.e., 2 nodes) only. Ideally, the number of subtasks should be a few times the number of devices. But in this case, the number of subtasks becomes less than the number of devices as the number of nodes in the cluster grows.

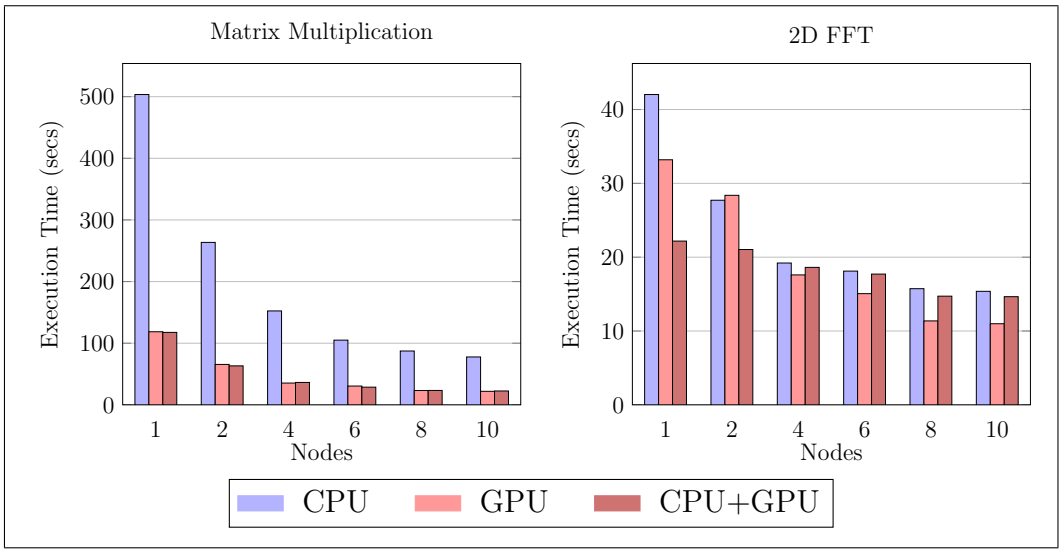


Figure 5.7: Experiments with matrices of size 32768×32768 (lower is better)

5.2.2 PageRank

Figure 5.8 presents results for the PageRank experiment. The experiment executes 25 map-reduce iterations. In the map stage, 250 subtasks process an equal share of input web pages and distribute their page ranks equally among all their outlinks. In the reduce stage, the page rank contributions from each subtask are summed up to compute the final page ranks of all web pages. For efficiency, reductions are performed in parallel on all cluster nodes. Once the intra-node reductions complete, nodes perform inter-node

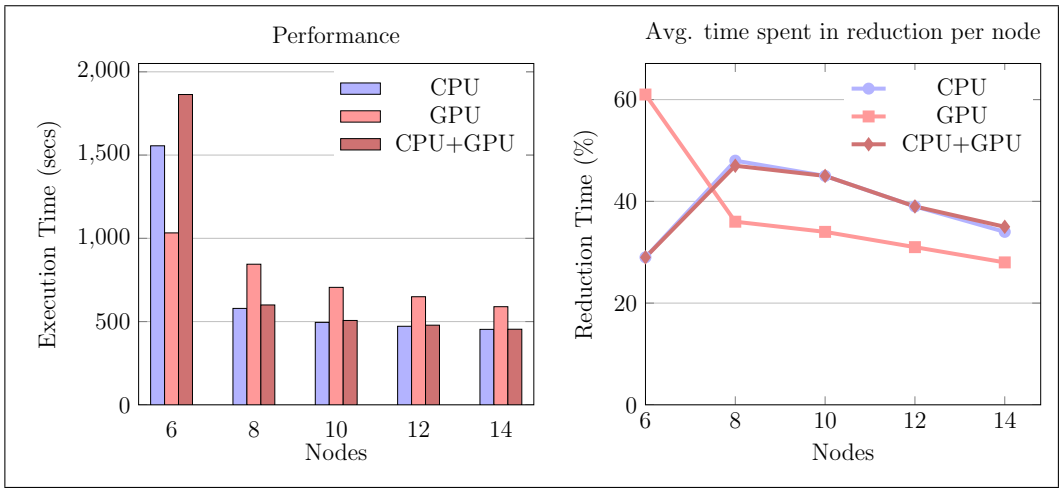


Figure 5.8: Page Rank

reductions to compute the final result.

Results show that CPUs perform better than GPUs for this experiment. This is because our GPU implementation uses slow atomic-add operations to compute subtask output. Also, in this case, there is additional overhead of large GPU-CPU data transfers (Figure 5.1). Along with performance results, Figure 5.8 plots the average time spent in reduce stage by every node in the cluster. Results show that the time spent in reduction stage decreases with an increasing number of nodes. This is because more nodes allow more reductions to be simultaneously executed and the total time spent in reduction decreases.

5.3 Scheduling

In this section, we compare our distributed and dynamic two-level scheduler with a dynamic centralized scheduler. The dynamic centralized scheduler assigns tasks to nodes, as they become free. There is no pre-emption. In

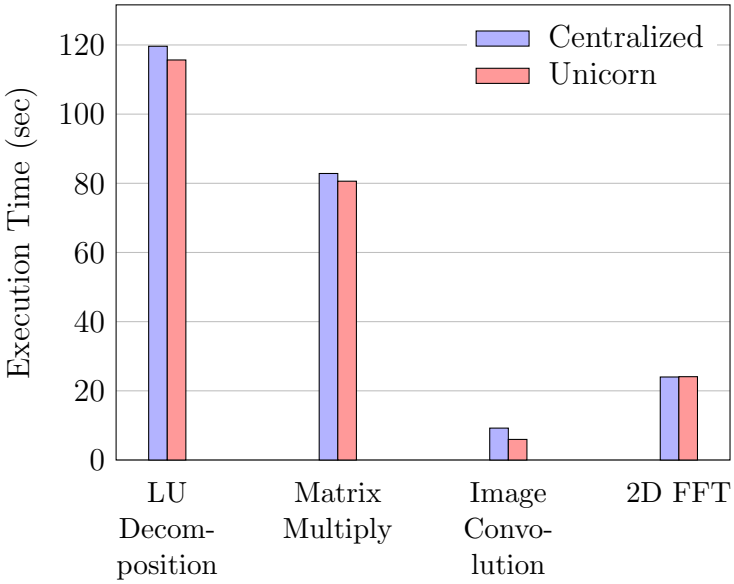


Figure 5.9: Centralized scheduling versus Unicorn – 14 nodes

order to reduce scheduling overhead, it assigns subtasks in chunks. It starts with a default static allocation of subtasks and then incrementally calibrates the chunk size based on the observed execution rate at each computing unit. Initially, a single subtask is assigned to each unit. Depending on the completion time of the assigned work, the scheduler chooses the number of subtasks to assign to that unit the next time around. If a unit finishes the job earlier than most others, its chunk size is doubled. On the other hand, if a unit completes its subtasks slower than most, its chunk is halved in the next allocation. Results in Figure 5.9 show that the performance of *centralized* scheduler is inferior to *Unicorn's* distributed scheduling.

5.3.1 Work Stealing

This section presents a comparison of one-level work stealing versus *Unicorn's* two-level scheme with a *steal agent* running per node (section 3.5.1). We also

compare the performance of the latter to our aggressive stealing scheme called *ProSteal*.

5.3.1.1 One-level vs. Two-level

Results in Figure 5.10 show comparable performance numbers for one-level and two-level scheduling schemes for matrix multiplication and LU decomposition. However, image convolution and 2D FFT respectively report 6.92% and 2.13% performance improvement with two-level stealing, compared to the one-level stealing scheme. The gain in image convolution is because of the small memory footprint of an image convolution subtask, which allows GPUs (which are the stealers in most cases) to co-execute more subtasks as compared to other experiments where despite stealing a set of subtasks large memory footprint prohibits their co-execution.

Despite moderate performance gains, two-level stealing reports significant reduction (over one-level) in the total number of steals attempts generated in the cluster. The average reduction for image convolution, matrix multiplication, LU decomposition and 2D FFT respectively is 65.88%, 64.9%, 67.03% and 66%. Due to this, the average number of successful steals in the cluster increase in two-level scheme (over one-level) by 44.24%, 54.25%, 60.08% and 52.3% for these experiments respectively. The same is not translated into performance gains because our steals are extremely light weight. The only information transferred as a result of steal is the *subtask id* and there is no associated direct data transfer.

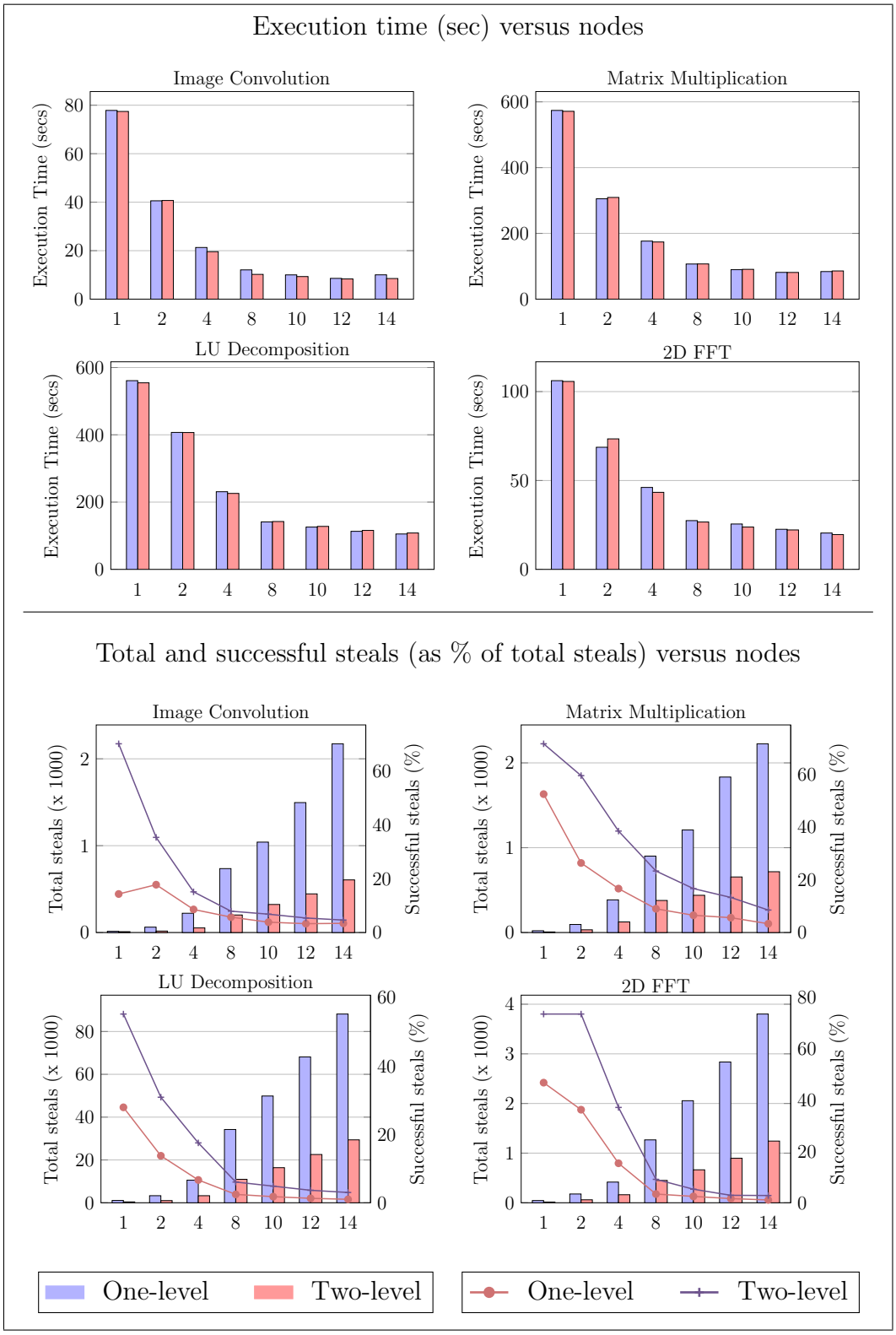
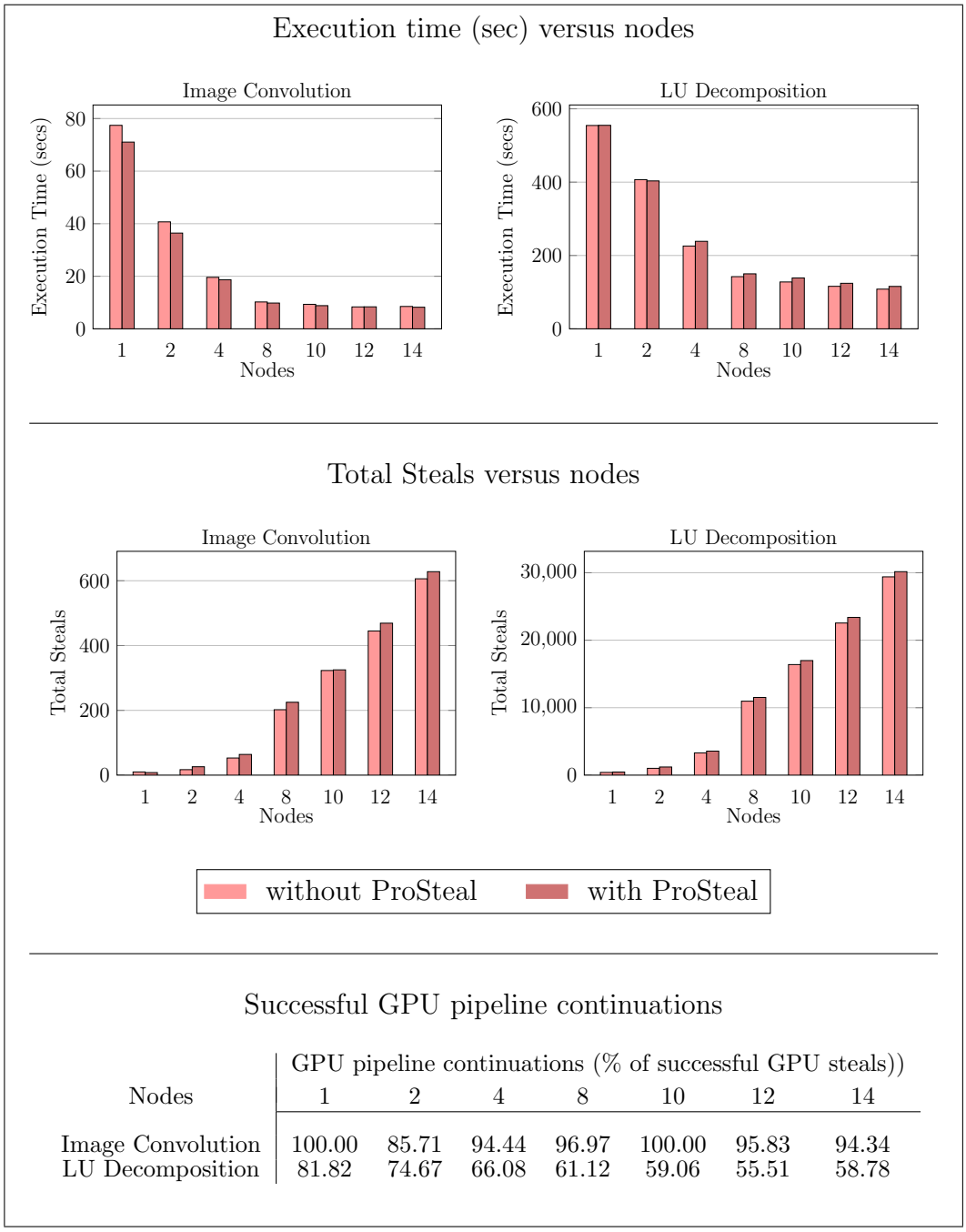


Figure 5.10: One-level versus two-level work stealing



Successful GPU pipeline continuations

Nodes	GPU pipeline continuations (% of successful GPU steals))						
	1	2	4	8	10	12	14
Image Convolution	100.00	85.71	94.44	96.97	100.00	95.83	94.34
LU Decomposition	81.82	74.67	66.08	61.12	59.06	55.51	58.78

Figure 5.11: Work Stealing – with and without ProSteal

5.3.1.2 ProSteal

Unicorn avoids GPU pipeline stalls by allowing devices to steal before their pipelines are completely flushed. As discussed in chapter 3, the appropriate time when a GPU should steal is dynamically computed based on the GPU's subtask execution rate and its latency to re-prime its pipeline after a stall. Figure 5.11 plots the performance of this scheme versus the performance obtained by Unicorn scheduler without ProSteal. On an average, ProSteal yields 5.11% better performance for image convolution but degrades by 4% for LU decomposition experiment. The performance gain in image convolution is because ProSteal is able to avert GPU pipeline stalls for 95.33% successful GPU steals (which are 33% of the total steals). The number is only 65.29% for LU decomposition where successful GPU steals are only 14% of the total steals. More detailed results of ProSteal can be found in [3].

5.3.2 Locality-aware Scheduling

In this section, we study node-affinity based scheduling in *Unicorn*. For this, we have modified our experiments such that there is no particular spatial coherence between consecutive subtasks' data. In other words, adjacent subtasks of our experiments do not necessarily execute on adjacent address space regions.

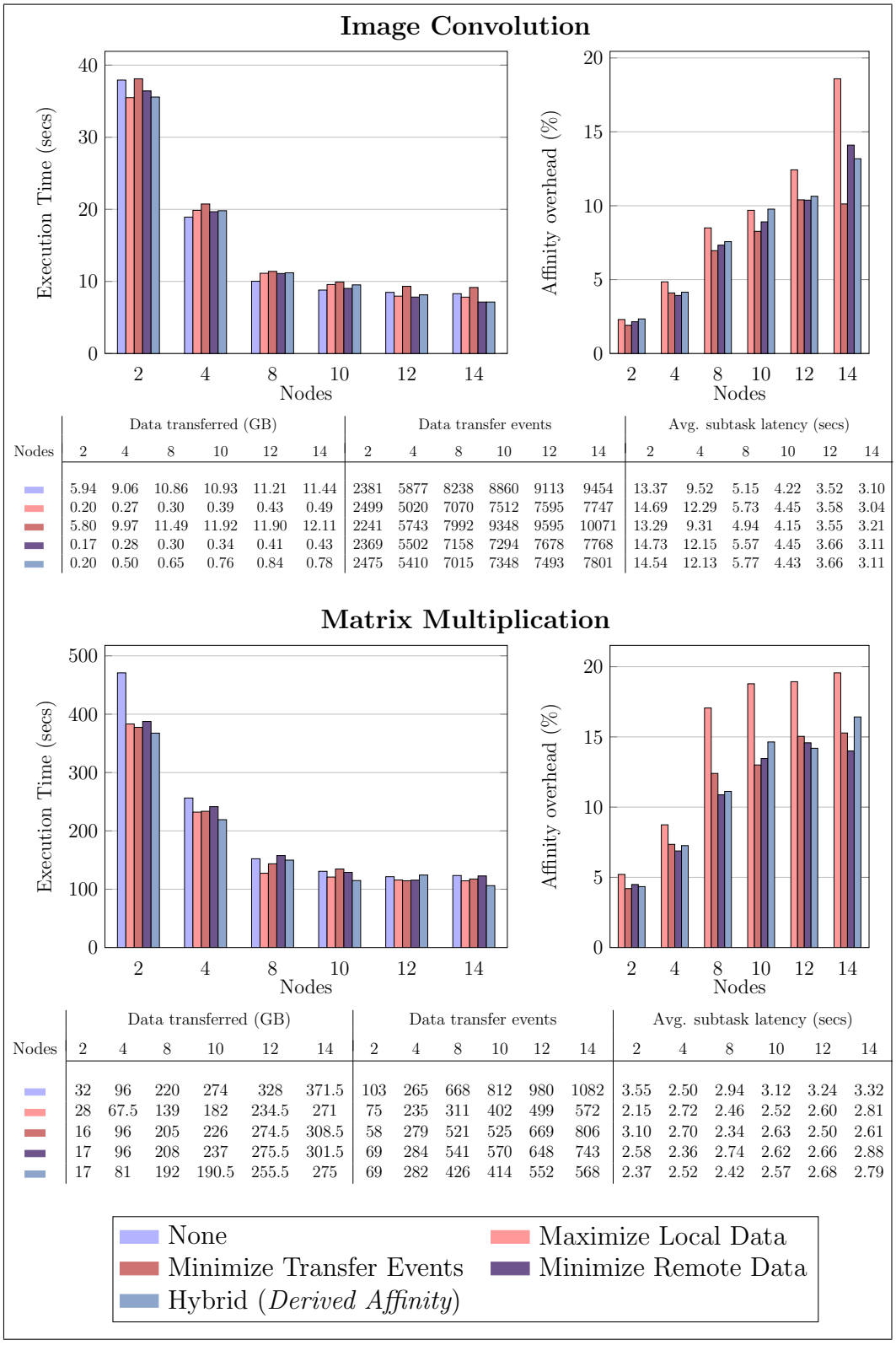
For locality-aware scheduling, we compute an affinity score for every subtask on every node and our scheduler uses this score to maximize global affinity. We experiment with four heuristics. We start with scheduling a subtask on a node where most of its input resides, maximizing *local data*. In this case, the

more the amount of local data (on a node) for a subtask is, the more the corresponding affinity score is. The other strategies, instead of maximizing local data, target the time to fetch remote data. The next strategy minimizes the number of remote data *transfer events* or requests. It is based on the observation that the incurred data fetch latency grows with data fragmentation and the number of data transfer requests. Accessing closely placed remote data is less expensive than accessing discontiguous remote data, which may cost additional latency. The third strategy optimizes the amount of *remote data*. It minimizes the total number of virtual memory pages to be fetched from remote nodes. Our last strategy, called *Hybrid* or *Derived Affinity* optimizes for both remote transfer events and time.

