Runtime Adaptation for Autonomic Heterogeneous Computing

Thomas R. W. Scogland

Dissertation submitted to the Faculty of the
Virginia Polytechnic Institute and State University
in partial fulfillment of the requirements for the degree of

Doctor of Philosophy
in
Computer Science

July 24, 2014
Blacksburg, Virginia

Keywords: Scheduling, Graphics Processing Unit (GPU), OpenMP, MPI, Heterogeneous Systems

Runtime Adaptation for Autonomic Heterogeneous Computing

Thomas R. W. Scogland

ABSTRACT

Heterogeneity is increasing across all levels of computing, with the rise of accelerators such as GPUs, FPGAs, and other coprocessors into everything from cell phones to supercomputers. More quietly it is increasing with the rise of NUMA systems, hierarchical caching, OS noise, and a myriad of other factors. As heterogeneity becomes a fact of life, efficiently managing heterogeneous compute resources is becoming a critical, and ever more complex, task. The focus of this dissertation is to lay the foundation for an autonomic system for heterogeneous computing, employing runtime adaptation to improve performance portability and performance consistency while maintaining or increasing programmability. We investigate heterogeneity arising from a myriad of factors, grouped into the dimensions of locality and capability. This work has resulted in runtime schedulers capable of automatically detecting and mitigating heterogeneity in physically homogeneous systems through MPI and adaptive coscheduling for physically heterogeneous accelerator based systems as well as a synthesis of the two to address multiple levels of heterogeneity as a coherent whole. We also discuss our current work towards the next generation of fine-grained scheduling and synchronization across heterogeneous platforms in the design of a highly-scalable and portable concurrent queue for many-core systems. Each component addresses aspects of the urgent need for automated management of the extreme and ever expanding complexity introduced by heterogeneity.

Dedicated to my fantastic wife Britany, who has always been there for me with love, support, and understanding.

ACKNOWLEDGEMENTS

First, I would like to thank my advisor, Professor Wu-chun Feng. Throughout my graduate school career, Wu has provided invaluable insight, support and encouragement time and again. He also provided me with invaluable opportunities to learn and to grow: from working on proposals, to reviewing papers, preparing class materials to writing and presenting technical papers. I would not be the same person today without his guidance. Thank you Wu for taking a chance on me, putting up with me, and teaching me to love research.

My sincere thanks also to my other committee members: Doctor Pavan Balaji, Professor Ali Butt, Professor Yong Cao, and Doctor Bronis de Supinski for their time, feedback and assistance during the course of this work.

This dissertation has also benefitted greatly from the involvement of all current and former members of the SyNeRGy lab. In particular I would like to thank Dr. Mark Gardner, Dr. Heshan Lin, Ashwin Aji, Jeremy Archuleta, Mayank Daga, Ganesh Narayanaswamy, Paul Sathre and Balaji Subramaniam for their valuable feedback and always stimulating discussions.

I have also had the good fortune to collaborate with both Lawrence Livermore National Laboratory and Argonne National Laboratory. Experiences as an intern at each lab has had a significant effect on the final shape of my dissertation. Special thanks go to Dr. Pavan Balaji (again), Dr. Bronis de Supinski (again) and Dr. Barry Rountree. These collaborations proved to be major turning points for me, both in my research and my life as a whole.

I also owe a debt of gratitude to all of the organizations that supported and funded my work: Virginia Tech, through teaching assistanceships; the Air Force Research Laboratory, through the Department of Defense National Defense Science & Engineering Graduate Fellowship program; the National Science Foundation; and the Air Force Office of Scientific Research. The staff of the Computer Science Department of Virginia Tech also deserves my thanks for their patience and continual assistance in managing the administrative and technical aspects of my work.

Lastly, and most importantly, I wish to thank my family for providing me with a limitless supply of support and encouragement. I will be forever grateful to my parents, William and Vicki, not only for their love and encouragement, but for teaching me the importance and value of education. Finally, I would like to thank my terrific wife, Britany, for her love, patience and support throughout this process. Without her, I may well never have reached this point, and I certainly would not have had as good a life along the way.

Contents

List of Figures

List of Tables

List of Abbreviations and Acronyms

AffinityTSAR Affinity-aware Task-Size Adapting Runtime. ix, 8, 10, 13, 21, 22, 27, 130, 146, 147, 149–151, 155, 156, 158, 188

BSP Bulk Synchronous Parallel. 2, 128, 133, 158, 190, 191

CAS Compare-And-Swap. xiii, 23–25, 164–166, 169, 173, 174, 177, 178, 183

CoreTSAR Task-Size Adapting Runtime. viii, xii, 6, 7, 10, 18–20, 28, 87–89, 93–96, 98, 99, 101, 103–117, 119–126, 128, 136, 140, 187, 188

CRQ Concurrent Ring Queue. 24, 179

FAA Fetch-And-Add. xiii, 23, 24, 164–166, 168, 169, 173

FFT Fast Fourier Transform. 27

HPF High-Performance Fortran. 20, 133, 134

HSA Heterogeneous System Architecture. 8, 191

LCRQ Linked Concurrent Ring Queue. 24, 173–175, 177–180, 183, 184

MIMD Multiple Instruction Multiple Data. 15

MPI Message Passing Interface. xi, 9, 26, 32–34, 37–40, 42–45, 50, 51, 186, 187, 191

MSI-X Message Signaled Interrupts (Extended). 187

NoC Network-on-Chip. 55

NUMA Non-Uniform Memory Access. ii, ix, xiii, 7, 20, 21, 30, 54, 55, 129, 130, 132, 136, 149–151, 154, 155, 157, 158

RDMA Remote Direct Memory Access. 187

SIMD Single Instruction Multiple Data. 15, 18, 65, 91, 108, 109

SIMT Single Instruction Multiple Thread. 15

SMP Symmetric Multi-Processing. 13, 54, 55

SyMMer Systems Mapping Manager. xi, 4, 5, 7, 9, 10, 26, 31, 32, 39, 40, 42, 43, 46–56, 128, 186–188

Chapter 1

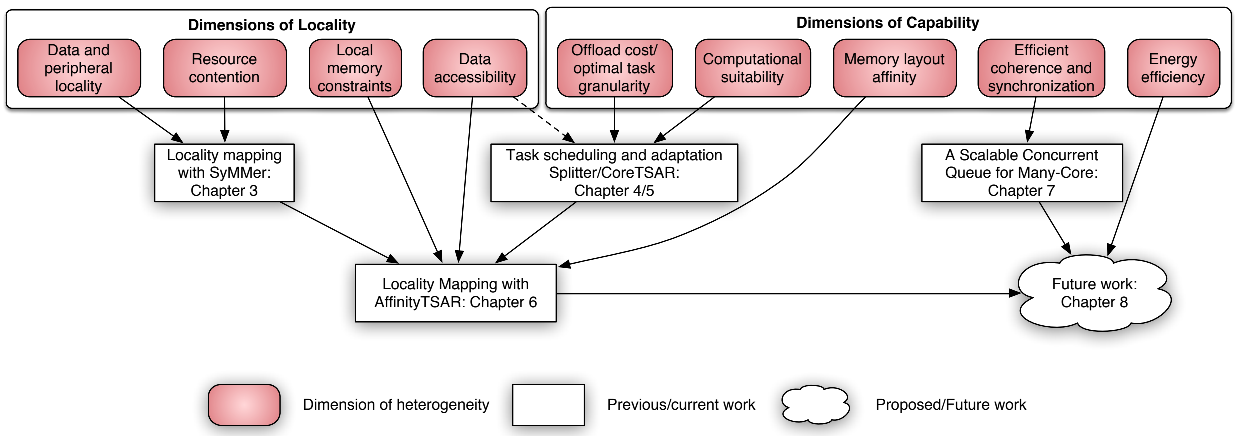
Introduction

1.1 Motivation

Over the past ten years, the average computer has evolved from a serial machine with a single processor into a highly complex parallel system. The rise in intra-node complexity is an unavoidable consequence of the ever-growing trend toward parallelism as the main driver of performance in new processor and system designs. While we classically expect our supercomputers and high performance workstations to have complex architectures, the complexity is now increasing throughout the computing ecosystem. Devices from supercomputer nodes to cell phones, CAD workstations to smart watches now carry multiple CPU cores. Beyond the expansion to parallelism, computational coprocessors with separate instruction sets, architectures and programming models are now being employed for general purpose computation alongside traditional CPUs. Over the last few years, heterogeneous computers have begun to dominate the top positions on the Top500 list [2] and yet more of the top of the Green500 list [40, 88]. They also accelerate operating system functions of the laptop on which this dissertation was written, and the smartphone sitting next to it.

With or without physically heterogeneous hardware, heterogeneity continues to increase with the rise in CPU core counts, multiple adaptive frequency domains, and increasingly hierarchical and complex memory systems. As an example, consider the composition of a node with two CPU sockets and four GPUs. The two sockets each have two memory nodes, associated with four cores apiece. Each GPU has a distinct address space and tens to hundreds of cores programmed like a Bulk Synchronous Parallel (BSP) [93] cluster. Further complicating the equation, some of the GPUs cannot directly inter-communicate due to multiple PCI-E bridges. In this case, we are effectively programming clusters of miniature constellations (in a node), the complexity of which is akin to how we programmed constellation-based supercomputers prior to the advent of multicore processors. A modern application *must* employ parallelism, and often heterogeneous parallelism, in order to attain reasonable performance. Even then, it must adapt to a myriad of possible hardware targets to the shifting capabilities of each piece of hardware at runtime.

Automatically and transparently adapting applications to system resources is now critically important. Just as evolution from single-celled organisms into human beings required the development of an autonomic nervous system to handle the intricate complexities of keeping our bodies running, the evolution from single-core computers into complex diversified systems requires new abstractions. Programmers have long relied on programming models, schedulers and middleware to manage the complexities of their hardware. The issue today is that our tools have not kept pace



Dimensions of Locality
Data and peripheral locality
Resource contention
Local memory constraints
Data accessibility
Dimensions of Capability
Offload cost/ optimal task granularity
Computational suitability
Memory layout affinity
Efficient coherence and synchronization
Energy efficiency
Locality mapping with SyMMer: Chapter 3
Task scheduling and adaptation Splitter/CoreTSAR: Chapter 4/5
A Scalable Concurrent Queue for Many-Core: Chapter 7
Locality Mapping with AffinityTSAR: Chapter 6
Future work: Chapter 8
Dimension of heterogeneity
Previous/current work
Proposed/Future work

Figure 1.1: Dimensions of heterogeneity as they relate to our work

with the hardware. They no longer offer the level of performance portability, or the depth of capabilities, to exploit heterogeneous systems for the full performance and efficiency benefits that they can offer.

Heterogeneous scheduling is a complex multi-dimensional problem with many aspects to consider. In this dissertation we explore the dimensions of heterogeneity in modern computers and present our approaches toward creating a new autonomic system through performance modeling, scheduling and programming model extensions. We attempt to summarize a number of the important factors in Figure 1.1 along with the elements of our work that address them.

1.2 Dimensions of Locality

While the most common *definition* of a heterogeneous system is one containing multiple differing computational units, heterogeneity caused by the topology and contention in a system is even more prevalent. Even in traditional “homogeneous” systems the performance of an application thread or process can differ significantly based on its placement in the system topology. A process’s distance from another that is shares data with, contention for cache resources, the latency to to a heavily used peripheral device or even the load on its core can raise or lower its affinity for that location. In cases where poor affinity causes reduced performance, it can be beneficial to alter the mapping of resources in order to achieve higher overall performance.