Figures 5.12 and 5.13 plot the performance of *Unicorn's* locality-oblivious scheduler as well as the performance of the *local data* based affinity and compares these to *remote data* based affinity (*transfer events*, *remote data* and *hybrid*). The figures also record the cluster-wide data transfers and subtask latency incurred in these experiments.

Results show that one or more of our heuristics perform better than *Unicorn's* locality oblivious scheduler at most of the data points. For image convolution experiment, a maximum gain of 13.93% (over *Unicorn's* default scheduler) is observed with *remote data* and *hybrid* heuristics in the fourteen node case. This is attributed to a substantial data transfer reduction from 11.44 GB (for default scheduler) to 0.43 GB (for *remote data* heuristic) and 0.78 GB (for *hybrid* heuristic). A gain of similar magnitude is not reflected in execution time because of the high throughput of our network (32 Gbps).

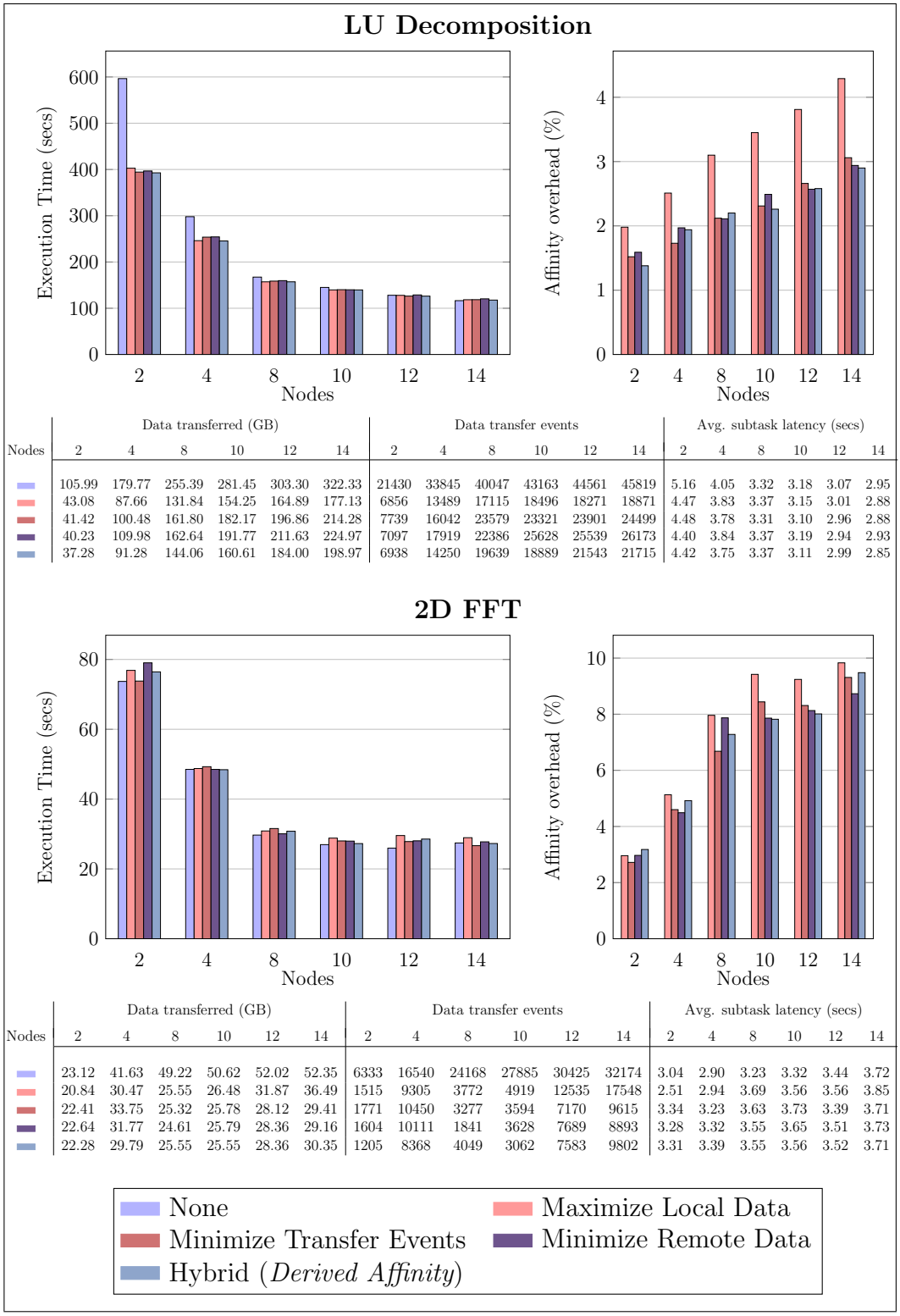
For the matrix multiplication experiment, we observe the maximum gain



Nodes	Data transferred (GB) 2	4	8	10	12	14	Data transfer events 2	4	8	10	12	14	Avg. subtask latency (secs) 2	4	8	10	12	14
	5.94	9.06	10.86	10.93	11.21	11.44	2381	5877	8238	8860	9113	9454	13.37	9.52	5.15	4.22	3.52	3.10
	0.20	0.27	0.30	0.39	0.43	0.49	2499	5020	7070	7512	7595	7747	14.69	12.29	5.73	4.45	3.58	3.04
	5.80	9.97	11.49	11.92	11.90	12.11	2241	5743	7992	9348	9595	10071	13.29	9.31	4.94	4.15	3.55	3.21
	0.17	0.28	0.30	0.34	0.41	0.43	2369	5502	7158	7294	7678	7768	14.73	12.15	5.57	4.45	3.66	3.11
	0.20	0.50	0.65	0.76	0.84	0.78	2475	5410	7015	7348	7493	7801	14.54	12.13	5.77	4.43	3.66	3.11

Nodes	Data transferred (GB) 2	4	8	10	12	14	Data transfer events 2	4	8	10	12	14	Avg. subtask latency (secs) 2	4	8	10	12	14
	32	96	220	274	328	371.5	103	265	668	812	980	1082	3.55	2.50	2.94	3.12	3.24	3.32
	28	67.5	139	182	234.5	271	75	235	311	402	499	572	2.15	2.72	2.46	2.52	2.60	2.81
	16	96	205	226	274.5	308.5	58	279	521	525	669	806	3.10	2.70	2.34	2.63	2.50	2.61
	17	96	208	237	275.5	301.5	69	284	541	570	648	743	2.58	2.36	2.74	2.62	2.66	2.88
	17	81	192	190.5	255.5	275	69	282	426	414	552	568	2.37	2.52	2.42	2.57	2.68	2.79

Figure 5.12: Locality aware scheduling



Nodes	Data transferred (GB) 2	4	8	10	12	14	Data transfer events 2	4	8	10	12	14	Avg. subtask latency (secs) 2	4	8	10	12	14
	105.99	179.77	255.39	281.45	303.30	322.33	21430	33845	40047	43163	44561	45819	5.16	4.05	3.32	3.18	3.07	2.95
	43.08	87.66	131.84	154.25	164.89	177.13	6856	13489	17115	18496	18271	18871	4.47	3.83	3.37	3.15	3.01	2.88
	41.42	100.48	161.80	182.17	196.86	214.28	7739	16042	23579	23321	23901	24499	4.48	3.78	3.31	3.10	2.96	2.88
	40.23	109.98	162.64	191.77	211.63	224.97	7097	17919	22386	25628	25539	26173	4.40	3.84	3.37	3.19	2.94	2.93
	37.28	91.28	144.06	160.61	184.00	198.97	6938	14250	19639	18889	21543	21715	4.42	3.75	3.37	3.11	2.99	2.85

Nodes	Data transferred (GB) 2	4	8	10	12	14	Data transfer events 2	4	8	10	12	14	Avg. subtask latency (secs) 2	4	8	10	12	14
	23.12	41.63	49.22	50.62	52.02	52.35	6333	16540	24168	27885	30425	32174	3.04	2.90	3.23	3.32	3.44	3.72
	20.84	30.47	25.55	26.48	31.87	36.49	1515	9305	3772	4919	12535	17548	2.51	2.94	3.69	3.56	3.56	3.85
	22.41	33.75	25.32	25.78	28.12	29.41	1771	10450	3277	3594	7170	9615	3.34	3.23	3.63	3.73	3.39	3.71
	22.64	31.77	24.61	25.79	28.36	29.16	1604	10111	1841	3628	7689	8893	3.28	3.32	3.55	3.65	3.51	3.73
	22.28	29.79	25.55	25.55	28.36	30.35	1205	8368	4049	3062	7583	9802	3.31	3.39	3.55	3.56	3.52	3.71

Figure 5.13: Locality aware scheduling (Contd.)

of 21.94% in the two node case with the *hybrid* heuristic. This result is attributed to a 88.24% reduction in data transfer, a 49.28% reduction in data transfer events and a 1.18 sec gain in average subtask latency. Note that this is a communication bound experiment and the reduction in data transfer has resulted in a large gain in performance. However, there is an observed overhead of 16.42% in affinity computation.

For the LU decomposition experiment, the *hybrid* heuristic performs better than others at most of the data points and we observe an average gain of 8.6% with this heuristic. On an average, this heuristic results in a 45.37% reduction in total data transfer and has a reported average overhead of 2.21%. The experiment has moderate performance gains as compared to data transfer savings as it is an iterative experiment, with a mix of compute and communication bound tasks per iteration.

The 2D FFT experiment does not gain much by affinity as the savings in data transfer are mitigated by high affinity computation overheads. The maximum gain for this experiment is observed with *transfer events* heuristic which, in the fourteen node case, yields a speed-up of 2.8% after accounting for 9.31% affinity computation overhead.

Among all our heuristics, the *derived affinity* scheme gives close to the best results (at most data points). The overhead of computing and using affinity is non-trivial, though. The results suggest that employing other methods of affinity computation (e.g. taking affinity as input from application or augmenting address space directory with affinity data all the time) might help in making the heuristics more useful. Further study on affinity and its overhead can be found in [4].

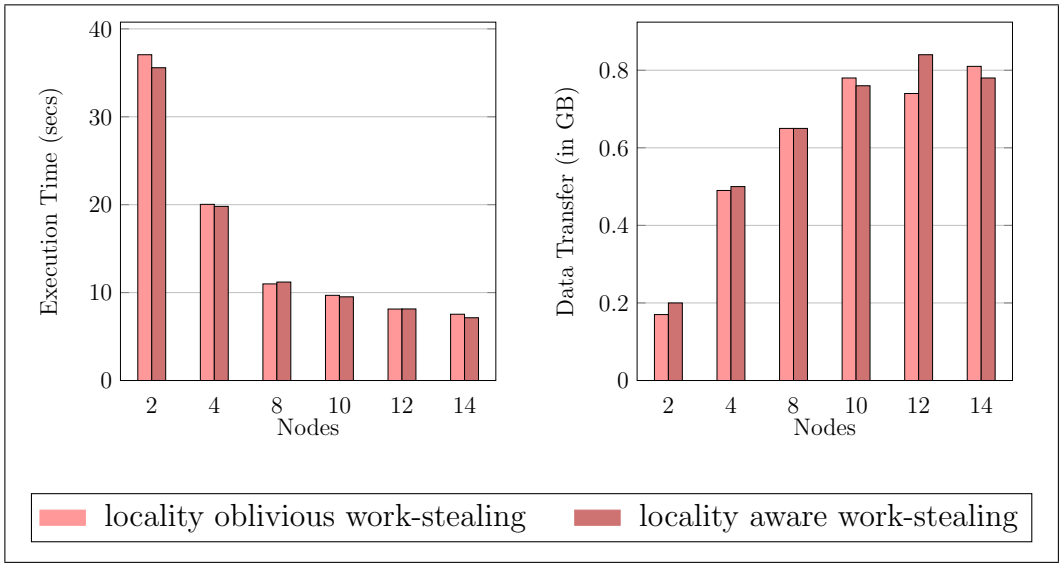


Figure 5.14: Image Convolution: Locality aware work-stealing (*Derived Affinity*)

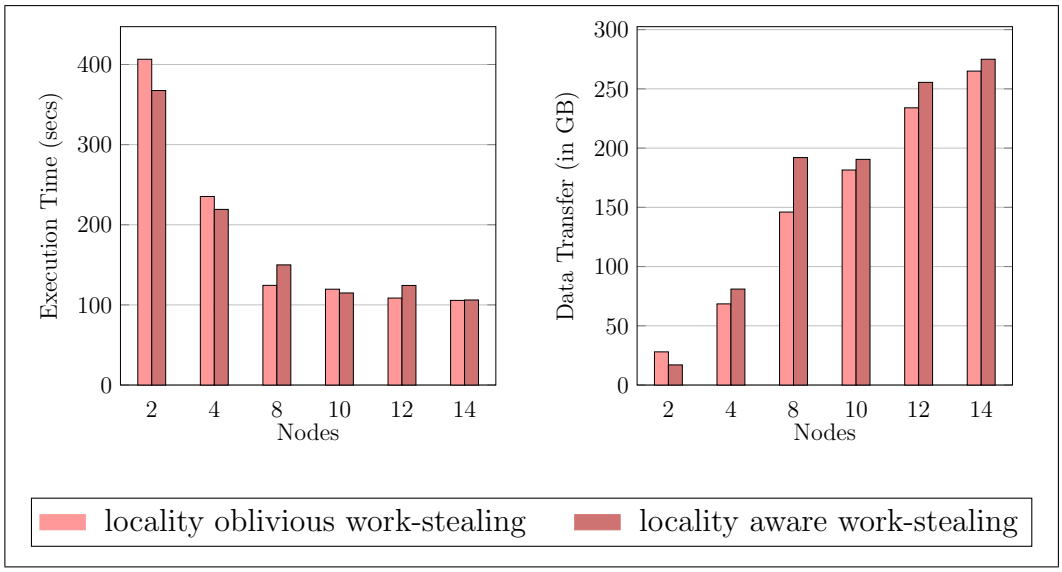


Figure 5.15: Matrix Multiplication: Locality aware work-stealing (*Derived Affinity*)

Our runtime also uses the computed affinity information while work stealing. We let the victim assign those subtasks to the stealer that are high on affinity scores for the stealer but low on affinity scores for the victim. Figures 5.14 and 5.15 plot performance numbers of locality oblivious work-stealing versus locality aware work-stealing for the *derived affinity* heuristic. For the image convolution experiment, we observe an average performance gain of 1.74% while the matrix multiplication experiment reports a flat response on an average with locality aware work-stealing (as compared to locality oblivious work-stealing). We do not observe much improvement with locality-aware work-stealing primarily because by the time stealing happens most of the subtasks have already executed and secondly because we use stale affinity information (computed at task start) for work-stealing and do not update this as the task progresses (see section 3.5.1.2.2).

5.4 Load Balancing

In this section, we study the effectiveness of our scheduler in achieving a balanced load on all cluster devices. Figures 5.16 and 5.17 respectively plot the finishing times of all cluster devices for the image convolution benchmark and for one iteration of the PageRank experiment. The figures also plot the number of subtasks executed by each of these devices. Note that all the CPU devices on a node are represented as work-groups numbered from $W1$ to $W14$. Similarly, GPU devices in the cluster are labelled $G1$ to $G28$.

Recall that for the matrix multiplication experiment, the entire input data is initially equally distributed randomly among all cluster nodes. For this

reason, all GPUs execute roughly the same number of subtasks (as they face similar data transfer overheads and subtasks are homogeneous). The same is true for CPU work-groups. For the PageRank experiment, however, the input data is resident on NFS. The graph plots 10th iteration of the experiment which means that the input data for the iteration additionally comes from different cluster nodes (as it is computed in last iteration). As such, variable input data latency is incurred by various CPU work-groups, causing them to execute different number of subtasks. Despite the disparity in subtask execution rate of GPUs and CPU work-groups, the finishing times of each of them is quite close to each other (for both experiments). This shows that our scheduler is able to balance the cluster load despite this device heterogeneity.

In another experiment (matrix multiplication on 10 nodes using GPUs only), we study the effectiveness of our load balancer (Figure 5.18) in the *centralized* initial data placement scheme (section 5.5.2). Since the entire data is located on the first node, our runtime schedules more subtasks on the two GPUs of this node as compared to the others. Since all 20 GPUs in the cluster finish at roughly the same time, load balance can be inferred.

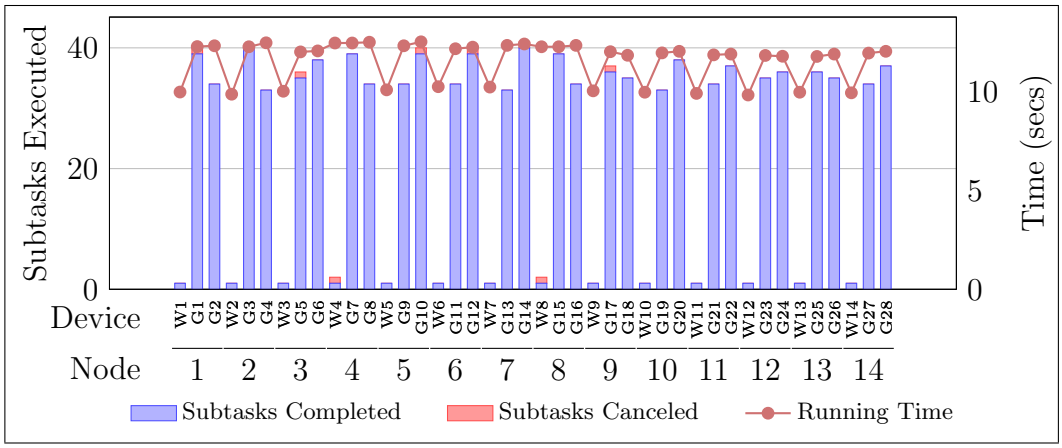


Figure 5.16: Load Balancing (Image Convolution) – W denotes a CPU work group and G denotes a GPU device – Block random data distribution

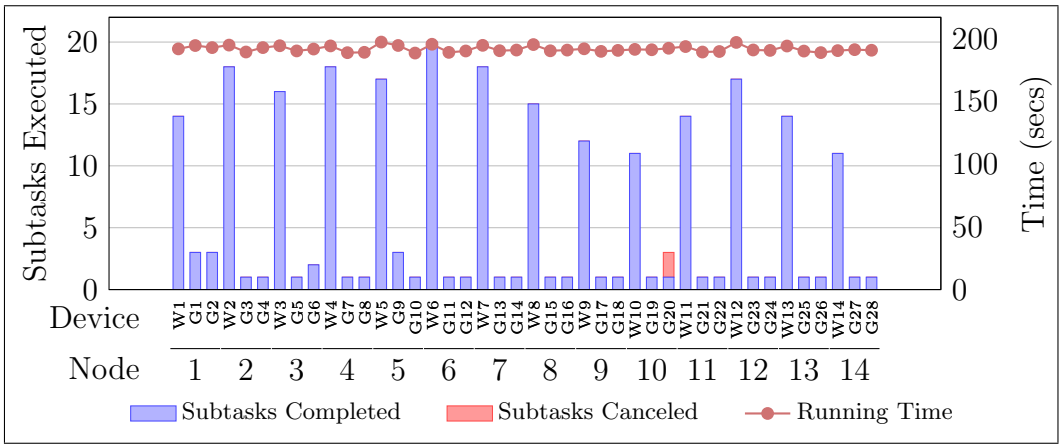


Figure 5.17: Load Balancing (Page Rank) – W denotes a CPU work group and G denotes a GPU device

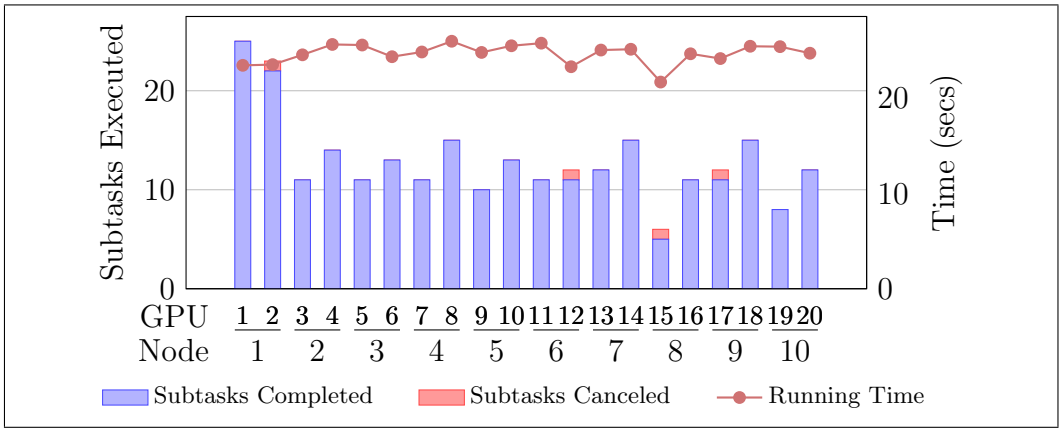


Figure 5.18: Load Balancing (Matrix Multiplication) – 32768 × 32768 matrices – Centralized data distribution

5.5 Stress Tests

In this section, we put our runtime under non-favorable conditions and study its response to various experiments. We study three things – firstly the response of our runtime when subtasks in the image convolution experiment are made to execute subtasks of different sizes, secondly the impact of changing the input data availability pattern before start of the experiment and thirdly

the impact of changing the size of subtasks employed in the experiment.