In order to investigate the causes, symptoms and treatments for such affinity problems we design the SyMMer [85] library discussed in Chapter 3. SyMMer exploits the fact that software is often as heterogeneous as the hardware on which it runs. SyMMer arranges the processes that compose the application across the available hardware to create the best possible match. Given an application where one process does mostly communication and the other computation, it schedules the communicator onto a core near the network interface, and moves the other away from protocol processing to reduce contention on caches. By scheduling heterogeneous processes across heterogeneous cores in this manner, SyMMer provides consistently high performance in otherwise unpredictable systems.

1.3 Dimensions of Capability

The most obvious source of heterogeneity in a system comes from actually building it with physically disparate compute units, such as a combination of CPU and GPU processors. Unlike traditional CPU-based systems where all compute units may be targeted with a single unified model and instruction set, there are currently no models that target accelerators as well in a unified way, forcing users to explicitly write their applications to target accelerators. While some of the concepts used in SyMMer could be useful in a physically heterogeneous environment, process to core mapping is no longer sufficient. A CPU process cannot, at this time, be mapped onto a core on a GPU. Thus, we investigate other abstractions to extend.

The performance and productivity balance of applications on CPUs and GPUs has become a popular target for discussion and attempted "debunking" in recent years [23, 60]. We do not address this directly because there are a myriad of accelerator programming models and systems in the marketplace. For example OpenMP for accelerators, or Accelerated OpenMP as we refer to all extensions of OpenMP for accelerators, allows a user to target accelerators by annotating their serial C or Fortran code with directives based on the classic shared memory directives of OpenMP.

Unfortunately however, Accelerated OpenMP lacks a native mechanism to workshare a parallel region across heterogeneous resources. One of the banner features of traditional OpenMP, its flexibility to adapt to different core counts and system scales automatically, is thus lost. Schedulers that exist for other models offer worksharing at the granularity of tasks, or function calls, but rely on the user to supply tasks of an appropriate granularity for use on all device types.

Reliance on a static task granularity is especially troublesome because heterogeneous systems present a variety of hardware, and there may be no consistently appropriate task size to use. The cost to offload a task to the device, its suitability to executing a certain pattern of computations and other factors can all affect how applications behave on a system. Some of these factors can even change while a code is executing, let alone across systems with different types or distributions of accelerators and CPUs. The result is that users who require performance portability and consistency face a choice. They can manually implement dynamic load balancing, distributed memory data coherency and handle the myriad of issues in each system. Alternatively they can use a generic task scheduler, but then must assign a task granularity, which is unlikely to be appropriate across all devices involved, and re-work their design to work within the constraints of that system.

We seek to address these shortcomings by creating a mechanism to allow safe and efficient work-sharing across devices, handling issues of computational suitability, disparate offload costs, and data accessibility, with a focus on maintaining compatibility with Accelerated OpenMP. First, we investigate the design of a coscheduling runtime for systems with CPUs and a single GPU device called Splitter, which is discussed in detail in Chapter 4. Second, we investigate the design of a syntax and runtime system capable of coscheduling across an arbitrary number of devices and device types, along with handling the attendant memory management. The Task-Size Adapting Runtime (CoreTSAR) library, discussed in Chapter 5 is the culmination of that effort. CoreTSAR automates load balancing, runtime adaptation of task granularity, and memory movement in order to provide the flexibility of multicore OpenMP on heterogeneous systems.

1.4 The Intersection of Locality and Capability

With ever-greater heterogeneity and parallelism in computer systems, efficiently managing the distribution of work and data *together* is critical. While existing task scheduling systems can function in this environment, they either transparently maintain the existing memory layout on accelerators, as CoreTSAR does, or force the user to re-write their compute code to conform to a new layout passed through the task scheduler, as in the case of OmpSs and StarPU. StarPU attempts to assist users in this process by providing "filters," functions that split data regions or add shadow rows so that a user need not directly implement these common operations. Even so, these filters must be applied manually, and are still just convenience wrappers for static manual division of work and data. Further, the compute code must be written to expect the transformed data rather than the original layout. These issues motivate a new way of handling input and output in a task scheduling system.

Our existing task scheduler for heterogeneous systems, CoreTSAR, does not possess a mechanism to address issues of affinity or locality. It can detect such issues, in terms of lowered computational capability, and adjust work assignments accordingly, but the underlying issue remains. Alternatively, SyMMer can resolve these issues by re-mapping processes to other cores in systems with symmetric access latency to main memory. Even so, SyMMer cannot address affinity issues effectively in Non-Uniform Memory Access (NUMA) systems, nor can it handle inherent load-balance in an application as CoreTSAR can. Combining the approaches taken in each, we develop a hierarchical work and data partitioning and scheduling system capable of worksharing across heterogeneous resources and *re-mapping, replicating and transforming memory,* in the way SyMMer remaps processes, to address affinity concerns.

In summary, we create a system capable of exploiting the memory hierarchy of heterogeneous and NUMA systems, as well as a given device's natural affinity for memory ordering, in novel ways. Some existing task schedulers, including StarPU, include modes that allow for locality-based optimization. In these, the focus is on reducing the number of copies of intermediate data and ensuring that combined tasks share a memory node. Our system likewise addresses these issues, but goes farther. We explore the additional benefits of a runtime that can reason about data requirements at a fine grain. For example, the benefits of data packing, or software caching across memory nodes. We call this system AffinityTSAR, and discuss it in greater detail in Chapter 6.

1.5 Building Capabilities for Future Work

As we look toward the future, and the next steps to be taken to extend the generality and effectiveness of our solutions to tackling heterogeneous systems, we also need to investigate basic building blocks of concurrent programs. The scheduling systems discussed throughout this dissertation assume a model of heterogeneous systems where a CPU is assigned to "push" work to a GPU or co-processor. Without the ability to have each device participate directly in scheduling, techniques such as work stealing and truly fine-grained task scheduling remain out of reach.

The newest generations of some accelerator platforms, notably AMD's Heterogeneous System Architecture (HSA) and reportedly the upcoming NVIDIA Maxwell architecture, are gaining support for the most basic building blocks necessary to take this next step. They each offer coherent access to main memory from the accelerator, and more importantly coherent atomics on memory visible to the entire *platform*. Given that, we have a vast array of new opportunities to improve the way heterogeneous devices interact with one another.

That opportunity comes with a cost however. Accesses to system-wide memory are expensive, high-latency, and only guaranteed to be coherent using atomic instructions (if then). Combining this fact with the reality that a platform-wide concurrent application might have *thousands* of threads within a node, and we exceed the scalability of most existing structures. For this reason, we investigate the creation of a highly scalable concurrent queue for many-core architectures in Chapter 7. It is not designed to provide a low-latency progress guarantee, but rather for the maximum throughput. On that basis, we find that it bests current designs by as much as one-thousand-fold on a GPU with over a thousand contending threads.

1.6 Organization of this Dissertation

The remainder of this dissertation is organized as follows:

Chapter 2 presents background information, related work and general context for the dissertation as a whole. The related work is categorized into four broad areas on which our work draws, and with which it contrasts.

In Chapter 3, we investigate the symptoms of poor process affinity in multi-process MPI codes, and design a set of heuristics to detect such asymmetric interactions. We present an implementation of these heuristics as a framework that is hooked into MPICH2, which we call SyMMer. Further, we demonstrate higher performance and greater performance consistency when using SyMMer enabled MPICH2 over the baseline implementation.

In Chapter 4, we design a runtime coscheduling system for cross-device worksharing for Accelerated OpenMP. We implement this mechanism as the Splitter library. This work demonstrates significant performance improvements over both CPU OpenMP and unmodified Accelerated OpenMP codes by allowing both resources to participate cooperatively.

Chapter 5 builds on Splitter to create a syntax and runtime system for arbitrary cross-device worksharing and memory management in Accelerated OpenMP. We implement this mechanism as the Task-Size Adapting Runtime (CoreTSAR) library along with a source-to-source translator component for the syntax. CoreTSAR demonstrates significant performance improvements both over unmodified Accelerated OpenMP codes as well as over an alternative heterogeneous task scheduling system.

Chapter 6 presents the synthesis of the affinity mapping and scheduling mechanisms used in CoreTSAR and SyMMer into a unified design. We design a library, AffinityTSAR, that automatically transforms the input and output memory layouts and memory placements of tasks independently of the user code within them, all in the context of the physical layout of the target system. With control over memory location and layout, AffinityTSAR can adjust the affinity of an application, as with SyMMer, without incurring costly process migrations across memory nodes as part of the worksharing process CoreTSAR employs. We find that while we do gain some of the expected benefits of using this approach on GPUs, especially in terms of using less memory and incurring fewer data transfers, we also gain significant benefits by exposing the data layout to the runtime at the CPU level.

Chapter 7 discusses the first building block of the next phase of our work, to expand our scheduling systems to support each device, or even the sub-components of each device, participating as first-class entities in scheduling. This chapter focuses on the design of a high-throughput highly scalable concurrent queue for many-core.

Finally, Chapter 8 presents a summary of the work presented in this dissertation and future work.

Chapter 2

Background and Related Work

As our work sits at the cross-section of task scheduling, hardware asymmetry and heterogeneous programming models, there is a significant body of related work. The following sections address each of these in turn.

2.1 Early Hardware Asymmetry

Li et al. [62] quantify performance differences caused by asymmetric multicore architectures. Further they define operating system constructs for efficient scheduling on such architectures. This closely relates to the *effective capability* of cores and devices discussed throughout this dissertation, but the authors only investigate physical asymmetry due to differing clock rates. In our work, we study heterogeneity across homogeneous systems as well as across systems with heterogeneous cores of various designs.

Chai et al. [30] study the impact of the Intel multicore architecture on the performance of various applications by looking at the amount of intra-node and inter-node communication performed. Some of the performance problems inherent in multicore designs are identified and the importance of multicore-aware communication libraries is underlined. They also discuss a technique of *data tiling* to reduce cache contention. While our initial work on affinity did not deal directly with this aspect, our combined approach to scheduling and affinity, AffinityTSAR, is designed to facilitate some of these transformations as part of the runtime system.

The impact of shared caches on operating system scheduling and placement of processes and threads on multicore architectures have also been studied [9, 39]. Each study found certain mappings of processes to cores would work better than others, generally focusing on the differences between the behavior in single-core Symmetric Multi-Processing (SMP) systems and multicores. We improve upon their work by generalizing the detection of factors that affect application performance on multicore systems and by devising a framework for dynamically mapping work to cores.

Various studies have further found that code written and optimized for a given piece of hardware, especially in the case of accelerators such as GPUs, cannot be trusted to run equally well on other hardware, even of the same type. Du et al. [36] conclude that performance across GPU architectures cannot be assumed to be portable, and offer some methods to make it more portable. Even within a given GPU architecture and vendor, Archuleta et al. [10] show that different GPUs react differently to algorithmic and mapping changes. Each case calls the portability of accelerator performance into question. The PolyBench/GPU project [46] attempts to address these issues through auto-tuning, establishing the lack of native performance portability in the process. Our work also attempts to address this issue by dynamically assigning appropriate amounts of work regardless of device performance.

2.2 Task Scheduling

Automatic task scheduling and task based parallelism as a concept are becoming increasingly popular in parallel computing. Traditionally the domain of shared memory parallel systems like OpenMP [33], Intel TBB [4], Cilk[++] [22, 61] or even languages like Erlang [94], task scheduling has begun to move outside that realm with the rise of frameworks for cluster and cloud computing. Frameworks like Charm++ [53] for clusters, and MapReduce [34] for the cloud, or heterogeneous environments [38, 48, 51, 81], offer many of the same productivity benefits of their shared memory ancestors, and provide interesting testbeds for advances in scheduling and work partitioning. MapReduce in particular has seen a significant amount of work to become more aware of, and amenable to, heterogeneous computing resources. CellMR [81] presents a MapReduce framework for IBM Cell-based clusters for example.