5.5.1 Heterogeneous Subtasks

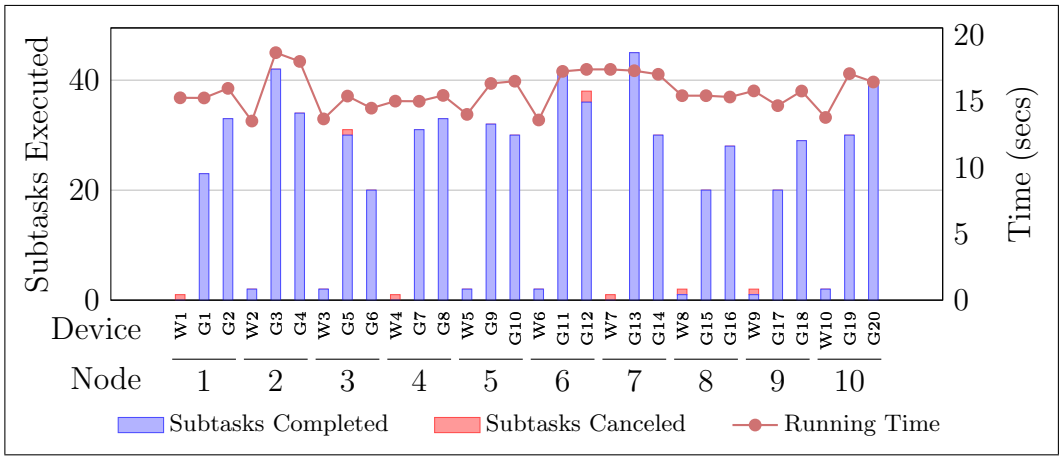


Figure 5.19: Load Balancing (Heterogeneous Subtasks) – W denotes a CPU work group and G denotes a GPU device

Figure 5.19 shows load balancing achieved by *Unicorn* when subtasks in Image Convolution experiment perform different amount of work. The top half of the image is convolved using 512 subtasks of size 2048 × 2048 while the bottom half is convolved using 128 subtasks of size 4096 × 4096. The experiment is executed on 10 nodes. Despite, the four fold execution disparity in subtasks, our runtime maintains a decent load balance in the cluster.

5.5.2 Input Data Distributions

In this section, we vary the placement of the initial input data in the address space(s). Figure 5.20 evaluates image convolution and matrix multiplication for various schemes like *centralized* (entire address space on one cluster node), *row random* (rows of 2048×2048 blocks placed randomly on all cluster nodes),

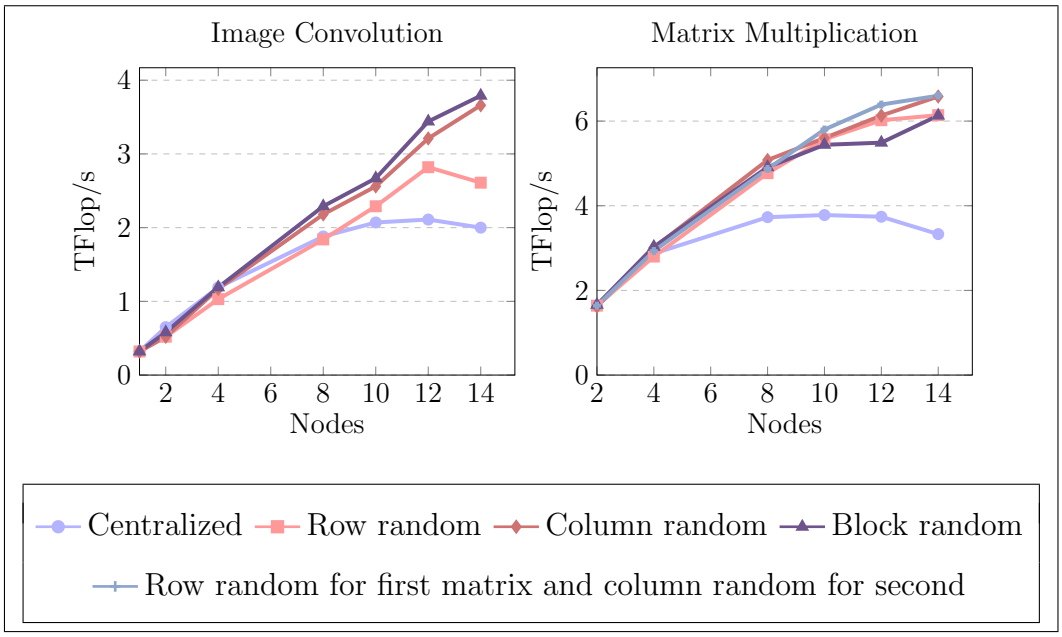


Figure 5.20: Impact of initial data distribution pattern

column random (columns of 2048×2048 blocks placed randomly on all cluster nodes) and *block random* (2048 × 2048 blocks placed randomly on any cluster node). Additionally, a fifth scheme is plotted for matrix multiplication where rows of 2048 × 2048 blocks for the first input matrix and columns of 2048 × 48 blocks for the second input matrix are placed randomly in the cluster. Results show that our runtime maintains performance despite the changes in data availability pattern. Only the *centralized* scheme behaves poorly as the network interface of the node containing the entire data becomes a bottleneck.

5.5.3 Varying Subtask Size

Figure 5.21 shows the response of our runtime to change in user's sizing of subtask. Within a reasonable range – (2048-8192) for matrix multiplication and (1024-4096) for image convolution – our system is able to maintain a

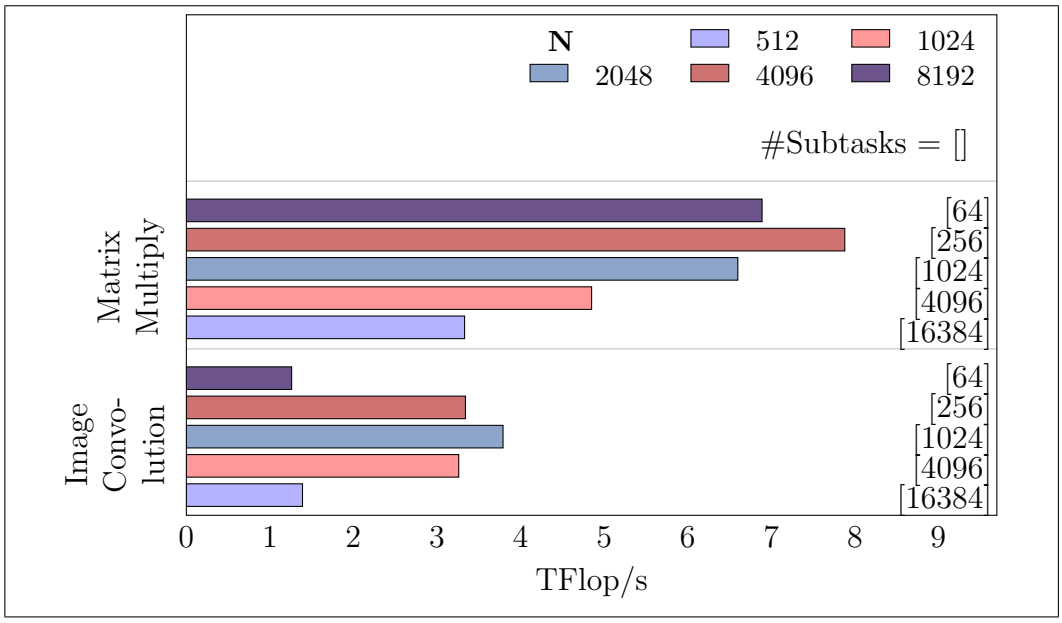


Figure 5.21: Subtask size ($N \times N$) – experiments executed on 14 nodes

throughput within 20% of the peak performance. On either side, system overheads begin to dominate. For extreme sizes, the throughput degrades as on one extreme there are too few subtasks to generate enough parallelism and on the other there are too many subtasks resulting in data transfers dominating the exploitable parallelism.

5.6 Unicorn Optimizations

In this section, we study the impact of two major Unicorn optimizations – *multi-assign* and *pipelining*. We study the overhead of *multi-assign* along with its performance benefits. This is followed by a study of performance gains achieved by creating a pipeline of subtasks, which enables communications of a few subtasks be overlapped with computations of others.

5.6.1 Multi-Assign

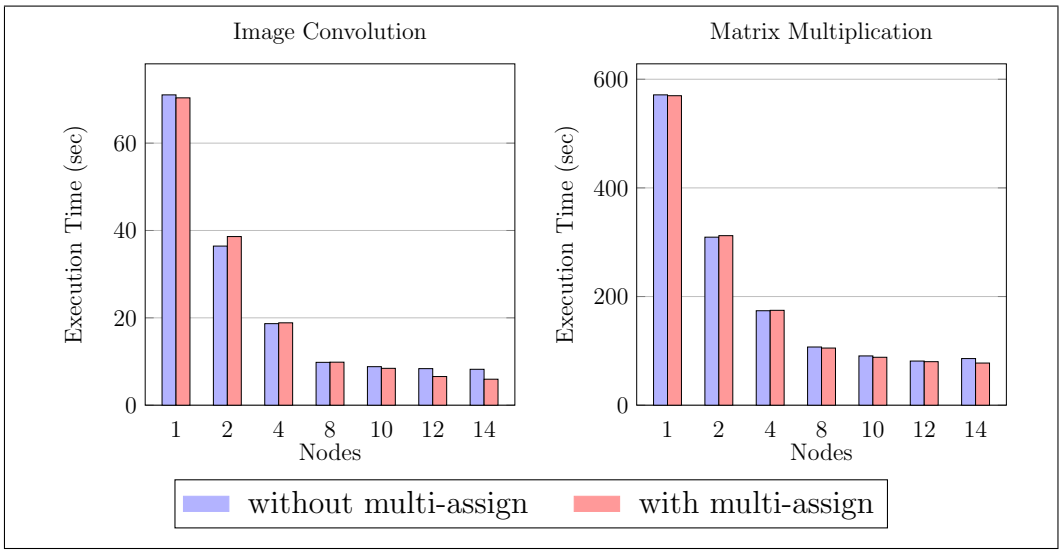


Figure 5.22: Multi-Assign (no external load)

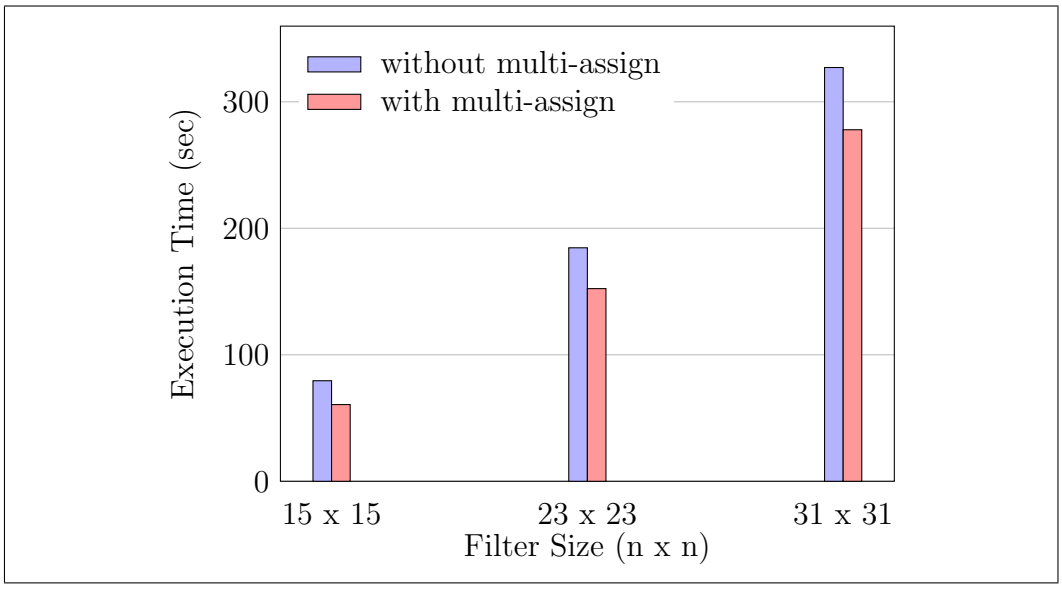


Figure 5.23: Multi-assign under external load (Image Convolution) – 4 nodes

We study two cases to understand the implications of *multi-assign*. First, under the absence of any external load we compare the performance of image convolution and matrix multiplication experiments with *multi-assign* enabled

to the case with *multi-assign* disabled. Second, we create an external load and study how image convolution behaves with and without *multi-assign*.

For the first case, results in Figure 5.22 show that there is no performance penalty in enabling *multi-assign*, in general. The observed average performance gain (with *multi-assign*) for image convolution and matrix multiplication respectively are 9.19% and 2.23% respectively (in comparison to no *multi-assign*). In fact, the performance gains increase with increasing number of nodes – for image convolution, *multi-assign* reports a maximum gain of 38.26% (over no *multi-assign*) in the fourteen node case. Similarly, the maximum performance gain of 10.62% is observed in the fourteen node case for matrix multiplication.

Next, we study multi-assignment over four nodes with the image convolution benchmark. We artificially overload one of the nodes with one process per core computing trigonometric functions indefinitely. In this case, we expect subtasks assigned to the overloaded node to be moved away from it. Our scheduler does this through stealing and multi-assignment. In the absence of multi-assignment a subtask may start running on a slow device and take a long time to finish, thereby, delaying the entire task. We run this test twice: once allowing multi-assignment and once preventing it. Results in Figure 5.23 show that with multi-assignment, subtasks of the overloaded cores get re-assigned and the task completes faster. Without it, the task remains bottlenecked by the 'slow' cores. Our heuristics generally only multi-assign fewer than 1% of the subtasks, but the later-assigned unit finishes first about 50% of the time. Of course, when one finishes, the other is aborted, leading to a faster overall time. The cancellation protocol itself has

insignificant overhead.

5.6.2 Pipelining

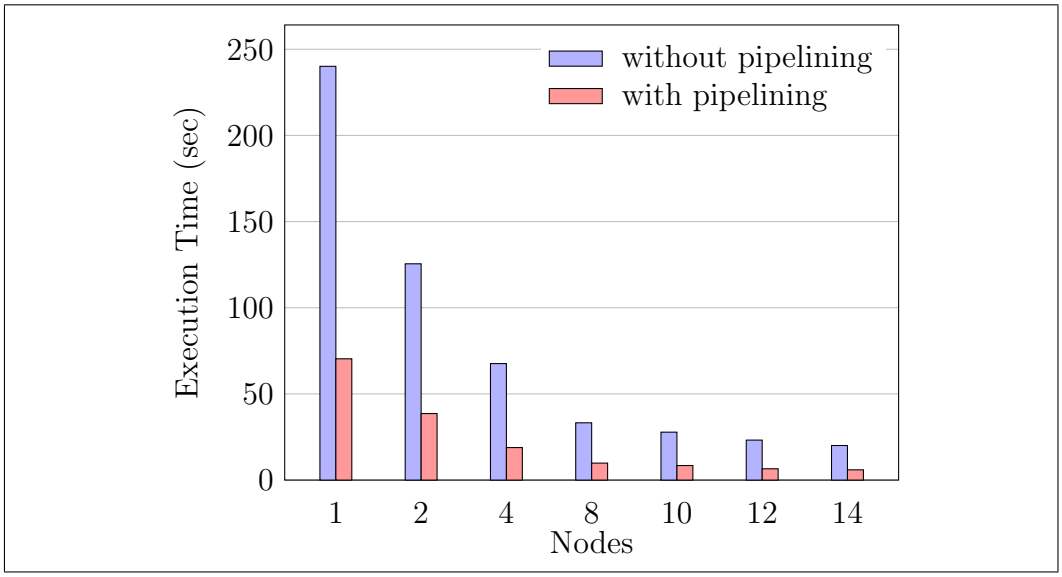


Figure 5.24: Pipelining (Image Convolution)

Figure 5.24 shows the impact of pipelining with the help of the image convolution experiment. With pipelining, our runtime overlaps computation of one subtask with the communication of the next. Further, on GPUs this makes multiple simultaneous kernel executions possible. Results show that our runtime achieves 3-4x speed-up with pipelining.

In [2], we study pipelining for an experimental 10-node cluster and report its effectiveness in hiding remote data access latency. If all required data were locally available on each of the 10 nodes, image convolution gets only 1% faster and matrix multiplication gets 31% faster. On the other hand, disabling pipelining makes them 2.29x and 1.19x slower, respectively.

5.6.3 Software cache for GPUs

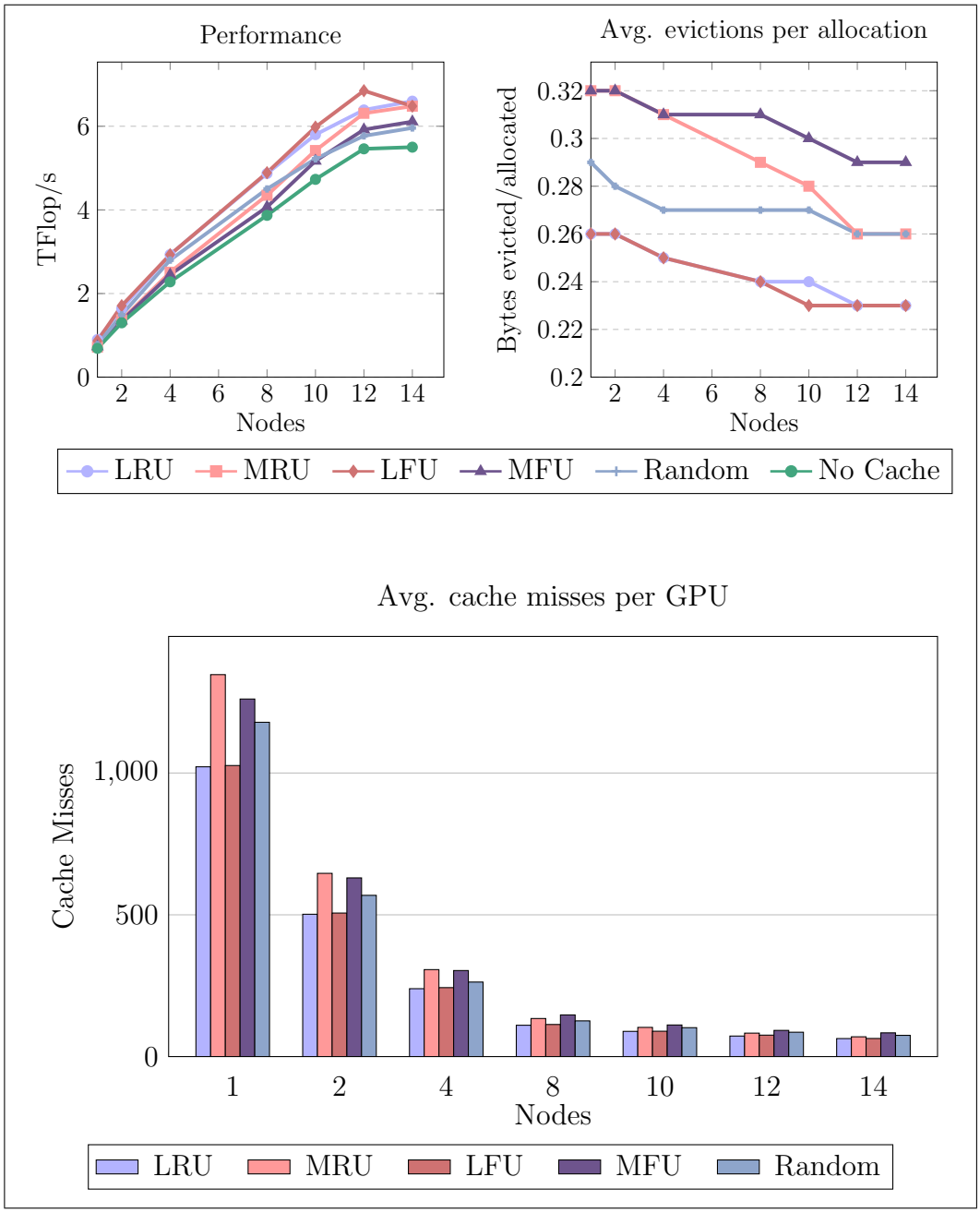


Figure 5.25: Matrix Multiplication – GPU Cache Eviction Strategies

Unicorn employs a software cache to reduce DMA data transfers to all GPUs in the cluster. The cache prevents read-only data shared by two or more

subtasks (of a task) executing on a GPU from being DMA'ed more than once. The cache also prevents data transfers in case a write-only or read-write data generated by a subtask is later consumed by a subtask of another task.

In this section, we study four cache eviction policies and compare their performances. The four policies are *least recently used* (which evicts the data of subtask that was used the earliest in time), *most recently used* (which evicts the data of subtask that was used the latest in time), *least frequently used* (which evicts the data of subtask that was used the minimum number of times) and *most frequently used* (which evicts the data of subtask that was used the maximum number of times). Figure 5.25 plots the performances of these policies where these are respectively denoted by shorthand notations LRU, MRU, LFU and MFU. Results show that both LRU and LFU perform better than MRU and MFU. This result can also be inferred from lesser number of cache evictions per allocation and lesser number of average cache misses in LRU and LFU as compared to MRU and MFU.

5.6.4 Data Compression

To ameliorate high data transfer latency in PageRank reduction, we compress data computed by subtasks before transferring over the network or from GPU to CPU. The employed compression algorithm is based on Run Length Encoding (section 10.1) and is executed on GPUs for GPU→CPU data transfers and on CPUs for inter-node data transfers. Figure 5.26 presents the cluster-wide size of uncompressed data (i.e. before compression) for reduction, compression ratio (i.e. the ratio of uncompressed size to compressed

size) and performance boost with compression (over the case when there is no compression). Results show that the performance gains increase with increase in the number of nodes and reach a peak of 23.81% for network compression and a peak of 2.45% for GPU→CPU compression (for the 14 node case). Note that because of lower latency of GPU→CPU transfers, the gain observed with GPU compression is moderate as compared to inter-node compression. Also, note that the amount of data transferred in the cluster increases with the number of nodes. However, this reduces the number of subtasks processed per node which means that inter-node reductions take place with more sparsity in data (and thus higher compression yield).

Nodes	Network reduction statistics			GPU→CPU reduction statistics		
	Uncompressed Data Size (GB)	Compression Ratio	Performance Gain (%)	Uncompressed Data Size (GB)	Compression Ratio	Performance Gain (%)
8	162.98	3.63	6.44	27.01	44.63	0.31
10	209.55	4.72	16.10	25.15	44.64	0.83
12	256.11	5.29	21.00	23.28	44.66	0.64
14	302.68	5.78	23.81	31.67	44.61	2.45

Figure 5.26: PageRank data compression (250 million web pages)

5.7 Overhead Analysis

In this section, we evaluate the overhead of our implementation of *Unicorn's* runtime firstly by varying the number of CPU cores used in the experiments and secondly by studying the amount of time experiments spend inside runtime's code versus the application code.

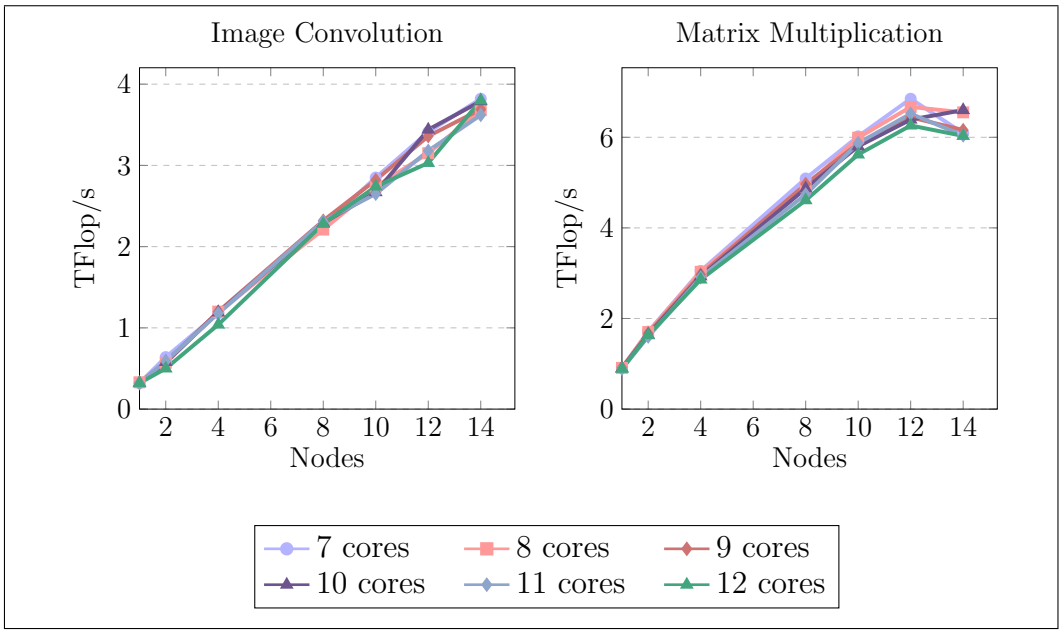


Figure 5.27: Varying CPU cores used in subtask computation

5.7.1 Varying CPU cores

CPU cores are not only used for subtask computations but for many other critical operations like CPU-GPU data transfers, network data transfers, scheduling and *Unicorn* runtime's control operations. For all our experiments presented in this section, we have reserved two CPU cores (out of 12 available) per node for these support functions and presented results by using the rest for subtask computations. Figure 5.27 varies the number of CPU cores allowed for the application subtasks per node from 7 to 12 and compares their performances. Results show that the performance of all these cases remain close to each other. The average difference between the maximum and minimum performing cases (at all data points) for both experiments is less than 10%.

This narrow performance difference is an indication of our runtime's low

overhead. Because of this low overhead, we currently do not dynamically vary the number of CPU cores used in a *Unicorn* task. An exploration of this is desired in future. However, we currently do allow applications to statically specify the number of CPU cores to be reserved for a task.

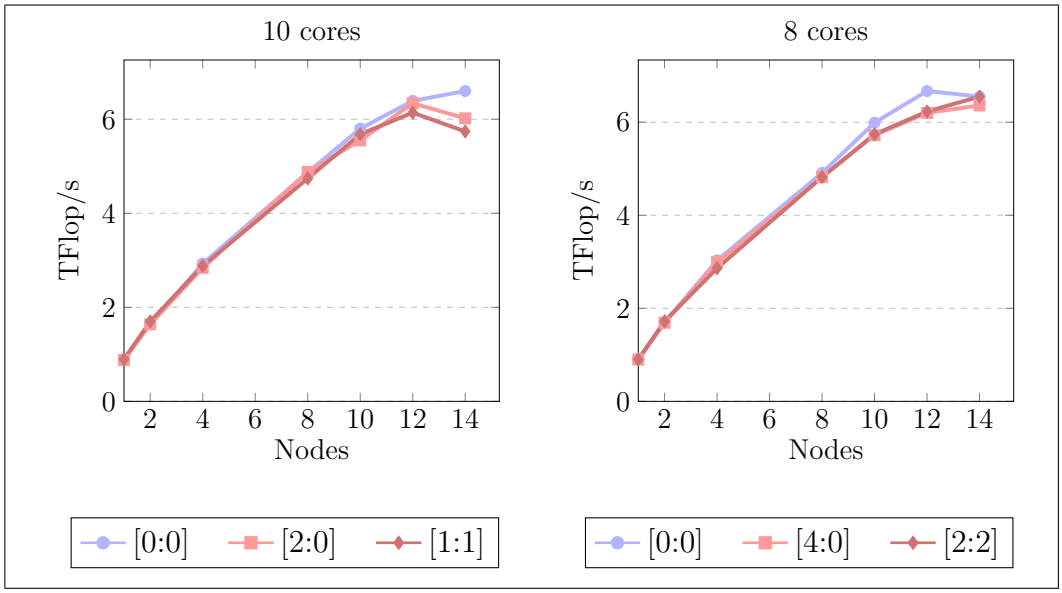


Figure 5.28: Matrix Multiplication – Varying CPU core affinity

We also study the impact of binding our *compute threads* (section 3) to CPU cores. In all the experiments presented thus far, we have not explicitly bound *compute threads* to processing cores, allowing the operating system to manage them. Figure 5.28 compares this scenario to the case when all our *compute threads* are explicitly bound to CPU cores (which leaves a few cores for other critical operations like data transfers). The figure plots the performance of the matrix multiplication experiment for two cases - when 10 cores and 8 cores are employed in subtask computation. For each case, we report performance when there is no explicit core for these critical operations (plotted as [0:0]). Relative to this, we plot performances of cases where [n:m] cores are designated for non-subtask computations (i.e. n cores are

explicitly freed from first CPU while m cores are explicitly freed from the second one). Results show that explicit binding of *compute threads* (to cores) does not have a significant performance impact. However, not explicitly binding threads and letting the operating system to freely migrate them yields best performance.

5.7.2 Unicorn Time versus Application Time

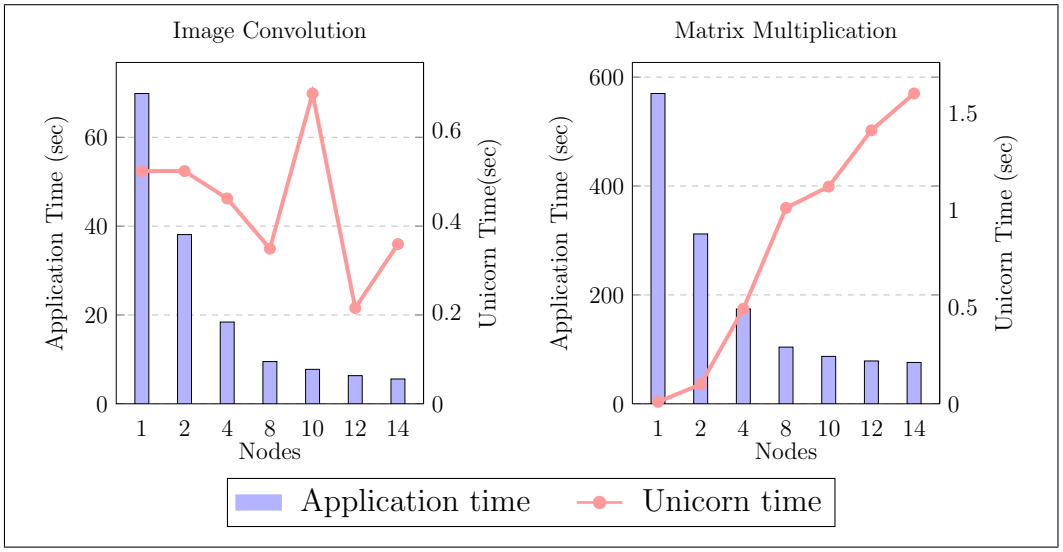


Figure 5.29: Library time versus application time

Figure 5.29 compares the time spent (by image convolution and matrix multiplication experiments) in runtime's code to the time spent in application execution. The latter includes the time taken for data transfers (both network and CPU-GPU) and callback executions. The rest of the experimental time is considered as our runtime's overhead. Results show that the average overhead for image convolution is 3.6% while it is 0.91% for the matrix multiplication experiment. In absolute value, the average overhead is 0.43 seconds for image convolution and 0.82 seconds for matrix multiplication.

5.7.3 Data Transfer Frequency

Nodes	Image Convolution			Matrix Multiplication			
	Avg. unique data sent per node (GB)	Avg. total data sent per node (GB)	Avg. transfer freq. per byte per node	Avg. unique data sent per node (GB)	Avg. total data sent per node (GB)	Avg. transfer freq. per byte per node	Avg. first matrix transfer freq. per byte per node
2	3.05	3.05	1.00	8.50	8.50	1.00	1.00
4	2.26	2.27	1.00	7.38	15.88	2.15	1.18
8	1.31	1.32	1.01	5.63	18.19	3.23	1.86
10	1.07	1.07	1.01	4.50	18.20	4.04	2.17
12	0.91	0.92	1.01	4.00	18.83	4.71	2.78
14	0.87	0.88	1.01	3.18	18.36	5.78	2.88

Figure 5.30: Data Transfer Frequency

In this section, we study the number of times each byte in the address space gets transferred in the cluster (between nodes). Two examples are considered – on one end is the image convolution experiment which has very little subscription overlap (fringe) among subtasks. At the other end is matrix multiplication where entire input data of every subtask overlaps with that of other subtasks. Figure 5.30 lists the average number of unique and total bytes transferred by every node in the cluster along with the average data transfer frequency (i.e., the ratio of total bytes transferred to unique bytes transferred per node). Results show that the data transfer frequency stays close to 1 for the image convolution experiment, whereas it grows with increasing number of nodes for the matrix multiplication experiment. This is because one of input matrices is required on all cluster nodes and because of the initial random placement of data, (n-1) transfers are required (where n is the number of nodes). The other matrix, however, exhibits relatively moderate transfer frequency (the last column in the table).

Most of the data transferred is only because it resides at a node different from

the subtask that requires it. For matrix multiplication, e.g., disabling stealing and multi-assignment reduces the unique data sent per node by 5.88% on average. Correspondingly, the total data sent and transfer frequency per byte decrease by 9.76% and 4.97% respectively. Thus, there is only 3.88% additional transfer due to re-sending of data to the new destination after stealing or multi-assignment. The rest 96.12% data transfer is due to subscription overlap among subtasks.

5.8 Unicorn versus others

Figure 5.31 compares the performance of our matrix multiplication experiment (single node) to StarPU [6]. The three bars for StarPU plot, respectively, its default *eager* scheduler, the first run of its advanced *dmdas* scheduler (i.e., without calibration; this scheduler requires calibration runs for optimal performance), and the best run out of three successive runs of *dmdas* after calibration. This best run performs better than *Unicorn* until a matrix size of 16384×16384. For larger matrices, it starts to lag Unicorn and eventually fails at 65536×65536 reporting it ran out of memory. Yet, *Unicorn* runs at this and even higher sizes.

Figure 5.32 compares *Unicorn* to SUMMA[58] for multiplication of two square matrices (on CPUs only) of size 32768×32768. Results show that *Unicorn* performs quite close to SUMMA (which is hand tuned for matrix multiplication) for this double precision computation. Note that SUMMA incurs a bit of overhead in the sense that it requires a different MPI process per CPU core whereas *Unicorn* works with one MPI process per node. For this experiment,

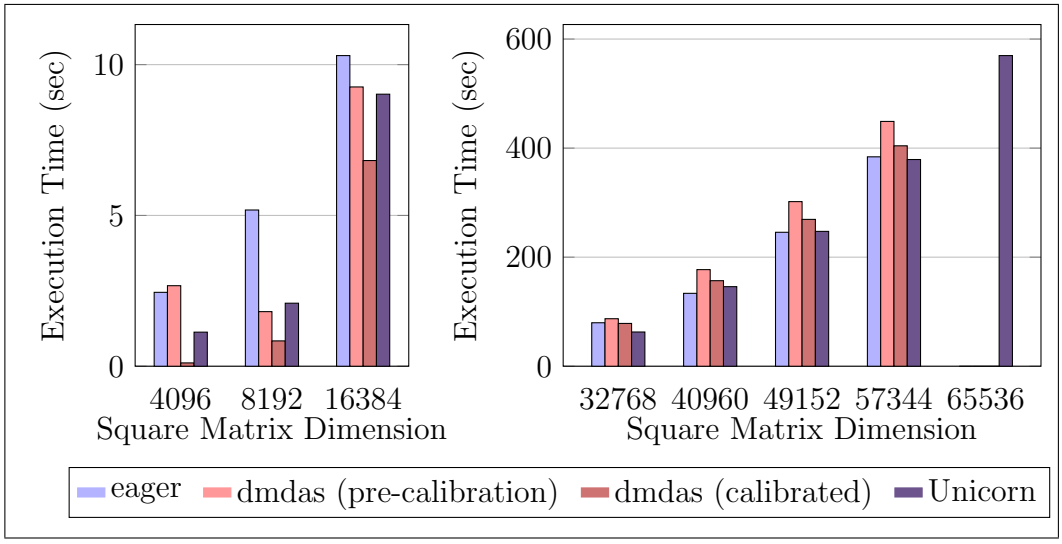


Figure 5.31: Unicorn versus StarPU

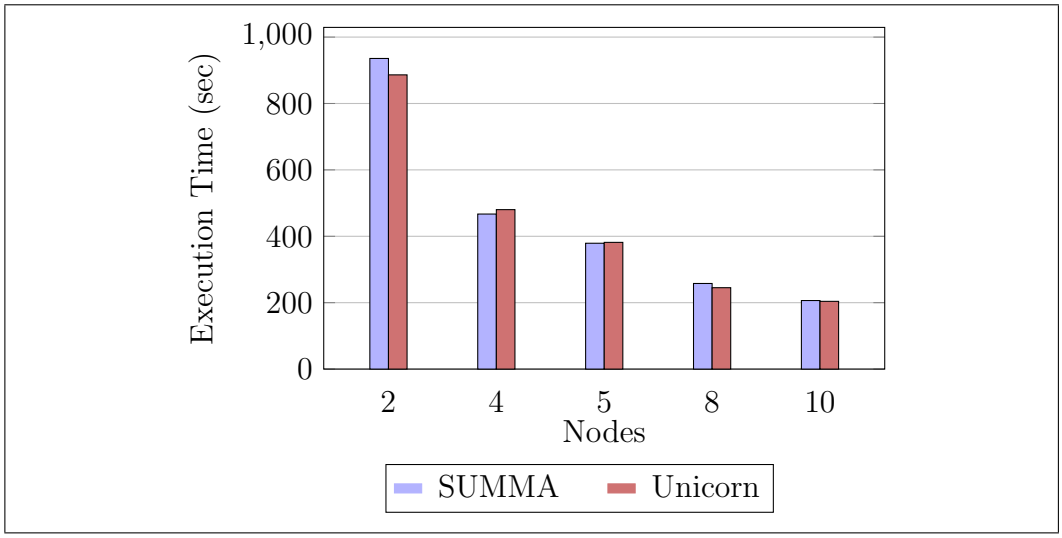


Figure 5.32: Unicorn versus SUMMA

we have used different block sizes for both implementations (1024×1024 for *Unicorn* and 128×128 for SUMMA) in order to compare their best CPU performances.

Chapter 6

Application Profiling

```
Parallel Task 6 Execution Time = 15.7919 [Scheduling Policy: WS]

Subtask distribution for task [0, 1] ...

Device Subtask Execution Profile ...
Device 0 Subtasks 4
Device 12 Subtasks 108
Device 13 Subtasks 56
Device 14 Subtasks 3
Device 26 Subtasks 90
Device 27 Subtasks 75

Machine Subtask Execution Profile ...
Machine 0 Subtasks 168 CPU Subtasks 4
Machine 1 Subtasks 168 CPU Subtasks 3

Total Acknowledgements Received 336

Address Space [0, 1] memory transfer statistics on [Host 0] ...
1235888826 bytes memory received in 155 events
1236443811 bytes memory transferred in 163 events

Address Space [0, 2] memory transfer statistics on [Host 0] ...
0 bytes memory received in 0 events
0 bytes memory transferred in 0 events
```

Figure 6.1: Sample Unicorn Logs (part 1)

Unicorn provides compile time controls for profiling various modules of the library. The resultant logs help identify application and runtime bottlenecks. Among others, the logs dump address space allocations and their data transfers across cluster nodes, work-stealing and multi-assign statistics, task and subtask execution timeline, MPI calls issued at each cluster node, node affinity statistics, load balancing and scheduling information. The logs are dumped out in textual format like the ones in figures 6.1, 6.2 and 6.3.