Along with these platforms, a wide range of work explores task scheduling policies and their effects on performance and scalability. Ayguadé et al. [14] question the need for a schedule clause in OpenMP. Their proposed alternative uses a history-based approach to calculate the schedule for future passes through a parallel region. Their results are quite positive, but do not universally best traditional OpenMP schedules. Their work influenced the design of our adaptive schedulers, as it offers low overhead and does not require that tasks be broken into very fine granularities before execution to achieve balance.

2.3 Accelerator Programming

To understand the extensions for heterogeneous programming models in this dissertation, one needs a basic background in accelerator programming. Many types of accelerators exist, from very CPU-like accelerators that share the cache hierarchy, to completely separate devices that must be treated as though they were compute nodes unto themselves. Arguably the most popular type of accelerator is the GPU, which is highly divergent from the shared memory programming model assumed by classic CPU models like OpenMP.

Using terminology coined for NVIDIA's CUDA [3] architecture, GPUs consist of several multiprocessors, each of which contains several cores, 8 to 32 depending on the generation up to this point. An NVIDIA Tesla C2050, for example, consists of 14 multiprocessors, each with 32 cores. These cores follow the Single Instruction Multiple Thread (SIMT) execution model in which all cores on a multiprocessor must run the same instruction each cycle although the hardware hides the details of computing each branch independently from the software if threads diverge. In essence, this mechanism allows a Single Instruction Multiple Data (SIMD) processor to be programmed as though it were Multiple Instruction Multiple Data (MIMD).

Unlike CPU cores, GPU cores do not have direct access to main memory but rather access a hierarchy of dedicated, on-board memory modules. Each multiprocessor has a set of caches and *shared memory*, a local memory space that only threads run on that multiprocessor can access. The only memory space that multiprocessors share is *global memory*, some parts of which can also be used as read-only texture memory. Our example architecture, the Tesla C2050, contains three gigabytes of global memory. While all cores share the global memory, similarly to how CPU cores share main memory, the GPU memory system does not offer coherence guarantees. Changes to global memory are only guaranteed to be globally visible after either the termination of a kernel (i.e., a GPU function.)

Programming a GPU effectively requires exploitation of many of these architectural details. The most significant issue is the separate address space from the CPUs, which means it cannot be treated as a shared-memory entity. Instead, we must program it as a distributed memory system. Essentially, we must send and receive messages to load data to the GPU, to retrieve results, or to synchronize.

2.4 Heterogeneous Programming Models

Heterogeneous programming is nearing the mainstream, and more models targeting accelerators are released all the time. In this space, we consider there to be three general classes of programming models, we discuss each class below.

Loop offload models: These carry the loop worksharing tradition of OpenMP and other directive-based or data parallel models to accelerators. Accelerated OpenMP is part of a long history of distributed memory extensions to OpenMP. These include the work of Basumallik et al. [16] as well as Sato et al. [84]. Implementations specifically for GPUs are beginning to proliferate. Academic versions such as the OpenMP to GPGPU translator proposed by Eigenmann [58], as well as in industry with the release of HMPP [35], OpenACC [7, 59], PGI Accelerator [95] and OpenMP for Accelerators OpenMP [19] are now all available. Each one offers a method for a user to target multiple GPUs and CPUs by explicitly splitting the workload and targeting each device with a region or codelet. They do not offer coscheduling within a single region. As they all offer a different implementation of C or Fortran to GPU translation, and do not offer in-region coscheduling, they are orthogonal to our work. Our goal is to offer an extension to this model, using one of these as a platform for coscheduling rather than competing with them.

An experimental extension to the Chapel language [90] adds native support for CUDA GPUs as part of their domain concept. Chapel is designed to support implicitly parallel algorithms, that are divided across multiple compute units and memory spaces based on user-defined partitioning. In adding GPUs to the list of targets, and adding capabilities for hierarchical parallelism, Chapel becomes a single target language capable of spreading across a wide range of heterogeneous platforms. The downside is that while it is a high productivity language, it requires that codes be rewritten completely to reap the benefits, making it impractical for large legacy codebases. Further, it lacks many of the optimizations that more established languages exploit.

While they are not strictly “loop-offload” models, several models use similar data-parallel region scheduling to target multiple devices. Most relevant of these are the approaches taken by Qilin [65], Mariano et al. [66], Maestro [91] and the scheduling framework presented by Ravi et al. [82]. In each case the authors present a novel heterogeneous programming API that supports adaptive scheduling between CPUs and a GPU. In the case of Qilin, the API is in the form of a C/C++ template library that operates on special array structures, and allows runtime generation of CPU and GPU code. Ravi et al.'s work generates CPU and GPU code from generalized reduction specifications. In each case, use of these options requires reimplementation of existing codes in the associated model, constrains the adaptive scheduling approach to that used by the respective system, and targets only one GPU. Qilin uses a class-based system similar to Thrust [5] and schedules with an adaptive approach similar to that used in our work on HTS [86], supporting one GPU, although it is calculated in a training pass and simply reused in latter runs. Maestro takes a similar approach to ours in that they model performance of subsequent kernels to determine the work division to use. The material differences are that Maestro is designed to work in OpenCL, and requires kernels to be aware of the work division for data movement and addressing purposes. The framework by Ravi et al. uses a chunk-based mechanism, with an option to combine chunks for scheduling on the GPU. CoreTSAR on the other hand is designed to handle memory movement and adaptive scheduling of work while preserving existing code inside the region. Further it supports a range of scheduling mechanisms that allow a user to select an adaptive or chunk based approach on a per region basis, it also supports an arbitrary number of arbitrarily capable GPUs and CPUs.

Block and grid models: These include CUDA [3] and OpenCL [6]. These low-level models specifically target GPU-like hardware by offloading blocks or groups of threads to an array of cores, each of which is a SIMD unit. Generally these cores share memory with one another but not directly with the CPU. The lower left of Figure 5.1 shows an example using CUDA. In addition to changing the array accesses, explicit memory allocation and copies are required to move data to and from the device. The loop is converted into a grid of threads, each of which execute a single iteration in the `cudag()` kernel, which must be called with the number of blocks and threads per block.

Blocked task models: With the proliferation of GPUs and other computational accelerators, several programming models and task schedulers have been proposed specifically for these environments. The most mature of these are StarPU [12, 13] and OmpSs [28, 37], both of which are block task schedulers. As a class, block task schedulers allow users to specify that a given function should be run asynchronously as a task. To do so, they define its input and output data blocks, dependencies and alternative implementations, if applicable, for the scheduler to be run on any appropriate device. In each case, they support automatic transfers of data to GPU resources where appropriate. While on the surface they appear quite similar to CoreTSAR, they address a different aspect of heterogeneous scheduling. Block task schedulers can be thought of as a version of the OpenMP "task" construct, they schedule asynchronous tasks with no knowledge of their relationships other than that specified in their dependencies. CoreTSAR addresses ranges of tasks that are known to be related to one-another, effectively extending OpenMP's parallel looping constructs. The major differences between the approaches derive from that fundamental difference in goals. In block task schedulers, users must explicitly divide their work into chunks, where the chunks must be of an appropriate size to be run either on an entire GPU or a single CPU core at the discretion of the runtime. CoreTSAR creates tasks to fit the computational needs of devices at the time, reducing the dependence on user-splitting of work and reducing the overhead of managing fine-grained tasks. As with task granularity, block task schedulers do not automatically divide data, requiring users to pass a start pointer to a contiguous chunk of data upon which to operate. The lack of general support for non-contiguous data patterns can require significant re-designing of code. In addition to the task block schedulers, Jiménez et al. [52] propose a runtime scheduling system based on performance in past runs. Unlike CoreTSAR however the goal of their work is the scheduling of multiple *applications* across a set of CPU and GPU resources to lower contention and increase performance of each application instance.

2.5 Memory Association and Distribution

Many approaches have been taken to address issues of NUMA and remote memory access issues in multithreaded, and particularly OpenMP, software. We classify these approaches into three broad categories: distributed-array models, shared-memory affinity models, and task-associative models. Each of these is discussed in greater detail below.

Distributed-array models: High-Performance Fortran (HPF) [64] and its descendants including the work by Bircsak et al. [20], XcalableMP [57, 75, 77] and Chapel [31, 90] tackle the issue of memory distribution through an explicit distribution of memory. Each of these employs either a directive or a data mapping specification to specify layouts. For example, HPF uses a directive, `DISTRIBUTE`, to specify how arrays should be distributed across the memory nodes of a system. Another directive is then used to bind computations on distributed arrays to distribution imposed by the previous distribute directive. Only threads local to the data will be used to compute on it. This approach has the downside of binding data, and thus computation, explicitly to devices, which restricts or at the least complicates load-balancing. While the small imbalances imposed by different CPU cores might be tolerable, incorporating GPUs and other factors make the need for re-balancing far greater. AffinityTSAR and its associated extension avoid this result by specifying the association of data to *computation* rather than to explicit locations or distributions in the system memory hierarchy. As a result, load-balancing can proceed however is best for the computation and the data distribution adjusted without further user intervention.

Shared-memory affinity models: Nikolopoulos et al. [74] developed a user-level dynamic page migration scheme for use with OpenMP on NUMA systems. Their subsequent paper [73] explicitly posed the question "is data distribution necessary in OpenMP?" The result at the time, the year 2000, was that it was not. The bandwidth between memories in ccNUMA machines of the time was relatively similar to local, and they expected it to improve over time. There was no reason to expect distributed memory devices to appear within a node either, making transparent dynamic page migration a viable alternative. Today however, the discrepancy in bandwidth between sockets can be extreme, has become more common, and we frequently must contend with GPUs or other non-coherent devices within the node. Given the new state of affairs, we contend that if it is not necessary, it is certainly worthwhile. Recent work to improve the user-level affinity control options in Linux also support this conclusion, including work on adding support for a next-touch affinity policy [63] and even extensions to the Linux kernel itself to support future auto-migration policies in the 3.8 release.

The approach taken in the ForestGOMP [25, 26] runtime system is particularly relevant to our work. The ForestGOMP approach focuses on two aspects of the memory association problem. The first is efficient support of nested thread groups, naturally mapping hierarchical computations to a hierarchical memory structure. We also employ nested parallel or target regions to the same effect in AffinityTSAR. The second is an alternate memory management interface to allocate, register, attach, or migrate memory regions. Using this interface, ranges of memory can be "attached" to "bubbles," a rough equivalent to a thread team, and migrated if appropriate by the runtime system. This approach bears important similarities to our own, but it lacks the flexibility introduced by our extension's ability to re-shape memory as it is passed into a region, and lacks native support for non-shared-memory devices.

Task-associative models: These models overlap significantly with the **blocked task models** discussed in Section 2.4. In scheduling their computational tasks, they also map the data explicitly tied to that task by dependencies into whatever memory space in the system requires it. The NUMA behavior often works in the same way, simply replicating memory for a NUMA node just as it would for an accelerator.

2.6 Concurrent Data Structures

In particular we are interested in efficient multi-reader multi-writer concurrent queues for use in scheduling systems. These queues have been studied for decades, nearly as long as computers with multiple computational units have existed to run them. We will elide some of the early history and refer the reader to the surveys provided by the papers referenced below, especially the Michael and

Scott [69] survey, which provides significant discussion of early designs. Below we discuss the four main classes of concurrent queue algorithms designed, to some extent at least, for performance.