Logs in Figure 6.1 show the total experimental time taken under the default work stealing scheduling policy and the number of subtasks executed by each

```
1  Task Profiler [Host 0] ...........

2  INPUT_MEMORY_TRANSFER => Accumulated Time: 20515.3s; Actual Time =
       6.86013s; Overlapped Time = 20508.4s
3  OUTPUT_MEMORY_TRANSFER => Accumulated Time: 0s; Actual Time = 0s;
       Overlapped Time = 0s
4  TOTAL_MEMORY_TRANSFER => Accumulated Time: 20517.9s; Actual Time =
       6.86123s; Overlapped Time = 20511.1s
5  DATA_PARTITIONING => Accumulated Time: 0.406627s; Actual Time =
       0.376811s; Overlapped Time = 0.0298162s
6  SUBTASK_EXECUTION => Accumulated Time: 137.482s; Actual Time =
       15.324s; Overlapped Time = 122.158s
7  DATA_REDUCTION => Accumulated Time: 0s; Actual Time = 0s;
       Overlapped Time = 0s
8  DATA_REDISTRIBUTION => Accumulated Time: 0s; Actual Time = 0s;
       Overlapped Time = 0s
9  MEMORY_COMMIT => Accumulated Time: 0s; Actual Time = 0s;
       Overlapped Time = 0s
10 SUBTASK_STEAL_WAIT => Accumulated Time: 8.32788s; Actual Time =
       6.59817s; Overlapped Time = 1.72971s
11 SUBTASK_STEAL_SERVE => Accumulated Time: 0.000205755s; Actual Time
       = 0.000205755s; Overlapped Time = 0s
12 STUB_WAIT_ON_NETWORK => Accumulated Time: 8.46128s; Actual Time =
       4.97415s; Overlapped Time = 3.48713s
13 COPY_TO_PINNED_MEMORY => Accumulated Time: 0.700807s; Actual Time
       = 0.663236s; Overlapped Time = 0.037571s
14 COPY_FROM_PINNED_MEMORY => Accumulated Time: 1.1685s; Actual Time
       = 1.1685s; Overlapped Time = 0s
15 CUDA_COMMAND_PREPARATION => Accumulated Time: 0.00107765s; Actual
       Time = 0.00107765s; Overlapped Time = 0s
16 UNIVERSAL => Accumulated Time: 20679.8s; Actual Time = 15.3001s;
       Overlapped Time = 20664.5s
```

Figure 6.2: Sample Unicorn Logs (part 2)

machine and device (CPU/GPU) in the cluster. The figure also shows the amount of memory sent or received (and the number of send/receive events) for every address space used on a node in the cluster.

Figure 6.2 shows the work done (*Accumulated Time*) and the wall clock time spent (*Actual Time*) in various events like memory transfers, data subscription, stealing, copy to/from pinned buffers, etc. The work done is the arithmetic sum of the times spent by various threads while the wall clock time accounts for the time overlapped between these threads only once. The common time between threads is displayed under the heading *Overlapped Time*.

Figure 6.3 shows the cumulative memory transfers (across all address spaces) for a task on a node. Beneath that, the subtask execution rate for each stub

```
1  Task Exec Stats [Host 0] ............

2  Memory Transfers   Received = 1235888826 bytes; Receive Events =
       155; Sent = 1236443811 bytes; Send Events = 163
3  Scattered Memory Transfers   Received = 1235888826 bytes; Receive
       Events = 155; Sent = 1236443811 bytes; Send Events = 163

4  Device 0   Subtask execution rate = 0.0386306; Steal attempts = 2;
       Successful steals = 0; Failed steals = 2; Pipelines across
       ranges = 0
5  Device 12   Subtask execution rate = 7.05162; Steal attempts = 3;
       Successful steals = 1; Failed steals = 2; Pipelines across
       ranges = 1
6  Device 13   Subtask execution rate = 4.131; Steal attempts = 2;
       Successful steals = 0; Failed steals = 2; Pipelines across
       ranges = 0


7  Unicorn [Host 0] Event Timeline Device 0
8  Task [0, 1] Subtask 281 (Split 0 of 12) 41.2708 43.8583
9  Task [0, 1] Subtask 281 Event SubtaskExecution 41.4977 43.858
10 Task [0, 1] Subtask 281 Event WaitOnNetwork 41.2789 41.497
11 Task [0, 1] Subtask 325 (Split 0 of 12) 43.8586 46.797
12 Task [0, 1] Subtask 325 Event SubtaskExecution 44.2574 46.7967
13 Task [0, 1] Subtask 325 Event WaitOnNetwork 43.942 44.2572

14 Unicorn [Host 0] Event Timeline Device 12
15 Task [0, 1] Subtask 11 43.1959 44.7673
16 Task [0, 1] Subtask 11 Event CopyFromPinnedMemory 44.6632 44.7669
17 Task [0, 1] Subtask 11 Event CopyToPinnedMemory 43.3631 43.3665
18 Task [0, 1] Subtask 11 Event SubtaskExecution 43.3668 43.3672
19 Task [0, 1] Subtask 32 43.6812 45.0229
20 Task [0, 1] Subtask 32 Event CopyFromPinnedMemory 44.8802 44.8843
21 Task [0, 1] Subtask 32 Event CopyToPinnedMemory 43.6829 43.6867
22 Task [0, 1] Subtask 32 Event SubtaskExecution 43.687 43.6875
23 Task [0, 1] Subtask 33 41.5666 43.8472
24 Task [0, 1] Subtask 33 Event CopyFromPinnedMemory 43.6878 43.8468
25 Task [0, 1] Subtask 33 Event CopyToPinnedMemory 41.6809 41.6841
26 Task [0, 1] Subtask 33 Event SubtaskExecution 41.6843 41.6847
27 Task [0, 1] Subtask 33 Event WaitOnNetwork 41.5898 41.6744
```

Figure 6.3: Sample Unicorn Logs (part 3)

is reported. This is accompanied by the total number of steal attempts by a device, the number of successful ones (out of the total) and in case the stealer is a GPU device, the figure also reports the number of steal attempts that were able to bring in new subtasks before the GPU pipeline stalled. Lastly, the figure displays the event timeline, i.e., the start and end times for various events happening on each device in the cluster. These events include subtask execution, wait for remote data transfers, time spent in transfers to/from pinned buffers for GPUs, etc.

Unicorn has an analysis engine that can consume these textual logs as input and generate easily understandable graphs. The graphs include performance and scalability charts, CPU versus GPU versus CPU+GPU charts, load balancing, multi-assign and event timeline charts. A few sample charts are shown in figures 6.4, 6.5, 6.6 and 6.7.

Figure 6.4 depicts the performance improvements in a sample experiment when the number of nodes used in the cluster are increased from 2 to 10. Figure 6.5 shows the load balance achieved by various GPUs in the cluster. Figure 6.6 compares the centralized work-sharing scheduler (*Push*) described in section 5.3 with the *Unicorn's* work-stealing scheduler (*Pull*). In both cases, there is a plot with compute communication overlap enabled and a plot with this optimization disabled. Finally, Figure 6.7 shows activities of cluster devices over a timeline during the experiment.