Array queues: The array queue proposed by Gottlieb et al. [45] in 1983 is notable for scaling near-linearly to 100 cores in simulation at the time. The Gottlieb queue can scale to as many threads as the hardware can process concurrently due to the use of a combination of an FAA on a pair of counters to select a location and fine-grained locking on each location in the queue. Unfortunately however, the Gottlieb queue has been proven to be non-linearizable [21]. Orozco et al. [79] present two related array queues called the Circular Buffer Queue (CB-queue) and the High-Throughput Queue (HT-queue). The CB-queue merges the Gottlieb queue's two counters per side into one and preserves linearizability, but the authors assert that full and empty status cannot be determined for the CB-queue and provide only blocking enqueue and dequeue calls. Their solution to the weaknesses of the CB-queue is the HT-queue, which regains the ability to detect full and empty by using the same flawed double-counter mechanism employed by the Gottlieb queue.

Contended-CAS queues: Michael and Scott [69] present a pair of unbounded linked-list queues, one lock-free (MS-queue hereafter) and one lock-based. The MS-queue offers a linearizable, lock-free queue using a portable single-word CAS operation and has become the standard unbounded lock-free queue. An alternative bounded variant has also been proposed by Tsigas and Zhang (TZ-queue) [92], which uses a slightly different mechanism but performs similarly due to its use of contended CAS for committing operations. It also serves as the basis for a variety of later lock-free queues including the wait-free queues presented by Kogan and Petrank [55, 56], which offer even stronger progress guarantees, but with a cost that increases with each additional thread. Queues like these are also common components of relaxed queues [47, 54]. Relaxed queues reduce contention by relaxing the semantics of linearizability from a strict FIFO queue and spreading the operations across multiple underlying queues.

List of array queues: To gain some of the benefits of array queues, recent work has begun to employ linked lists of arrays. Gidenstam et al. [43] employ this technique to improve their cache efficiency. Morrison et al. [71] combine array and list queues to create the Linked Concurrent Ring Queue (LCRQ). The LCRQ retains lock-freedom while avoiding contended CAS operations in the common case, by using a FAA to select a target element like a blocking array queue might. Since the item selection method is inherently blocking, a dequeuer could reserve a location and then be forced to wait indefinitely on a slow enqueuer, the LCRQ maintains lock-freedom by allowing threads to skip operations that block for too long, introducing the need for retries. After a certain number of operations, or retries, the underlying Concurrent Ring Queue (CRQ) is closed, requiring enqueuers to allocate and initialize new CRQs and then enqueue them into the LCRQ. The downsides to this approach are the reliance on a double-wide CAS (which while common in x86 is not widely available in mobile or many-core architectures) and the reliance on the potentially frequent and expensive allocation and initialization of new CRQs.

Combining queues: Hendler et al. [49] embrace the serial nature of lock-free designs and propose a queue that uses coarse-grain locking along with a request-and-assist model called the flat-combining (FC) queue. Since only one thread is actually accessing the queue at any given time, fulfilling requests from other threads, the synchronization overhead and cache coherence traffic are comparatively low. The downside is that the maximum throughput of the FC queue is the maximum throughput of a single thread, regardless of the number of accessors. Even so, the throughput limit is higher than with CAS queues like MS-queue, but it is still bounded to serial performance. A recent addition in combining queues from Min et al.[70] extends the design to use both lock-free enqueues and combining dequeues to increase overall throughput without need for parameter tuning. While this approach does increase the potential concurrency to two, from one, it still limits the overall maximum throughput. This design is also not “lock-free” in the classic sense, nor is it non-blocking, because an enqueue operation can block the combining dequeue thread by being scheduled out after the tail swap and before updating the next pointer.

Finally, several of these queues have been evaluated on CUDA GPUs by Cederman et al. [29]. Out of a number of lock-based and two lock-free designs (i.e., MS-queue and TZ-queue), they conclude that for higher concurrency, the two lock-free queue designs are nearly always highest performing. The performance they observe for the MS and TZ queues is similar to that found in our results for the same number of workers on comparable GPUs.

2.7 Benchmarks

Throughout this dissertation we characterize the performance of our extensions and libraries by evaluating benchmarks with and without access to the functionality. Many benchmark suites for accelerators exist, but no suite of Accelerated OpenMP benchmarks existed during much of this work. As such, we incorporate benchmarks from some of those suites, but extended with our libraries or programming model extensions.

2.7.1 MPI

When evaluating the SyMMer framework and its effect on locality, we employ a set of MPI-enabled scientific applications and benchmarks. In particular, we use two molecular dynamics codes and a high-performance Fourier transform library.

GROMACS (GROningen MAchine for Chemical Simulations) [17] is a molecular dynamics application developed at Groningen University, primarily designed to simulate the dynamics of millions of biochemical particles in a molecular structure. GROMACS is optimized towards locality of processes. It splits the particles in the overall molecular structure into segments, distributes different segments to different processes, and each process simulates the dynamics of the particles within its segment. If a particle interacts with another particle that is not within the process' local segment, MPI communication is used to exchange information regarding the interaction between the two processes. The overall simulation time is broken into many steps, and performance is reported as the number of nanoseconds per day of simulation time. For our measurements, we use the GROMACS LZM application.

LAMMPS [80] is a molecular dynamics simulator developed at Sandia National Laboratory. It uses spatial decomposition techniques to partition the simulation domain into small 3D sub-domains, one of which is assigned to each processor. Such a decomposition allows it to run large problems in a scalable way wherein both memory and execution speed scale linearly with the number of atoms being simulated. We use the Lennard-Jones liquid simulation with LAMMPS scaled up 64 times for our evaluation and use the communication time for measuring performance.

Fourier Transform libraries are extensively used in several high-end scientific computing applications, especially those that rely on the periodicity of data volumes in multiple dimensions (e.g., signal processing, numerical libraries). Due to its high computational complexity, scientists typically use Fast Fourier Transform (FFT) algorithms to compute the Fourier transform and its inverse. FFTW [42] is a popular parallel implementation of FFT.