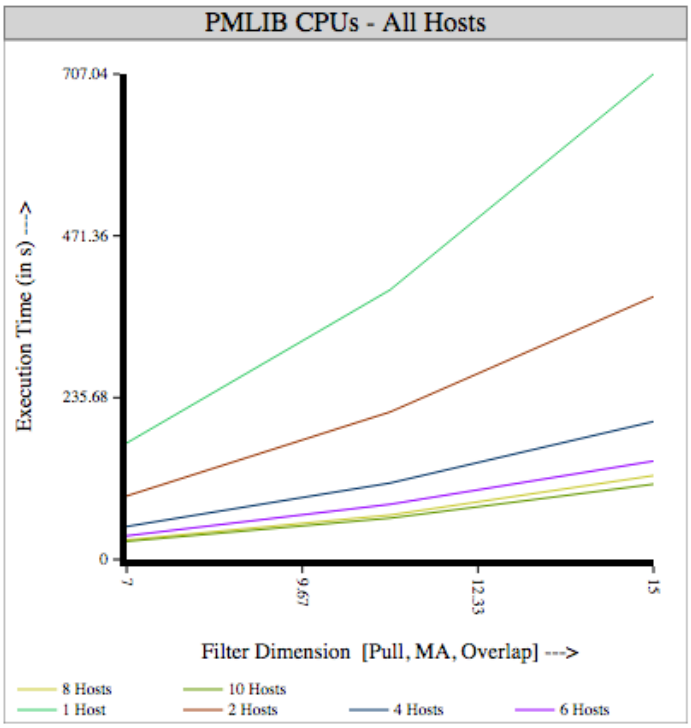


Figure 6.4: Performance

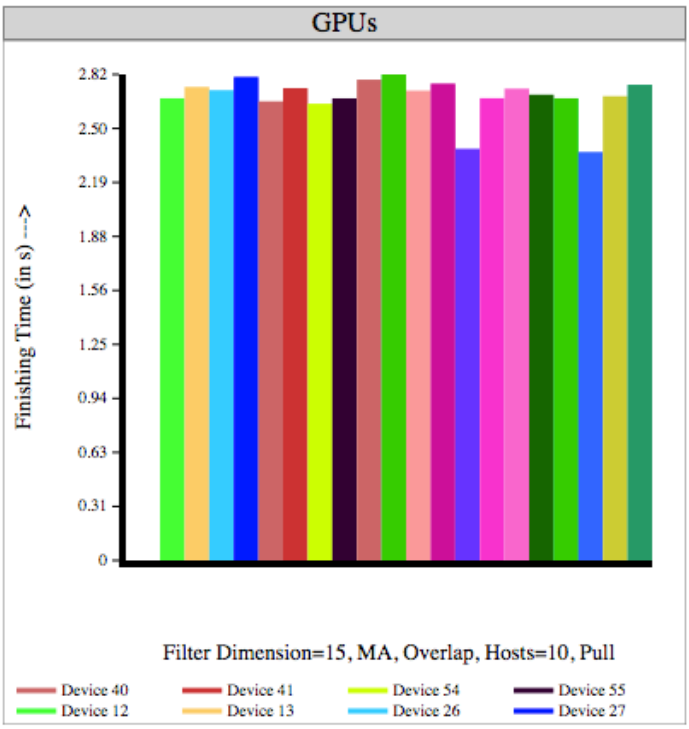


Figure 6.5: Load Balance

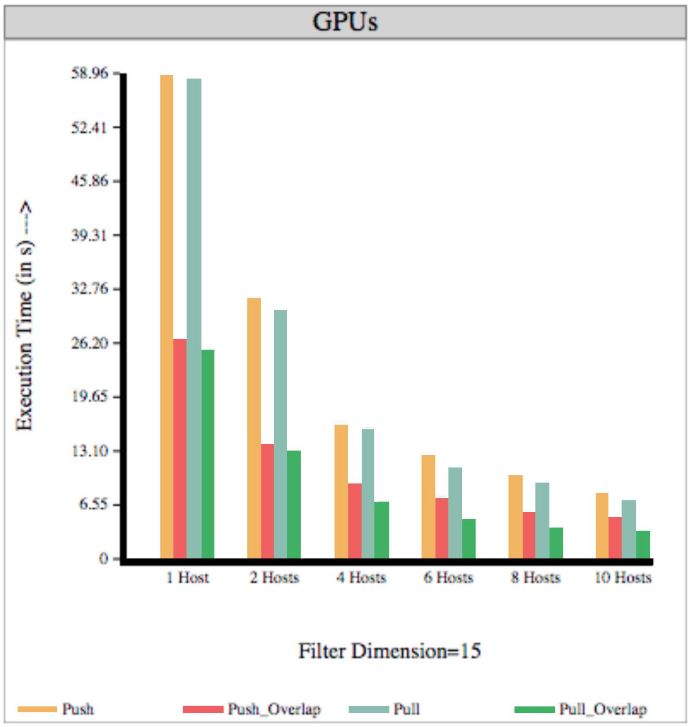


Figure 6.6: Compute Communication Overlap

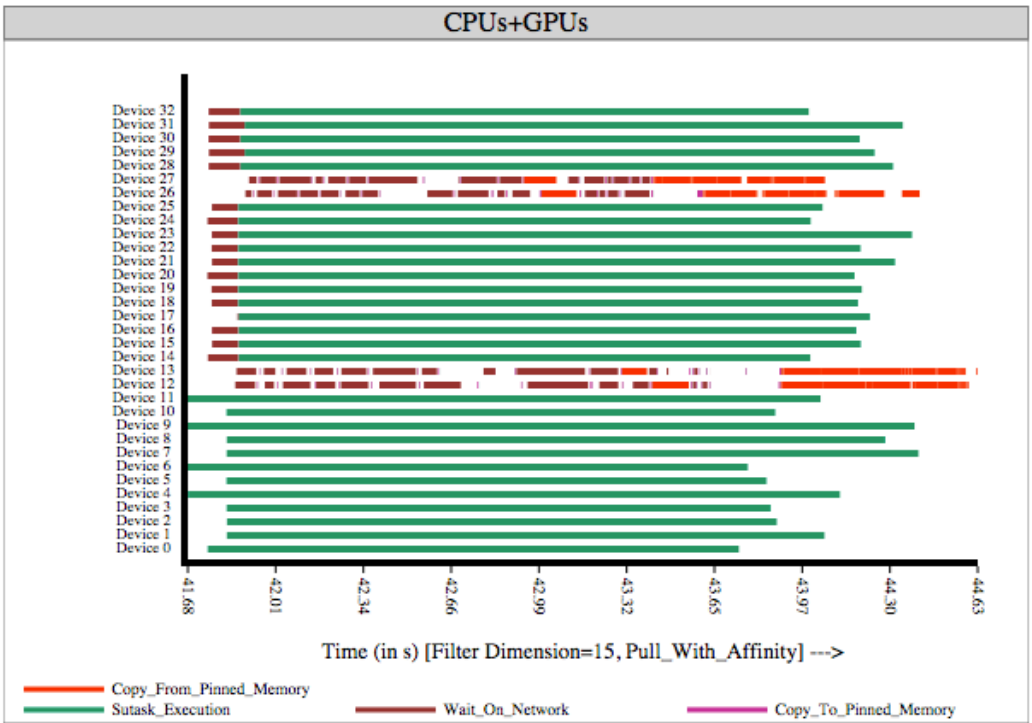


Figure 6.7: Event Timeline

Chapter 7

Public API

This section documents Unicorn's public headers - *pmPublicDefinitions.h* (Listing 7.1) and *pmPublicUtilities.h* (Listing 7.2). The prefix *pm* is taken from code name (*PMLIB* which is short for Partitioned Memory Library) of this project.

```
namespace pm
{
    typedef unsigned short int ushort;
    typedef unsigned int uint;
    typedef unsigned long ulong;

    const size_t MAX_NAME_STR_LEN = 256;
    const size_t MAX_DESC_STR_LEN = 1024;
    const size_t MAX_CB_KEY_LEN = 128;
    const size_t MAX_MEM_SECTIONS_PER_TASK = 8;

    /**
     * This enumeration defines success and all
       error conditions for the PMLIB Application
       Programming Interface (API).
     * Applications can depend upon these status
       flags to know the outcome of PMLIB functions.
     * Applications may use pmGetLastError for a
       brief description of the error.
    */
    typedef enum pmStatus
    {
        pmSuccess = 0,
        pmOk,
```

```
19          pmStatusUnavailable,
20          pmFatalError,
21          pmInitializationFailure,
22          pmNetworkInitError,
23          pmNetworkTerminationError,
24          pmInvalidIndex,
25          pmInvalidCommand,
26          pmThreadingLibraryFailure,
27          pmTimerFailure,
28          pmMemoryError,
29          pmNetworkError,
30          pmIgnorableError,
31          pmGraphicsCardError,
32          pmBeyondComputationalLimits,
33          pmUnrecognizedMemory,
34          pmInvalidKey,
35          pmMaxKeyLengthExceeded,
36          pmDataProcessingFailure,
37          pmNoCompatibleDevice,
38          pmConfFileNotFound,
39          pmInvalidOffset,
40          pmInvalidCallbacks,
41          pmUserError,
42          pmMaxStatusValues
43      } pmStatus;


44      /** This function returns the PMLIB's version
           number as a char* in the format
           MajorVersion_MinorVersion_Update */
45      const char* pmGetLibVersion();

46      /** This function returns a brief description of
           the last error (if any) caused by execution
           of any PMLIB function */
47      const char* pmGetLastError();

48      /** This function initializes the PMLIB library.
           It must be the first PMLIB API called on all
           machines under MPI cluster. */
49      pmStatus pmInitialize();
```

```
50      /** This function marks the termination of use
           of PMLIB in an application. This must be the
           application's last call to PMLIB. */
51      pmStatus pmFinalize();

52      /** This function returns the id of the calling
           host */
53      uint pmGetHostId();

54      /** This function returns the total number of
           hosts */
55      uint pmGetHostCount();


56      /** Some basic type definitions */
57      typedef void* pmMemHandle;
58      typedef void* pmRawMemPtr;
59      typedef void* pmTaskHandle;
60      typedef void* pmDeviceHandle;
61      typedef void* pmCallbackHandle;
62      typedef void* pmClusterHandle;

63      typedef enum pmMemType
64      {
65          READ_ONLY,
66          WRITE_ONLY,
67          READ_WRITE,
68          READ_ONLY_LAZY,
69          WRITE_ONLY_LAZY,
70          READ_WRITE_LAZY,
71          MAX_MEM_TYPE
72      } pmMemType;

73      typedef enum pmSubscriptionType
74      {
75          READ_SUBSCRIPTION,
76          WRITE_SUBSCRIPTION,
77          READ_WRITE_SUBSCRIPTION
78      } pmSubscriptionType;

79      typedef enum pmSubscriptionVisibilityType
80      {
```

```
        SUBSCRIPTION_NATURAL,    // Disjoint subscriptions are mapped at same distances as in task memory (default option)
        SUBSCRIPTION_COMPACT,    // Disjoint subscriptions are mapped back to back
        SUBSCRIPTION_OPTIMAL,    // Either SUBSCRIPTION_NATURAL or SUBSCRIPTION_COMPACT whichever is optimal for the subtask
        MAX_SUBSCRIPTION_VISBILITY_TYPE
    } pmSubscriptionVisibilityType;

    /** Structures for memory subscription */
    typedef struct pmSubscriptionInfo
    {
        size_t offset;  /* Offset from the start of the memory region */
        size_t length;  /* Number of bytes to be subscribed */

        pmSubscriptionInfo();
        pmSubscriptionInfo(size_t, size_t);
    } pmSubscriptionInfo;

    typedef struct pmScatteredSubscriptionInfo
    {
        size_t offset;
        size_t size;
        size_t step;
        size_t count;

        pmScatteredSubscriptionInfo();
        pmScatteredSubscriptionInfo(size_t, size_t, size_t, size_t);
    } pmScatteredSubscriptionInfo;

    /** GPU context for subtask */
    typedef struct pmGpuContext
    {
        void* scratchBuffer;
        void* reservedGlobalMem;
```

```
108         pmGpuContext();
109     } pmGpuContext;

110     /** User information typedefs */
111     typedef struct pmSplitInfo
112     {
113         uint splitId;
114         uint splitCount;

115         pmSplitInfo();
116         pmSplitInfo(uint, uint);
117     } pmSplitInfo;

118     typedef struct pmMemInfo
119     {
120         pmRawMemPtr ptr;
121         pmRawMemPtr readPtr;
122         pmRawMemPtr writePtr;
123         size_t length;
124         pmSubscriptionVisibilityType visibilityType;

125         pmMemInfo();
126         pmMemInfo(pmRawMemPtr, pmRawMemPtr,
                pmRawMemPtr, size_t);
127         pmMemInfo(pmRawMemPtr, pmRawMemPtr,
                pmRawMemPtr, size_t,
                pmSubscriptionVisibilityType);
128     } pmMemInfo;

129     typedef struct pmSubtaskInfo
130     {
131         ulong subtaskId;
132         pmMemInfo memInfo[MAX_MEM_SECTIONS_PER_TASK];
133         uint memCount;
134         pmGpuContext gpuContext;
135         pmSplitInfo splitInfo;

136         pmSubtaskInfo();
137         pmSubtaskInfo(ulong, pmMemInfo*, uint);
138     } pmSubtaskInfo;
```

```
139     typedef struct pmTaskInfo
140     {
141         pmTaskHandle taskHandle;
142         void* taskConf;
143         uint taskConfLength;
144         ulong taskId;
145         ulong subtaskCount;
146         ushort priority;
147         uint originatingHost;

148         pmTaskInfo();
149     } pmTaskInfo;

150     typedef struct pmDataTransferInfo
151     {
152         pmMemHandle memHandle;
153         size_t memLength;
154         size_t* operatedMemLength;   // Mem Length
               after programmer's compression/encryption
155         pmMemType memType;
156         uint srcHost;
157         uint destHost;

158         pmDataTransferInfo();
159     } pmDataTransferInfo;

160     typedef enum pmDeviceType
161     {
162         CPU = 0,
163     #ifdef SUPPORT_CUDA
164         GPU_CUDA,
165     #endif
166         MAX_DEVICE_TYPES
167     } pmDeviceType;

168     typedef struct pmDeviceInfo
169     {
170         pmDeviceHandle deviceHandle;
171         char name[MAX_NAME_STR_LEN];
172         char description[MAX_DESC_STR_LEN];
173         pmDeviceType deviceType;
174         uint host;
```

```
175         uint deviceIdOnHost;
176         uint deviceIdInCluster;

177         pmDeviceInfo();
178     } pmDeviceInfo;

179     typedef enum pmSchedulingPolicy
180     {
181         SLOW_START,
182         RANDOM_STEAL,    /* default policy */
183         RANDOM_STEAL_WITH_AFFINITY,
184         EQUAL_STATIC,
185         PROPORTIONAL_STATIC,
186         NODE_EQUAL_STATIC
187     } pmSchedulingPolicy;

188     typedef enum pmAffinityCriterion
189     {
190         MAXIMIZE_LOCAL_DATA,
191         MINIMIZE_REMOTE_SOURCES,
192         MINIMIZE_REMOTE_TRANSFER_EVENTS,
193         MINIMIZE_REMOTE_TRANSFERS_ESTIMATED_TIME,
194         DERIVED_AFFINITY,
195         MAX_AFFINITY_CRITERION
196     } pmAffinityCriterion;

197     /* The lifetime of scratch buffer for a subtask
           */
198     typedef enum pmScratchBufferType
199     {
200         PRE_SUBTASK_TO_SUBTASK,          // Scratch
               buffer lives from data distribution
               callback to subtask callback
201         SUBTASK_TO_POST_SUBTASK,         // Scractch
               buffer lives from subtask callback to
               data redistribution/reduction callback
202         PRE_SUBTASK_TO_POST_SUBTASK,     // Scratch
               buffer lives from data distribution
               callback to data redistribution/reduction
               callback
203         REDUCTION_TO_REDUCTION           // Scratch
               buffer lives and travels from one data
```

```
            reduction callback to the next (even
            across machines)
204     } pmScratchBufferType;

205     typedef struct pmRedistributionMetadata
206     {
207         uint order;
208         uint count;

209         pmRedistributionMetadata();
210         pmRedistributionMetadata(uint, uint);
211     } pmRedistributionMetadata;


212     /** The following type definitions stand for the
           callbacks implemented by the user programs.*/
213     typedef pmStatus
           (*pmDataDistributionCallback)(pmTaskInfo
           pTaskInfo, pmDeviceInfo pDeviceInfo,
           pmSubtaskInfo pSubtaskInfo);
214     typedef pmStatus
           (*pmSubtaskCallback_CPU)(pmTaskInfo
           pTaskInfo, pmDeviceInfo pDeviceInfo,
           pmSubtaskInfo pSubtaskInfo);
215     typedef void
           (*pmSubtaskCallback_GPU_CUDA)(pmTaskInfo
           pTaskInfo, pmDeviceInfo* pDeviceInfo,
           pmSubtaskInfo pSubtaskInfo, pmStatus*
           pStatus); // pointer to CUDA kernel
216     typedef pmStatus
           (*pmSubtaskCallback_GPU_Custom)(pmTaskInfo
           pTaskInfo, pmDeviceInfo pDeviceInfo,
           pmSubtaskInfo pSubtaskInfo, void*
           pCudaStream);
217     typedef pmStatus
           (*pmDataReductionCallback)(pmTaskInfo
           pTaskInfo, pmDeviceInfo pDevice1Info,
           pmSubtaskInfo pSubtask1Info, pmDeviceInfo
           pDevice2Info, pmSubtaskInfo pSubtask2Info);
218     typedef pmStatus
           (*pmDataRedistributionCallback)(pmTaskInfo
           pTaskInfo, pmDeviceInfo pDeviceInfo,
```

```
        pmSubtaskInfo pSubtaskInfo);
    typedef bool
        (*pmDeviceSelectionCallback)(pmTaskInfo
        pTaskInfo, pmDeviceInfo pDeviceInfo);
    typedef pmStatus
        (*pmPreDataTransferCallback)(pmTaskInfo
        pTaskInfo, pmDataTransferInfo
        pDataTransferInfo);
    typedef pmStatus
        (*pmPostDataTransferCallback)(pmTaskInfo
        pTaskInfo, pmDataTransferInfo
        pDataTransferInfo);
    typedef pmStatus
        (*pmTaskCompletionCallback)(pmTaskInfo
        pTaskInfo);

    /** Unified callback structure */
    typedef struct pmCallbacks
    {
        pmDataDistributionCallback dataDistribution;
        pmSubtaskCallback_CPU subtask_cpu;
        pmSubtaskCallback_GPU_CUDA subtask_gpu_cuda;
        pmSubtaskCallback_GPU_Custom
            subtask_gpu_custom;     // Atleast one of
            subtask_gpu_cuda and subtask_gpu_custom
            must be NULL
        pmDataReductionCallback dataReduction;
        pmDataRedistributionCallback
            dataRedistribution;
        pmDeviceSelectionCallback deviceSelection;
        pmPreDataTransferCallback preDataTransfer;
        pmPostDataTransferCallback postDataTransfer;
        pmTaskCompletionCallback
            taskCompletionCallback;
        const char* subtask_opencl;

        pmCallbacks();
        pmCallbacks(pmDataDistributionCallback,
            const char* subtask_opencl);
        pmCallbacks(pmDataDistributionCallback,
            const char* subtask_opencl,
            pmDataReductionCallback);
```

```
240         pmCallbacks(pmDataDistributionCallback,
                const char* subtask_opencl,
                pmDataRedistributionCallback);
241         pmCallbacks(pmDataDistributionCallback,
                pmSubtaskCallback_CPU,
                pmSubtaskCallback_GPU_CUDA);
242         pmCallbacks(pmDataDistributionCallback,
                pmSubtaskCallback_CPU,
                pmSubtaskCallback_GPU_Custom);
243         pmCallbacks(pmDataDistributionCallback,
                pmSubtaskCallback_CPU,
                pmSubtaskCallback_GPU_CUDA,
                pmDataReductionCallback);
244         pmCallbacks(pmDataDistributionCallback,
                pmSubtaskCallback_CPU,
                pmSubtaskCallback_GPU_Custom,
                pmDataReductionCallback);
245         pmCallbacks(pmDataDistributionCallback,
                pmSubtaskCallback_CPU,
                pmSubtaskCallback_GPU_CUDA,
                pmDataRedistributionCallback);
246         pmCallbacks(pmDataDistributionCallback,
                pmSubtaskCallback_CPU,
                pmSubtaskCallback_GPU_Custom,
                pmDataRedistributionCallback);
247     } pmCallbacks;

248     /** The callback registeration API. The
            callbacks must be registered on all machines
            using the same key.
249      * The registered callbacks are returned in the
             pointer pCallbackHandle (if registeration is
             successful).
250      */
251     pmStatus pmRegisterCallbacks(const char* pKey,
            pmCallbacks pCallbacks, pmCallbackHandle*
            pCallbackHandle);

252     /* The registered callbacks must be released by
            the application using the following API */
253     pmStatus pmReleaseCallbacks(pmCallbackHandle
            pCallbackHandle);
```

```
254     /** The memory creation API. The allocated
           memory is returned in the variable pMemHandle
           */
255     pmStatus pmCreateMemory(size_t pLength,
           pmMemHandle* pMemHandle);

256     /** The 2D memory creation API. The allocated
           memory is returned in the variable pMemHandle.
257      *  The allocated memory has a two dimensional
            layout with pRows rows and pCols columns.
258      */
259     pmStatus pmCreateMemory2D(size_t pRows, size_t
           pCols, pmMemHandle* pMemHandle);

260     /* The memory destruction API. The same
           interface is used for both input and output
           memory */
261     pmStatus pmReleaseMemory(pmMemHandle pMemHandle);

262     /** This routine reads the entire distributed
           memory pointed to by pMem from the entire
           cluster into the local buffer. This is a
           blocking call.
263      */
264     pmStatus pmFetchMemory(pmMemHandle pMemHandle);

265     /** This routine fetches pLength bytes of
           distributed memory pointed to by pMem from
           offset pOffset into the local buffer. This is
           a blocking call.
266      */
267     pmStatus pmFetchMemoryRange(pmMemHandle
           pMemHandle, size_t pOffset, size_t pLength);

268     /** This routine returns the naked memory
           pointer associated with pMem handle.
269      *  This pointer may be used in memcpy and
            related functions.
270      */
271     pmStatus pmGetRawMemPtr(pmMemHandle pMemHandle,
           pmRawMemPtr* pPtr);
```

```
    /** The memory subscription APIs. These establish memory dependencies for a subtask.
     * Any subtask is not allowed to subscribe on behalf any other subtask.
     * These function can only be called from DataDistribution callback. The effect
     * of calling this function otherwise is undefined. The functions can be mixed in
     * any order and may be called multiple times for a particular subtask.
     */
    pmStatus pmSubscribeToMemory(pmTaskHandle pTaskHandle, pmDeviceHandle pDeviceHandle, ulong pSubtaskId, pmSplitInfo& pSplitInfo, uint pMemIndex, pmSubscriptionType pSubscriptionType, const pmSubscriptionInfo& pSubscriptionInfo);

    pmStatus pmSubscribeToMemory(pmTaskHandle pTaskHandle, pmDeviceHandle pDeviceHandle, ulong pSubtaskId, pmSplitInfo& pSplitInfo, uint pMemIndex, pmSubscriptionType pSubscriptionType, const pmScatteredSubscriptionInfo& pScatteredSubscriptionInfo);

    /** The memory redistribution API. It establishes memory ordering for an
     * address space computed by a subtask in the final task memory. Order 0 is assumed
     * to be the first order. Data for order 1 is placed after all data for order 0.
     * There is no guaranteed ordering inside an order number if multiple subtasks
     * produce data for that order. This function can only be called from DataRedistribution
     * callback. The effect of calling this function otherwise is undefined.
     */
    pmStatus pmRedistributeData(pmTaskHandle pTaskHandle, pmDeviceHandle pDeviceHandle,
```

```
        ulong pSubtaskId, pmSplitInfo& pSplitInfo,
        uint pMemIndex, size_t pOffset, size_t
        pLength, uint pOrder);

288     /** The CUDA launch configuration structure */
289     typedef struct pmCudaLaunchConf
290     {
291         int blocksX;
292         int blocksY;
293         int blocksZ;
294         int threadsX;
295         int threadsY;
296         int threadsZ;
297         int sharedMem;

298         pmCudaLaunchConf();
299         pmCudaLaunchConf(int, int, int, int, int,
                int);
300         pmCudaLaunchConf(int, int, int, int, int,
                int, int);
301     } pmCudaLaunchConf;

302     /** The CUDA launch configuration setting API.
        It sets kernel launch configuration for the
        subtask specified by
303      * pSubtaskId. The launch configuration is
         specified in the structure pCudaLaunchConf.
304      * This function can only be called from
         DataDistribution callback. The effect
305      * of calling this function otherwise is
         undefined.
306      */
307     pmStatus pmSetCudaLaunchConf(pmTaskHandle
        pTaskHandle, pmDeviceHandle pDeviceHandle,
        ulong pSubtaskId, pmSplitInfo& pSplitInfo,
        pmCudaLaunchConf& pCudaLaunchConf);

308     /** If subtask_gpu_custom is set, application
        may need to allocate a CUDA buffer in the
        custom callback.
309      * cudaMalloc and like functions are
         synchronous and they interrupt any
```

```
        possibility of asynchronous launches,
310     *  resulting in limited occupancy on the
           device. By using this function, a subtask
           can upfront ask the library
311     *  to reserve that buffer. This buffer can be
           accessed in kernels using
           pSubtaskInfo.gpuContext→reservedGlobalMem
312     *  This function can only be called from
           DataDistribution callback. The effect of
           calling this function otherwise
313     *  is undefined.
314     */
315    pmStatus pmReserveCudaGlobalMem(pmTaskHandle
          pTaskHandle, pmDeviceHandle pDeviceHandle,
          ulong pSubtaskId, pmSplitInfo& pSplitInfo,
          size_t pSize);

316    /** The structure that associates tasks to
          address spaces */
317    typedef struct pmTaskMem
318    {
319        pmMemHandle memHandle;
320        pmMemType memType;
321        pmSubscriptionVisibilityType
              subscriptionVisibilityType;     /* By
              default, this is SUBSCRIPTION_NATURAL */
322        bool disjointReadWritesAcrossSubtasks;  /*
              By default, this is false. Applies only
              to RW adress spaces. */

323        pmTaskMem();
324        pmTaskMem(pmMemHandle, pmMemType);
325        pmTaskMem(pmMemHandle, pmMemType,
              pmSubscriptionVisibilityType);
326        pmTaskMem(pmMemHandle, pmMemType,
              pmSubscriptionVisibilityType, bool);
327    } pmTaskMem;

328    /** The task details structure used for task
          submission */
329    typedef struct pmTaskDetails
330    {
```

```
331         void* taskConf;
332         uint taskConfLength;
333         pmTaskMem* taskMem;
334         uint taskMemCount;
335         pmCallbackHandle callbackHandle;
336         ulong subtaskCount;
337         ulong taskId;    /* Meant for application to
               assign and identify tasks */
338         ushort priority;     /* By default, this is
               set to max priority level (0) */
339         pmSchedulingPolicy policy;  /* By default,
               this is SLOW_START */
340         int timeOutInSecs;  /* By default, this is
               max possible value in signed int,
               negative values mean no timeout */
341         bool multiAssignEnabled;     /* By default,
               this is true */
342         bool overlapComputeCommunication;    /* By
               default, this is true */
343         bool canSplitCpuSubtasks;    /* By default,
               this is false */
344         bool canSplitGpuSubtasks;    /* By default,
               this is false */
345     #ifdef SUPPORT_CUDA
346         bool cudaCacheEnabled;  /* By default, this
               is true */
347     #endif
348         bool suppressTaskLogs;  /* By default, this
               is false */
349         pmAffinityCriterion affinityCriterion;  /*
               By default, this is MAXIMIZE_LOCAL_DATA */
350         pmClusterHandle cluster;     /* Unused */

351         pmTaskDetails();
352         pmTaskDetails(void* taskConf, uint
               taskConfLength, pmTaskMem*, uint
               taskMemCount, pmCallbackHandle
               callbackHandle, ulong subtaskCount);
353     } pmTaskDetails;

354     /* The flag disjointReadWritesAcrossSubtasks
           should be true (for RW output memories) if
```

```
        the read subscriptions of a subtask do not
        overlap with write subscriptions of any
        subtasks other than itself. */

    /** The task submission API. Returns the task
        handle in variable pTaskHandle on success. */
    pmStatus pmSubmitTask(pmTaskDetails
        pTaskDetails, pmTaskHandle* pTaskHandle);

    /** The submitted tasks must be released by the
        application using the following API.
     *  The API automatically blocks till task
        completion. Returns the task's exit status.
     */
    pmStatus pmReleaseTask(pmTaskHandle pTaskHandle);

    /** A task is by default non blocking. The
        control comes back immediately.
     *  Use the following API to wait for the task
        to finish.
     *  The API returns the exit status of the task.
     */
    pmStatus pmWaitForTaskCompletion(pmTaskHandle
        pTaskHandle);

    /** Returns the task execution time (in seconds)
        in the variable pTime.
     *  The API automatically blocks till task
        completion.
     */
    pmStatus
        pmGetTaskExecutionTimeInSecs(pmTaskHandle
        pTaskHandle, double* pTime);

    /** A unified API to release task and it's
        associated resources (callbacks, input and
        output memory).
     *  Returns the task's exit status (if no
        error). The API automatically blocks till
        task finishes.
     */
    pmStatus pmReleaseTaskAndResources(pmTaskDetails
```

```
        pTaskDetails, pmTaskHandle pTaskHandle);

374     /** This function returns a writable buffer
           accessible to data distribution, subtask,
           data reduction and data redistribution
           callbacks.
375      Scratch buffer size parameter is only honored
            for the first invocation of this function
            for a particular subtask and scratch buffer
            type.
376      Successive invocations return the buffer
            allocated at initial request size. This
            buffer is only used to pass information
            generated in one callback
377      to other callbacks */
378     #ifdef __CUDACC__
379     __host__ __device__
380     #endif
381     inline void* pmGetScratchBuffer(pmTaskHandle
           pTaskHandle, pmDeviceHandle pDeviceHandle,
           ulong pSubtaskId, pmSplitInfo& pSplitInfo,
           pmScratchBufferType pScratchBufferType,
           size_t pBufferSize, pmGpuContext* pGpuContext)
382     {
383     #if defined(__CUDA_ARCH__)
384         return (pGpuContext ?
               pGpuContext->scratchBuffer : NULL);
385     #else
386         void*
               pmGetScratchBufferHostFunc(pmTaskHandle
               pTaskHandle, pmDeviceHandle
               pDeviceHandle, ulong pSubtaskId,
               pmSplitInfo& pSplitInfo,
               pmScratchBufferType pScratchBufferType,
               size_t pBufferSize);
387         return
               pmGetScratchBufferHostFunc(pTaskHandle,
               pDeviceHandle, pSubtaskId, pSplitInfo,
               pScratchBufferType, pBufferSize);
388     #endif
389     }
```

```
390     /** This function is only supported from CPU and
            can not be called from GPU kernels */
391     pmStatus pmReleaseScratchBuffer(pmTaskHandle
            pTaskHandle, pmDeviceHandle pDeviceHandle,
            ulong pSubtaskId, pmSplitInfo& pSplitInfo,
            pmScratchBufferType pScratchBufferType);

392     /** This function returns the
            REDUCTION_TO_REDUCTION scratch buffer
            associated with the final reduced subtask of
            the task pTaskHandle */
393     void*
            pmGetLastReductionScratchBuffer(pmTaskHandle
            pTaskHandle);

394     /** This function returns the redistribution
            metadata i.e. a set of tuples representing
            order number and redistributions received for
            that order.
395      *  The pCount output parameter is the total
             count of redistributions. */
396     pmRedistributionMetadata*
            pmGetRedistributionMetadata(pmTaskHandle
            pTaskHandle, uint pMemIndex, ulong* pCount);

397 } // end namespace pm
```

Listing 7.1: Unicorn Header: pmPublicDefinitions.h

```
1  namespace pm
2  {
3      typedef enum pmReductionOpType
4      {
5          REDUCE_ADD,
6          REDUCE_MIN,
7          REDUCE_MAX,
8          REDUCE_PRODUCT,
9          REDUCE_LOGICAL_AND,
10         REDUCE_BITWISE_AND,
11         REDUCE_LOGICAL_OR,
12         REDUCE_BITWISE_OR,
```

```
13          REDUCE_LOGICAL_XOR,
14          REDUCE_BITWISE_XOR,
15          MAX_REDUCTION_OP_TYPES
16      } pmReductionOpType;

17      typedef enum pmReductionDataType
18      {
19          REDUCE_INTS,
20          REDUCE_UNSIGNED_INTS,
21          REDUCE_LONGS,
22          REDUCE_UNSIGNED_LONGS,
23          REDUCE_FLOATS,
24          REDUCE_DOUBLES,
25          MAX_REDUCTION_DATA_TYPES
26      } pmReductionDataType;

27      /** The following function can be used for
           optimal inbuilt reduction of two subtasks. */
28      pmDataReductionCallback
           pmGetSubtaskReductionCallbackImpl
29          (pmReductionOpType pOperation,
               pmReductionDataType pDataType);

30      /**  The following function can be called from
           within a custom implementation of
           pmDataReductionCallback. */
31      pmStatus pmReduceSubtasks(pmTaskHandle
           pTaskHandle, pmDeviceHandle pDevice1Handle,
           unsigned long pSubtask1Id, pmSplitInfo&
           pSplitInfo1, pmDeviceHandle pDevice2Handle,
           unsigned long pSubtask2Id, pmSplitInfo&
           pSplitInfo2, pmReductionOpType pOperation,
           pmReductionDataType pDataType);

32      const size_t MAX_FILE_SIZE_LEN = 2048;

33      /** This function returns the starting address
           of the file specified by pPath and memory
           mapped by the call pmMapFile.
34       The number of bytes in pPath must be less than
            MAX_FILE_SIZE_LEN. */
35      void* pmGetMappedFile(const char* pPath);
```

```
36      /** This function memory maps an entire file
           specified by pPath on all machines in the
           cluster.
37       The number of bytes in pPath must be less than
            MAX_FILE_SIZE_LEN. */
38      pmStatus pmMapFile(const char* pPath);

39      /** This function unmaps the file specified by
           pPath and mapped by the call pmMapFile(s)
           from all machines in the cluster.
40       The number of bytes in pPath must be less than
            MAX_FILE_SIZE_LEN. */
41      pmStatus pmUnmapFile(const char* pPath);

42      /** This function memory maps pFileCount files
           specified by pPaths on all machines in the
           cluster.
43       The number of bytes in each pPaths entry must
            be less than MAX_FILE_SIZE_LEN. */
44      pmStatus pmMapFiles(const char* const* pPaths,
           uint pFileCount);

45      /** This function unmaps pFileCount files
           specified by pPaths and mapped by the call
           pmMapFile(s) from all machines in the cluster.
46       The number of bytes in each pPaths entry must
            be less than MAX_FILE_SIZE_LEN. */
47      pmStatus pmUnmapFiles(const char* const* pPaths,
           uint pFileCount);

48  }   // end namespace pm
```

Listing 7.2: Unicorn Header: pmPublicUtilities.h

Chapter 8

Related Work

Many single node CPU-GPU programming frameworks like [61, 43, 15] have been proposed in the literature. But these lack scheduling and load balancing capabilities and division of work between CPU and GPU is largely left to the programmer. StarPU [6] is a broader single node framework that supports CPU/GPU co-scheduling, but for optimal results it often requires calibration runs and scheduling hints from the programmer. It employs fine-grained schedulable units called codelets. Manual sizing of codelets in this framework is a challenge as the same size generally does not suit CPUs and GPUs well. GPUSs [7], a derivative of StarSs, also requires programmers to annotate specific code blocks with constructs (that identify tasks and target devices) and directionality clauses (that determine data movement). These hints are used to build a task dependency graph, which determines the scheduling of individual tasks. XKaapi [34] is another system that employs a work stealing scheduler to distribute the load on CPUs and GPUs. In all these systems, scheduling decisions once made are not re-assessed and the granularity of work division between various accelerators and CPU cores is left to the programmer. These systems also do not target multiple machines.

Existing cluster programming systems can be broadly classified into two categories. The first comprises language based approaches like [26, 42, 14, 50, 47, 38, 27], usually extending a sequential language like C or Fortran. The

second category includes library based approaches like [10, 29, 2, 60, 39]. The former ones focus variously on functional, loop or data parallelism and generally use shared address spaces (built on top of DSM or more specifically PGAS [54]) with fine grained synchronization. Their focus is to mainly allow the user to express parallelism at a high level and most do not support GPUs. In contrast, library based approaches employ some MPI-like communication, where machine specific details are not completely abstracted from the programmer. Instead of focusing on program logic, the programmer has to directly handle issues like synchronization, scalability and latency. Hence, usual problems like race conditions and deadlocks remain.

Other frameworks allow applications to use GPUs on remote nodes but they don't exhibit performance in the absence of optimizations like we report. Using remote GPU on a 40Gbps network, rCUDA [53, 25] reports remote matrix multiplication and 1D FFT respectively 12.9% and 72.9% slower than local GPU. Similarly, [8] and [35] report remote OpenCL extensions, but in spite of the application exercising direct control over the distribution of data and computation, their scalability and efficiency is not comparable to ours.

Our system *Unicorn* is built on top of pthreads, MPI and CUDA. Its novelty is in its general and intuitive interface, yet efficient implementation. It unifies computation on local and remote computing units (CPUs and GPUs) using bulk synchrony. Its runtime environment autonomously performs data distribution, dynamic load-balanced scheduling and synchronization. The closest existing works targeting CPU and GPU clusters are Phalanx [33], StarPU-MPI [5], G-Charm [59] and Legion [9]. Phalanx is a C++ template library with powerful mechanisms to create a thread hierarchy that is mapped onto

a hierarchy of processors and memories. But it lacks scheduling and load balancing capabilities. It also does not use CPUs and GPUs collectively for computations.

StarPU-MPI is an extension of StarPU but it does not fully abstract the existence of multiple machines: the programmer must either explicitly manage communication with an MPI-like interface or explicitly submit independent tasks to each node of the cluster. Rather than a unified cluster programming framework, it is an MPI based aggregation of independent StarPU instances running on each node. It lacks a cluster wide scheduler. It only provides independent schedulers on each node, with inter-node schedule managed by the user. StarPU maintains data replicas for potential use in upcoming tasks. However, if a task modifies data in one of the replicas, all others are invalidated. For optimal performance, it recommends applications to advise when and where not to keep data replicas. In contrast to this, *Unicorn* adopts a light weight memory consistency protocol where the invalidation messages are deferred to task boundaries (where they are piggy-backed on other regular message exchange between nodes) and no overhead is incurred during task execution. StarPU also supports a notion of data filters which allows data to be viewed in parts (or hierarchy) by associated codelets. These filters are usually synchronous. Asynchronous filters can also be created with some limitations on data usage by the application. *Unicorn*, on the other hand, is an entirely asynchronous system and achieves data partitioning through an application callback.

G-Charm, based on Charm++ [41], is a framework specially optimized for GPUs. It particularly focuses on reducing GPU data transfers by employ-

ing a software-cache over GPUs and grouping multiple Charm++ *chares* together to reduce the number of GPU kernel invocations. Each processor in the system runs an independent instance of Charm++ runtime and they communicate via message-passing (messages are buffered in a message queue with Charm++ runtime). Even the input data for chares is received by this mechanism. In contrast, *Unicorn* has a dedicated thread (on each node) to efficiently manage data transfers and segregate them from control messages. This approach guarantees progress of the entire asynchronous system. Also, unlike *Unicorn's multi-assign*, G-Charm lacks a mechanism to reconsider and correct poorly made scheduling decisions.

Legion is a powerful system that uses a software out-of-order processor to schedule application-created tasks with dependencies specified by the programmer. The programmer does so by explicitly partitioning data into memory regions and sub-regions and annotating these regions with access privileges (read-only, read-write, etc.) and coherence (exclusive access, atomic access, etc.). The system is complicated to program. For efficiency, the programmer is also required to write custom mappers which map regions and tasks to nodes.

StarPU-MPI, G-Charm and Legion are quite similar in the way they support generic task graphs where a task is individually schedulable and is capable of spawning more tasks. However, we argue that this is not the natural way the programmers think and write programs. In contrast, *Unicorn* abstracts generic task graphs into an interface where programmers write applications as a graph of tasks, with each task decomposable into several concurrently executable subtasks.

In contrast to StarPU-MPI and a few others, Unicorn applications need not write any code for task graphs, data transfers, MPI initialization, etc. But Unicorn applications do need to provide additional subtask subscription code. In StarPU-MPI, there is no concept of subscriptions but data filters do exist. In addition to this, all programming models provide some optional performance boosting features at the cost of some extra code (e.g. scheduling customization in StarPU and subtask resizing in Unicorn). Thus, it is difficult to provide an eternal comparison of the number of lines of code required by client applications of these models. However, in general, Unicorn requires significantly less new code than corresponding StarPU-MPI and other implementations, which leave many runtime controls to the application.

Chapter 9

Conclusions and Future Work

This thesis presents a practical bulk synchronous programming model that allows distribution of computation across multiple CPU cores within a node, multiple GPUs, and multiple nodes connected over a network, all in a unified manner. Our model maps efficiently to modern devices like GPUs, as they are already bulk synchronous in nature. Our runtime undertakes all local and networked data transfers, scheduling, and synchronization in an efficient and robust manner. By design, we eliminate races and deadlocks as all devices operate in a private view of the address space.

For ease of programming, *Unicorn* exposes a distributed shared memory system with transactional semantics. Experiments show that our runtime overcomes the performance limitations of the distributed shared memory approach and achieves good performance gains for coarse-grained experiments. This performance is possible due to a number of critical optimizations working in concert. These include prefetching, pipelining, maximizing overlap between computation and communication, and scheduling/re-scheduling efficiently across heterogeneous devices of vastly different capacities. Unicorn also employs special optimizations for GPUs like a software LRU cache to reduce DMA transfers and a proactive work-stealer to reduce pipeline stalls.

Our framework can realize any task that may be optionally decomposed into a set of concurrently executable subtasks with checkout/checkin memory se-

mantics and a synchronized reduction step to resolve conflicting checkins. However, tasks having non-deterministic access pattern (like graph traversal) or fine-grained/frequent communication or complex conflict resolution may not perform efficiently in our system. This thesis has not focused on irregular workloads in general. However, dynamic load balancing mechanisms have been shown to work well for irregular applications as well. Detailed experimentation with irregular applications remains future work.

In the future, there is potential to optimize data transfers for a set of tasks rather than one task at a time. Except for data dependency or an explicitly specified user dependency, *Unicorn* tasks are independent of each other. However, wiser scheduling decisions can be made with à priori knowledge of tasks to come. *Unicorn's* locality aware scheduler produces a schedule optimized for the task at hand. By evaluating the data requirements of dependent tasks, it is possible to produce a globally optimal schedule. Of course, the time it takes to generate such a schedule must be weighed against the time saved in data transfers while executing the global schedule.

In the present implementation, *Unicorn* moves data between GPUs on a node or across nodes through address spaces stored on CPU. However, it is possible to exploit GPUDirect technology to directly transfer data between GPUs on a node. GPUDirect RDMA can be explored for inter-node GPU-GPU data transfers. MPI also offers RDMA features on high performance interconnects (like Infiniband) allowing applications to directly access remote node's memory bypassing operating systems on both nodes involved in communication.

Bibliography

[1] InfiniBand Trade Association, InfiniBand Architecture Specification, Release 1.1, 2002.

[2] A.M. Aji, J. Dinan, D. Buntinas, P. Balaji, Wu chun Feng, K.R. Bisset, and R. Thakur. MPI-ACC: An integrated and extensible approach to data movement in accelerator-based systems. In *HPCC'12*, pages 647–654, 2012.

[3] Cristiana Amza, Alan L. Cox, Sandhya Dwarkadas, Pete Keleher, Honghui Lu, Ramakrishnan Rajamony, Weimin Yu, and Willy Zwaenepoel. Treadmarks: Shared memory comput. on networks of workstations. *Computer*, 29(2):18–28, February 1996.

[4] James Aspnes, Yossi Azar, Amos Fiat, Serge Plotkin, and Orli Waarts. On-line routing of virtual circuits with applications to load balancing and machine scheduling. *J. ACM*, 44(3):486–504, May 1997.

[5] Cédric Augonnet, Olivier Aumage, Nathalie Furmento, Raymond Namyst, and Samuel Thibault. StarPU-MPI: Task programming over clusters of machines enhanced with accelerators. In *Euro. Conf. Recent Advances in the MPI*, EuroMPI, pages 298–299. Springer-Verlag, 2012.

[6] Cédric Augonnet, Samuel Thibault, Raymond Namyst, and Pierre-André; Wacrenier. StarPU: A unified platform for task scheduling on heterogeneous multicore architectures. *Concurr. Comput. : Pract. Exper.*, 23(2):187–198, February 2011.

[7] Eduard Ayguadé, Rosa M. Badia, Francisco D. Igual, Jesús Labarta, Rafael Mayo, and Enrique S. Quintana-Ortí. An extension of the StarSs programming model for platforms with multiple GPUs. In *Euro-Par '09*, Euro-Par, pages 851–862. Springer-Verlag, 2009.

[8] A. Barak, T. Ben-Nun, E. Levy, and A. Shiloh. A package for OpenCL based heterogeneous computing on clusters with many gpu devices. In *Cluster Computing Workshops and Posters*, pages 1–7, 2010.

[9] Michael Bauer, Sean Treichler, Elliott Slaughter, and Alex Aiken. Legion: Expressing locality and independence with logical regions. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC '12, pages 66:1–66:11, Los Alamitos, CA, USA, 2012. IEEE Computer Society Press.

[10] Adam Beguelin, Jack Dongarra, Al Geist, Robert Manchek, and Vaidy Sunderam. A user's guide to PVM parallel virtual machine. Technical report, University of Tennessee, 1991.

[11] T. Beri, S. Bansal, and S. Kumar. A scheduling and runtime framework for a cluster of heterogeneous machines with multiple accelerators. In *Parallel and Distributed Processing Symposium (IPDPS), 2015 IEEE International*, pages 146–155, May 2015.

[12] Tarun Beri, Sorav Bansal, and Subodh Kumar. Locality Aware Work-Stealing based Scheduling in Hybrid CPU-GPU Clusters. In *Parallel and Distributed Processing Techniques and Applications*, 2015.

[13] Tarun Beri, Sorav Bansal, and Subodh Kumar. Prosteal: A proactive work stealer for bulk synchronous tasks distributed on a cluster of

heterogeneous machines with multiple accelerators. In *2015 IEEE International Parallel and Distributed Processing Symposium Workshop, IPDPS 2015, Hyderabad, India, May 25-29, 2015*, pages 17–26, 2015.

[14] Robert D. Blumofe, Christopher F. Joerg, Bradley C. Kuszmaul, Charles E. Leiserson, Keith H. Randall, and Yuli Zhou. Cilk: An efficient multithreaded runtime system. *SIGPLAN Not.*, 30(8):207–216, August 1995.

[15] C++ AMP: Language and programming model. http://msdn.microsoft.com/en-us/library/hh265136.aspx, 2012.

[16] Colin Campbell and Ade Miller. *A Parallel Programming with Microsoft Visual C++: Design Patterns for Decomposition and Coordination on Multicore Architectures*. Microsoft, 1st edition, 2011.

[17] The NVIDIA CUDA basic linear algebra subroutines. https://developer.nvidia.com/cuBLAS.

[18] CUFFT: CUDA fast fourier transform (FFT). http://docs.nvidia.com/cuda/cufft/.

[19] Leonardo Dagum and Ramesh Menon. OpenMP: An industry-standard API for shared-memory programming. *IEEE Comput. Sci. Eng.*, 5(1):46–55, January 1998.

[20] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: Simplified data processing on large clusters. *Commun. ACM*, 51(1):107–113, January 2008.

[21] James W. Demmel, Nicholas J. Higham, and Robert S. Schreiber. Block LU factorization. http://www.netlib.org/utk/papers/factor/node7.html, 1995.

[22] P. Deutsch and J.-L. Gailly. Zlib compressed data format specification version 3.3, 1996.

[23] James Dinan, D. Brian Larkins, P. Sadayappan, Sriram Krishnamoorthy, and Jarek Nieplocha. Scalable work stealing. In *SC*, pages 53:1–53:11. ACM, 2009.

[24] Jack J. Dongarra, Jeremy Du Croz, Sven Hammarling, and Richard J. Hanson. An extended set of FORTRAN basic linear algebra subprograms. *ACM Trans. Math. Softw.*, 14(1):1–17, March 1988.

[25] J. Duato, A.J. Pena, F. Silla, R. Mayo, and E.S. Quintana-Ort. Performance of cuda virtualized remote gpus in high performance clusters. In *Parallel Processing (ICPP), 2011 International Conference on*, pages 365–374, 2011.

[26] Tarek El-Ghazawi and Lauren Smith. UPC: Unified parallel C. In *ACM/IEEE Conf. Supercomputing*, SC. ACM, 2006.

[27] Kayvon Fatahalian, Daniel Reiter Horn, Timothy J. Knight, Larkhoon Leem, Mike Houston, Ji Young Park, Mattan Erez, Manman Ren, Alex Aiken, William J. Dally, and Pat Hanrahan. Sequoia: Programming the memory hierarchy. In *ACM/IEEE Conf. Supercomputing*, SC. ACM, 2006.

[28] M. Fogue, F.D. Igual, E.S. Quintana-Orti, and R.A. van de Geijn. Retargeting plapack to clusters with hardware accelerators. In *HPCS*, pages 444–451, June 2010.

[29] Ian Foster. Globus toolkit version 4: Software for service-oriented systems. In *IFIP Intl. Conf. Network and Parallel Computing*, NPC, pages 2–13. Springer-Verlag, 2005.

[30] M. Frigo and S.G. Johnson. Fftw: an adaptive software architecture for the FFT. In *ICASSP*, volume 3, pages 1381–1384, 1998.

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

[32] Bill O. Gallmeister. *POSIX.4: Programming for the Real World*. O'Reilly & Associates, Inc., Sebastopol, CA, USA, 1995.

[33] Michael Garland, Manjunath Kudlur, and Yili Zheng. Designing a unified programming model for heterogeneous machines. In *Intl. Conf. on High Perf. Computing, Networking, Storage and Analysis*, SC, pages 67:1–67:11, Los Alamitos, CA, USA, 2012. IEEE Computer Society Press.

[34] Thierry Gautier, Joao V. F. Lima, Nicolas Maillard, and Bruno Raffin. Xkaapi: A runtime system for data-flow task programming on hetero-

geneous architectures. In *IPDPS '13*, IPDPS '13, pages 1299–1308, Washington, DC, USA, 2013. IEEE Computer Society.

[35] Ivan Grasso, Simone Pellegrini, Biagio Cosenza, and Thomas Fahringer. Libwater: Heterogeneous distributed computing made easy. In *International Conference on Supercomputing*, ICS '13, pages 161–172, New York, NY, USA, 2013. ACM.

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

[37] Antonin Guttman. R-trees: A dynamic index structure for spatial searching. In *Proceedings of the 1984 ACM SIGMOD International Conference on Management of Data*, SIGMOD '84, pages 47–57, New York, NY, USA, 1984. ACM.

[38] Paul N. Hilfinger, Dan Bonachea, David Gay, Susan Graham, Ben Liblit, Geoff Pike, and Katherine Yelick. Titanium language reference manual. Technical report, University of California at Berkeley, 2001.

[39] Jonathan M. D. Hill, Bill McColl, Dan C. Stefanescu, Mark W. Goudreau, Kevin Lang, Satish B. Rao, Torsten Suel, Thanasis Tsantilas, and Rob H. Bisseling. BSPlib: The BSP programming library. *Parallel Comput.*, 24(14):1947–1980, 1998.

[40] Seung jai Min, Costin Iancu, and Katherine Yelick. Hierarchical work stealing on manycore clusters. In *Fifth Conference on Partitioned Global Address Space Programming Models*, 2011.

[41] Laxmikant V. Kale and Sanjeev Krishnan. Charm++: A portable concurrent object oriented system based on c++. Technical report, Champaign, IL, USA, 1993.

[42] A. Krishnamurthy, D. E. Culler, A. Dusseau, S. C. Goldstein, S. Lumetta, T. von Eicken, and K. Yelick. Parallel programming in Split-C. In *ACM/IEEE Conf. Supercomputing*, SC, pages 262–273. ACM, 1993.

[43] Seyong Lee and Rudolf Eigenmann. OpenMPC: Extended OpenMP programming and tuning for GPUs. In *SC*, pages 1–11. IEEE Comput. Society, 2010.

[44] Jonathan Lifflander, Sriram Krishnamoorthy, and Laxmikant V. Kale. Work stealing and persistence-based load balancers for iterative overdecomposed applications. In *Intl. Symp. High-Perf. Parll. and Dist. Comput.*, HPDC, pages 137–148. ACM, 2012.

[45] Seung Jai Min, Ayon Basumallik, and Rudolf Eigenmann. Supporting realistic OpenMP applications on a commodity cluster of workstations. In *Intl. Conf. on OpenMP Shared Memory Parallel Programming*, WOMPAT, pages 170–179, Berlin, Heidelberg, 2003. Springer-Verlag.

[46] Eric Mohr, David A. Kranz, and Robert H. Halstead, Jr. Lazy task creation: A technique for increasing the granularity of parallel programs. In *Proceedings of the 1990 ACM Conference on LISP and Functional Programming*, LFP '90, pages 185–197, New York, NY, USA, 1990. ACM.

[47] Tuan-Anh Nguyen and Pierre Kuonen. Programming the grid with POP-C++. *Future Gener. Comput. Syst.*, 23(1):23–30, January 2007.

[48] John Nickolls, Ian Buck, Michael Garland, and Kevin Skadron. Scalable parallel programming with CUDA. *Queue*, 6(2):40–53, March 2008.

[49] Jarek Nieplocha, Bruce Palmer, Vinod Tipparaju, Manojkumar Krishnan, Harold Trease, and Edoardo Aprà. Advances, applications and perf. of the global arrays shared memory programming toolkit. *Int. J. High Perform. Comput. Appl.*, 20(2):203–231, May 2006.

[50] Robert W. Numrich and John Reid. Co-array Fortran for parallel programming. *SIGPLAN Fortran Forum*, 17(2):1–31, August 1998.

[51] Lawrence Page, Sergey Brin, Rajeev Motwani, and Terry Winograd. The pagerank citation ranking: Bringing order to the web. Technical Report 1999-66, Stanford InfoLab, November 1999. Previous number = SIDL-WP-1999-0120.

[52] Krzysztof Parzyszek. *Generalized Portable Shmem Library for High Perf. Computing*. PhD thesis, 2003.

[53] Antonio J Peña, Carlos Reaño, Federico Silla, Rafael Mayo, Enrique S Quintana-Ortí, and José Duato. A complete and efficient cuda-sharing solution for hpc clusters. *Parallel Computing*, 2014.

[54] Tim Stitt. An introduction to the partitioned global address space (pgas) programming model. In *Connexions*. Mar 2010.

[55] John E. Stone, David Gohara, and Guochun Shi. OpenCL: A parallel prog. standard for heterogeneous comput. sys. *IEEE Des. Test*, 12(3):66–73, 2010.

[56] Intel Threading Building Blocks. http://www.threadingbuildingblocks.org/.

[57] Leslie G. Valiant. A bridging model for parallel computation. *Commun. ACM*, 33(8):103–111, 1990.

[58] Robert A. van de Geijn and Jerrell Watts. Summa: Scalable universal matrix multiplication algorithm. Technical report, Austin, TX, USA, 1995.

[59] R. Vasudevan, Sathish S. Vadhiyar, and Laxmikant V. Kalé. G-charm: An adaptive runtime system for message-driven parallel applications on hybrid systems. In *Proceedings of the 27th International ACM Conference on International Conference on Supercomputing*, ICS '13, pages 349–358, New York, NY, USA, 2013. ACM.

[60] Hao Wang, Sreeram Potluri, Miao Luo, Ashish Kumar Singh, Sayantan Sur, and Dhabaleswar K. Panda. MVAPICH2-GPU: Optimized GPU to GPU commun. for infiniband clusters. *Comput. Sci.*, 26(3-4):257–266, June 2011.

[61] Sandra Wienke, Paul Springer, Christian Terboven, and Dieter an Mey. OpenACC: First experiences with real-world applications. In *ICPP*, Euro-Par, pages 859–870. Springer-Verlag, 2012.

Appendix

10.1 Unicorn's MapReduce Extension

The MapReduce [20] programming model visualizes a task in two stages. The first stage (called *map stage*) marshalls the input data into different groups while the second stage (called *reduce stage*) consumes these groups and produces a reduction (or summarization) of each. Both stages are individually parallelizable and can be run distributedly over a cluster. The reduce stage, however, must start after the map stage is complete.

In this thesis, we explore *Unicorn's* suitability to realize this high level programming abstraction. For the map stage, we deploy *Unicorn's subtask execution* callback while the reduce stage is executed by *data reduction* callback. Note that these two callbacks are already executed sequentially by *Unicorn's* runtime. In the former callback, subtasks concurrently process disjoint data (subscribed by of *data subscription* callback) from input address space(s) and produce a logical grouping in output address space(s). For example, in PageRank experiment (section 5), the callback results in each subtask producing an array whose indices represent web page IDs and values are real numbers that represent page rank contributions from the data (or web pages) processed by this subtask. For the reduce stage, our runtime takes two subtasks at a time and logically sums them up (i.e., adds the page rank values at every index of the output of both subtasks). Note that Unicorn provides built-in functions for mostly used reductions operations like summation.

In such experiments, however, the data produced by map stage is sparse

because a subtask contributes page ranks to only a small fraction of the web. Thus, most indices in a subtask's output array (in the address space) are zero. In a cluster environment like *Unicorn*, data reduction requires movement of a lot of data. This includes both inter-node data transfers and GPU to CPU transfers. Our runtime provides a few simple compression routines like *Run Length Encoding* (RLE) where a sequence of frequently occurring values (like zero) in a sparse array are replaced by their run lengths. Results in section 5 provide more insight into the performance benefits of this scheme.

We also experimented with Zlib [22] but found that the compression benefits are nullified by the time taken to compress (and uncompress). Rather, simpler techniques like RLE show promising results for latency bound applications.

10.2 Scratch Buffers

Unicorn runtime supports inter callback communication for subtasks. The *pmGetScratchBuffer* API call documented in chapter 7 creates a memory buffer, called *scratch buffer*, for this purpose. For every subtask, the first call to the said API creates the scratch buffer of the requested size while every subsequent call returns the buffer that was created by the first call. For GPU subtasks, the *subtask execution* callback runs on GPU while other callbacks are executed on CPU. In this case, the runtime transparently moves scratch buffers from CPU to GPU or vice versa, as required. A *scratch buffer* can be optionally deleted by its owning subtask using *pmReleaseScratchBuffer* API call if it is no longer needed or in case a scratch buffer of different size is

required.

10.3 Matrix Multiplication Source Code

This section lists the source code of an application that parallelizes square matrix multiplication using *Unicorn*. The CPU subtask code is implemented in C using BLAS [24] routines while the GPU subtask code is written in CUDA using CUBLAS [17] routines. Listing 10.1 contains the header file for the application. Listing 10.2 specifies the C/C++ code for the application while listing 10.3 contains the CUDA code. The code logically divides the output matrix into blocks of size $2048 * 2048$ and each of these blocks is computed by a different subtask. To compute its block, a subtask subscribes to all blocks in the corresponding row of the first input matrix and to all blocks in the corresponding column of the second input matrix.

```
1 #define ELEM_SIZE sizeof(float)
2 #define BLOCK_DIM 2048

3 #define BLOCK_OFFSET_IN_ELEMS(blockRow, blockCol,
      blockDim, matrixDim) (((blockRow) * (matrixDim) +
      (blockCol)) * (blockDim))

4 #define SUBSCRIBE_BLOCK(blockRow, blockCol,
      blockOffset, blockHeight, blockDim, matrixDim,
      subtaskId, splitInfo, memoryIndex,
      subscriptionType) \
5 { \
6     size_t bOffset = BLOCK_OFFSET_IN_ELEMS(blockRow,
          blockCol, blockDim, matrixDim); \
7     pmScatteredSubscriptionInfo sInfo(((blockOffset
          * matrixDim) + bOffset) * ELEM_SIZE, blockDim
```

```
        * ELEM_SIZE, matrixDim * ELEM_SIZE,
        (blockHeight)); \
8     pmSubscribeToMemory(pTaskInfo.taskHandle,
        pDeviceInfo.deviceHandle, subtaskId,
        splitInfo, memoryIndex, subscriptionType,
        sInfo); \
9 }

10 enum memIndex
11 {
12     INPUT_MATRIX1_MEM_INDEX = 0,
13     INPUT_MATRIX2_MEM_INDEX,
14     OUTPUT_MATRIX_MEM_INDEX,
15     MAX_MEM_INDICES
16 };

17 struct matMulTaskConf
18 {
19     size_t matrixDim;
20     size_t blockDim;
21 };

22 pmStatus matrixMultiply_cudaLaunchFunc(pmTaskInfo
    pTaskInfo, pmDeviceInfo pDeviceInfo,
    pmSubtaskInfo pSubtaskInfo, void* pCudaStream);
```

Listing 10.1: File matmul.h

```
1 void serialMatrixMultiply(float* pMatrixA, float*
    pMatrixB, float* pMatrixC, size_t pDim1, size_t
    pDim2, size_t pDim3, size_t pRowStepElems1,
    size_t pRowStepElems2, size_t pRowStepElems3)
2 {
3     cblas_sgemm(CblasRowMajor, CblasNoTrans,
        CblasNoTrans, (int)pDim1, (int)pDim3,
        (int)pDim2, 1.0f, pMatrixA,
        (int)pRowStepElems1, pMatrixB,
        (int)pRowStepElems2, 0.0f, pMatrixC,
        (int)pRowStepElems3);
4 }
```

```
5  bool GetSplitData(size_t* pBlockOffset, size_t*
       pBlockHeight, matrixMultiplyTaskConf* pTaskConf,
       pmSplitInfo& pSplitInfo)
6  {
7      *pBlockOffset = 0;
8      *pBlockHeight = pTaskConf->blockDim;
9      if(pSplitInfo.splitCount)
10     {
11         size_t lSplitCount = ((pTaskConf->blockDim <
               pSplitInfo.splitCount) ?
               pTaskConf->blockDim :
               pSplitInfo.splitCount);

12         if(pSplitInfo.splitId > lSplitCount    1)
13             return false;

14         size_t lSplittedBlockSize =
               (pTaskConf->blockDim / lSplitCount);
15         *pBlockOffset = pSplitInfo.splitId *
               lSplittedBlockSize;

16         if(pSplitInfo.splitId == lSplitCount    1)
17             *pBlockHeight = (pTaskConf->blockDim
                   (pSplitInfo.splitId *
                   lSplittedBlockSize));
18         else
19             *pBlockHeight = lSplittedBlockSize;
20     }

21     return true;
22 }

23 pmStatus matrixMultiply_dataDistribution(pmTaskInfo
       pTaskInfo, pmDeviceInfo pDeviceInfo,
       pmSubtaskInfo pSubtaskInfo)
24 {
25     matMulTaskConf* lTaskConf =
           (matMulTaskConf*)(pTaskInfo.taskConf);

26     // Subtask no. increases vertically in output
           matrix (for increased locality)
```

```
27      size_t lBlocksPerDim = (lTaskConf->matrixDim /
           lTaskConf->blockDim);
28      size_t lBlockRow = (pSubtaskInfo.subtaskId %
           lBlocksPerDim);
29      size_t lBlockCol = (pSubtaskInfo.subtaskId /
           lBlocksPerDim);

30      // Subscribe to entire lBlockRow of the first
           matrix (with equal split)
31      pmScatteredSubscriptionInfo lSInfo1((lBlockRow *
           lTaskConf->blockDim + lBlockOffset) *
           lTaskConf->matrixDim * ELEM_SIZE,
           lTaskConf->matrixDim * ELEM_SIZE,
           lTaskConf->matrixDim * ELEM_SIZE,
           lBlockHeight);
32      pmSubscribeToMemory(pTaskInfo.taskHandle,
           pDeviceInfo.deviceHandle,
           pSubtaskInfo.subtaskId,
           pSubtaskInfo.splitInfo,
           INPUT_MATRIX1_MEM_INDEX, READ_SUBSCRIPTION,
           lSInfo1);

33      // Subscribe to entire lBlockCol of the second
           matrix
34      pmScatteredSubscriptionInfo lSInfo2((lBlockCol *
           lTaskConf->blockDim) * ELEM_SIZE,
           lTaskConf->blockDim * ELEM_SIZE,
           lTaskConf->matrixDim * ELEM_SIZE,
           lTaskConf->matrixDim);
35      pmSubscribeToMemory(pTaskInfo.taskHandle,
           pDeviceInfo.deviceHandle,
           pSubtaskInfo.subtaskId,
           pSubtaskInfo.splitInfo,
           INPUT_MATRIX2_MEM_INDEX, READ_SUBSCRIPTION,
           lSInfo2);

36      // Subscribe to one block of the output matrix
           (with equal split)
37      SUBSCRIBE_BLOCK(lBlockRow, lBlockCol,
           lBlockOffset, lBlockHeight,
           lTaskConf->blockDim, lTaskConf->matrixDim,
           pSubtaskInfo.subtaskId,
```

```
        pSubtaskInfo.splitInfo,
        OUTPUT_MATRIX_MEM_INDEX, WRITE_SUBSCRIPTION)

38      return pmSuccess;
39  }

40  pmStatus matrixMultiply_cpu(pmTaskInfo pTaskInfo,
        pmDeviceInfo pDeviceInfo, pmSubtaskInfo
        pSubtaskInfo)
41  {
42      matrixMultiplyTaskConf* lTaskConf =
            (matrixMultiplyTaskConf*)(pTaskInfo.taskConf);

43      size_t lBlockOffset, lBlockHeight;
44      if(!GetSplitData(&lBlockOffset, &lBlockHeight,
            lTaskConf, pSubtaskInfo.splitInfo))
45          return pmSuccess;

46      float* lMatrix1 = (float*)(pSubtaskInfo.memInfo
            [INPUT_MATRIX1_MEM_INDEX].ptr);
47      float* lMatrix2 = (float*)(pSubtaskInfo.memInfo
            [INPUT_MATRIX2_MEM_INDEX].ptr);
48      float* lMatrix3 = (float*)(pSubtaskInfo.memInfo
            [OUTPUT_MATRIX_MEM_INDEX].ptr);

49      size_t lSpanMatrix2 = (pSubtaskInfo.memInfo
            [INPUT_MATRIX2_MEM_INDEX].visibilityType ==
            SUBSCRIPTION_NATURAL) ? lTaskConf->matrixDim
            : lTaskConf->blockDim;
50      size_t lSpanMatrix3 = (pSubtaskInfo.memInfo
            [OUTPUT_MATRIX_MEM_INDEX].visibilityType ==
            SUBSCRIPTION_NATURAL) ? lTaskConf->matrixDim
            : lTaskConf->blockDim;

51      serialMatrixMultiply(lMatrix1, lMatrix2,
            lMatrix3, lBlockHeight, lTaskConf->matrixDim,
            lTaskConf->blockDim, lTaskConf->matrixDim,
            lSpanMatrix2, lSpanMatrix3);

52      return pmSuccess;
53  }
```

```
54 // Both input matrices are of size pMatrixDim *
      pMatrixDim and are stored back to back in
      pInputMatrices
55 // pOutputMatrix is of size pMatrixDim * pMatrixDim
      and contains the result after parallel matrix
      multiplication
56 // pMatrixDim is a power of 2 and is at least 2048.
57 void matmul(size_t pMatrixDim, float*
      pInputMatrices, float* pOutputMatrix)
58 {
59     pmInitialize();

60     pmCallbacks lCallbacks;
61     lCallbacks.dataDistribution =
          matrixMultiply_dataDistribution;
62     lCallbacks.subtask_cpu = matrixMultiply_cpu;
63     lCallbacks.subtask_gpu_custom =
          matrixMultiply_cudaLaunchFunc;

64     const char* lKey = "MMKEY";
65     pmCallbackHandle lHandle;
66     pmRegisterCallbacks(lKey, lCallbacks, lHandle);

67     size_t lMatrixElems = pMatrixDim * pMatrixDim;
68     size_t lMatrixSize = lMatrixElems * ELEM_SIZE;
69     unsigned long lSubtaskCount = (pMatrixDim /
          BLOCK_DIM) * (pMatrixDim / BLOCK_DIM);

70     pmTaskDetails lTaskDetails;
71     lTaskDetails.callbackHandle = lHandle;
72     lTaskDetails.subtaskCount = lSubtaskCount;

73     pmMemHandle lInputMem1, lInputMem2, lOutputMem;
74     pmCreateMemory2D(pMatrixDim, pMatrixDim *
          ELEM_SIZE, &lInputMem1);
75     pmCreateMemory2D(pMatrixDim, pMatrixDim *
          ELEM_SIZE, &lInputMem2);
76     pmCreateMemory2D(pMatrixDim, pMatrixDim *
          ELEM_SIZE, &lOutputMem);
```

```
77      pmRawMemPtr lRawInputPtr1, lRawInputPtr2,
            lRawOutputPtr;
78      pmGetRawMemPtr(lInputMem1, &lRawInputPtr1);
79      pmGetRawMemPtr(lInputMem2, &lRawInputPtr2);

80      memcpy(lRawInputPtr1, pInputMatrices,
            lMatrixSize);
81      memcpy(lRawInputPtr2, pInputMatrices +
            lMatrixElems, lMatrixSize);

82      pmTaskMem lTaskMem[MAX_MEM_INDICES];
83      lTaskMem[INPUT_MATRIX1_MEM_INDEX] = {lInputMem1,
            READ_ONLY, SUBSCRIPTION_OPTIMAL};
84      lTaskMem[INPUT_MATRIX2_MEM_INDEX] = {lInputMem2,
            READ_ONLY, SUBSCRIPTION_OPTIMAL};
85      lTaskMem[OUTPUT_MATRIX_MEM_INDEX] = {lOutputMem,
            WRITE_ONLY, SUBSCRIPTION_OPTIMAL};

86      lTaskDetails.taskMem = (pmTaskMem*)lTaskMem;
87      lTaskDetails.taskMemCount = MAX_MEM_INDICES;

88      matMulTaskConf lTaskConf = {pMatrixDim,
            BLOCK_DIM};
89      lTaskDetails.taskConf = (void*)(&lTaskConf);
90      lTaskDetails.taskConfLength = sizeof(lTaskConf);

91      lTaskDetails.canSplitCpuSubtasks = true;

92      pmTaskHandle lTaskHandle = NULL;
93      pmSubmitTask(lTaskDetails, &lTaskHandle);

94      pmWaitForTaskCompletion(lTaskHandle);

95      pmFetchMemory(lOutputMem);

96      pmGetRawMemPtr(lOutputMem, &lRawOutputPtr);
97      memcpy(pOutputMatrix, lRawOutputPtr,
            lMatrixSize);

98      pmReleaseTask(lTask);
99      pmReleaseCallbacks(lCallbacks);
```

```
100     pmReleaseMemory(lInputMem1);
101     pmReleaseMemory(lInputMem2);
102     pmReleaseMemory(lOutputMem);

103     pmFinalize();
104 }
```

Listing 10.2: File matmul.cpp

```
1 const float gZero = (float)0.0;
2 const float gOne = (float)1.0;

3 pmStatus matrixMultiply_cudaLaunchFunc(pmTaskInfo
    pTaskInfo, pmDeviceInfo pDeviceInfo,
    pmSubtaskInfo pSubtaskInfo, void* pCudaStream)
4 {
5     cublasHandle_t lCublasHandle;
6     cublasCreate(&lCublasHandle);

7     matMulTaskConf* lTaskConf =
        (matMulTaskConf*)(pTaskInfo.taskConf);

8     float* lMatrix1 = (float*)(pSubtaskInfo.memInfo
        [INPUT_MATRIX1_MEM_INDEX].ptr);
9     float* lMatrix2 = (float*)(pSubtaskInfo.memInfo
        [INPUT_MATRIX2_MEM_INDEX].ptr);
10    float* lMatrix3 = (float*)(pSubtaskInfo.memInfo
        [OUTPUT_MATRIX_MEM_INDEX].ptr);

11    cublasSetStream(lCublasHandle,
        (cudaStream_t)pCudaStream);

12    cublasSetPointerMode(lCublasHandle,
        CUBLAS_POINTER_MODE_HOST);

13    size_t lSpanMatrix2 = (pSubtaskInfo.memInfo
        [INPUT_MATRIX2_MEM_INDEX].visibilityType ==
        SUBSCRIPTION_NATURAL) ? lTaskConf->matrixDim
        : lTaskConf->blockDim;
```

```
14      size_t lSpanMatrix3 = (pSubtaskInfo.memInfo
           [OUTPUT_MATRIX_MEM_INDEX].visibilityType ==
           SUBSCRIPTION_NATURAL) ? lTaskConf->matrixDim
           : lTaskConf->blockDim;

15      cublasSgemm(lCublasHandle, CUBLAS_OP_N,
           CUBLAS_OP_N, (int)llTaskConf->blockDim,
           (int)lTaskConf->blockDim,
           (int)lTaskConf->matrixDim, &gOne, lMatrix2,
           (int)lSpanMatrix2, lMatrix1,
           (int)lTaskConf->matrixDim, &gZero, lMatrix3,
           (int)lSpanMatrix3);

16      cublasDestroy(lCublasHandle);

17      return pmSuccess;
18  }
```

Listing 10.3: File matmul.cu

Unicorn Publications

[1] Tarun Beri, Sorav Bansal, and Subodh Kumar. The unicorn runtime: Making distributed shared memory style programming efficient for hybrid cpu-gpu clusters. *IEEE Transactions on Parallel and Distributed Systems*, 2016 (Under Review).

[2] T. Beri, S. Bansal, and S. Kumar. A scheduling and runtime framework for a cluster of heterogeneous machines with multiple accelerators. In *Parallel and Distributed Processing Symposium (IPDPS), 2015 IEEE International*, pages 146–155, May 2015.

[3] Tarun Beri, Sorav Bansal, and Subodh Kumar. Prosteal: A proactive work stealer for bulk synchronous tasks distributed on a cluster of heterogeneous machines with multiple accelerators. In *2015 IEEE International Parallel and Distributed Processing Symposium Workshop, IPDPS 2015, Hyderabad, India, May 25-29, 2015*, pages 17–26, 2015.

[4] Tarun Beri, Sorav Bansal, and Subodh Kumar. Locality Aware Work-Stealing based Scheduling in Hybrid CPU-GPU Clusters. In *Parallel and Distributed Processing Techniques and Applications*, 2015.

Biography

Tarun Beri has passed his B.Tech. from Thapar Institute of Engineering and Technology, Patiala and his M.S.(R) from Indian Institute of Technology Delhi. His research interests include parallel and distributed systems, programming languages and frameworks for heterogeneous environments, vector graphics and machine learning. He holds several patents and publications in his name.

The following is the list of papers published and communicated by *Tarun Beri*:

1. Tarun Beri, Sorav Bansal, and Subodh Kumar. The Unicorn Runtime: Making distributed shared memory style programming efficient for hybrid CPU-GPU clusters. In IEEE Transactions on Parallel and Distributed Systems, 2016 **(Communicated)**

2. Tarun Beri, Sorav Bansal, and Subodh Kumar. A scheduling and runtime framework for a cluster of heterogeneous machines with multiple accelerators. In Parallel and Distributed Processing Symposium, IPDPS 2015, **(Accepted)**

3. Tarun Beri, Sorav Bansal, and Subodh Kumar. Prosteal: A proactive work stealer for bulk synchronous tasks distributed on a cluster of heterogeneous machines with multiple accelerators. In 2015 IEEE International Parallel and Distributed Processing Symposium Workshop, IPDPS 2015 **(Accepted)**

4. Tarun Beri, Sorav Bansal, and Subodh Kumar. Locality Aware Work-Stealing based Scheduling in Hybrid CPU-GPU Clusters. In Parallel

and Distributed Processing Techniques and Applications, PDPTA 2015 **(Accepted)**

The following is the list of patents held by *Tarun Beri*:

1. US 8,316,337 - Method and System for Optimally Placing and Assigning Interfaces in a Cross-fabric Design Environment **(Issued Nov 20, 2012)**
2. US 8,479,134 - Method and System for Specifying System Level Constraints in a Cross-fabric Design Environment **(Issued July 2, 2013)**
3. US 8,527,929 - Method and System for Optimally Connecting Interfaces Across Multiple Fabrics **(Issued Sep 3, 2013)**
4. US 8,650,518 - Method and Apparatus for Rule based automatic Layout Parasitic Extraction in a Multi-Technology Environment **(Issued Feb 11, 2014)**
5. Method and Apparatus for managing calendar entries in a document **(Filed Dec 19, 2013)**
6. Method and Apparatus for controlling display of digital content using eye movement **(Filed Apr 28, 2014)**
7. Fast, Coverage-Optimized, Resolution-Independent and Anti-aliased Graphics Processing **(Filed Mar 21, 2016)**
8. Non-Multisampled Anti-aliasing for Clipping Paths **(Filed May 23, 2016)**
9. Dynamic Spread Anti-Aliasing **(Filed Jul 27, 2016)**