2.7.2 Accelerated OpenMP

We use four applications and the PolyBench/GPU [46] benchmark suite in our Accelerated OpenMP evaluations, and here characterize their general properties based on their performance when executed in CPU-only or GPU-only modes. CG [15] started out as a direct port of the NAS conjugate gradient benchmark, with a custom port of the main kernel of the benchmark from Fortran to C. In Chapter 6, when we evaluate AffinityTSAR, the CG benchmark is based on the native C SNU NPB suite [89] instead due to conflicts between the Fortran version's memory allocation behavior and the runtime. GEM [8] is a molecular modeling application for the study of the electrostatic potential along the surface of a macromolecule that has been extensively studied for GPU optimization [32]. Helmholtz is a discrete finite difference code that uses the Jacobi iterative method to solve the Helmholtz equation. K-means is a popular iterative clustering method. Our implementations of the 15 PolyBench/GPU benchmarks execute each computational kernel 10 times to mimic use in an iterative scientific application more closely. Tests at 5 and 15 kernel executions yield similar relative results. Since we are evaluating scheduling behavior, and not computational kernel performance, we made minimal changes in porting each benchmark. As such, our compu-

Benchmark	Passes	Time/ pass	CPU time	GPU time	Speedup on 1GPU
CG	1900	0.045	16.31	92.37	0.17
GEM	1	5.336	71.05	5.65	12.59
Helmholtz	50	0.138	1.18	7.22	0.16
kmeans	7	0.583	5.70	4.33	1.32
ATAX	10	0.646	32.23	6.60	4.88
BICG	10	0.822	21.86	8.78	2.49
CORR	10	0.162	157.73	1.64	96.07
COVAR	10	1.328	1558.30	13.80	112.90
FDTD2D	5000	0.000	0.99	1.23	0.80
GEMM	10	1.262	301.34	3.04	99.18
GESUMMV	10	1.902	2.10	20.38	0.10
GRAMSCHMIDT	10240	0.004	4.21	40.38	0.10
MVT	10	0.058	1.62	0.60	2.72
SYR2K	10	1.461	14.39	15.53	0.93
SYRK	10	0.769	7.86	8.18	0.96
THREEDCONV	10	1.031	5.77	10.95	0.53
THREEMM	30	0.284	126.03	3.78	33.35
TWODCONV	10	0.607	2.86	6.46	0.44
TWOMM	10	1.445	204.66	6.32	32.41

Table 2.1: Benchmark characteristics, times in seconds, time/pass for static schedule with CPUs and one GPU

tational kernels are not optimized for the GPU other than by the compiler.

For our purposes, benchmarks can be characterized by the number of passes through the parallel region that they make, the length of each of these passes, and how suitable they are to run on the GPU. Table 2.1 characterizes these properties for each benchmark. The table exhibits a wide range in number of passes through the parallel region – 1 to 1900 passes in the applications, and as high as 10240 passes for the GRAMSCHMIDT benchmark. Our adaptive scheduler operates primarily at the boundaries of parallel regions, so this number can greatly affect our results. For example, in the GEM benchmark, the adaptive schedulers are identical to the static scheduler because the training pass is the only pass in the application. Conversely, CG performs many short passes, which allows CoreTSAR to adjust scheduling decisions but incurs high scheduler overhead and data copy costs.

The table also shows a wide range of performance ratios. Values range from a $10\times$ slowdown to a $113\times$ speedup going from eight CPU cores to one GPU. Running GEM on only one GPU finishes the problem more than $10\times$ faster than on eight server class Intel CPU cores. CORR and COVAR also show extreme suitability, largely due to the static schedule employed in the CPU tests. Because the workloads are imbalanced, each CPU core performs a different amount of work. The GPU test, because of the load-balancing effect of over-provisioning work-groups on GPUs, handles this variation better. If we use the OpenMP dynamic schedule, COVAR runs in approximately 150 seconds, $10\times$ faster than the static performance. Alternatively, GRAMSCHMIDT and Helmholtz are not suited to GPU computation according to these results. Generally, the suitabilities match our expectations, with the exception of CG. CG is, generally speaking, suitable for GPUs. Some of our experiments on other platforms showed a ratio of approximately 0.55 on one GPU. Here, the GPU version takes more than $5\times$ longer than the CPU version. This is due to the high cost of data re-distribution across GPUs each iteration. We leave optimization of CG to future work.

Chapter 3

Affinity Mapping with SyMMer

3.1 Introduction

This chapter explores heterogeneity as caused by system topology and resource contention issues, which are along the dimension of locality. Many still cling to the assumption that because hardware is physically symmetric, cores of the same design and clock rate for example, the hardware's *effective capability* will also be the same. Unfortunately, as uncovered in previous work by Narayanaswamy et al. [72], this assumption is often faulty. Two primary issues prevent symmetric hardware from behaving in a consistent manner.

The first comes from the fact that computer designs are moving consistently toward greater distribution of resources and loosening of time guarantees to increase performance. In a system with multiple NUMA nodes, or hierarchically shared caches, codes that access shared memory regions will see higher effective capabilities from nearby cores. Likewise, peripheral devices are now being distributed more and more, with many multi-socket systems shipping with multiple IO hubs as well.

The second issue is that however symmetric the processing hardware may be, operating system processing and peripheral handling are not perfectly distributed. For example, network interfaces operate by interrupts and traps to inform the system when new data is available. These interrupts are normally routed either to one core, or distributed by round robin scheduling, rather than being sent to the core requesting the work. As a result, a single core may become overloaded with processing. Additionally, the data communicated by the peripheral is loaded into the cache of that core for processing. The data is then highly local both to it, and to cores sharing a cache with it.

In the rest of this chapter, we attempt to distill these issues down to a set of symptoms, causes, and ultimately solutions. Specifically, we investigate asymmetric behaviors that persist even in a physically symmetric design, in this case a dual-socket dual-core uniform memory access system. This work addresses the dimensions of data and peripheral locality as well as contention. The contributions in this chapter can be summarized as follows:

1. A detailed analysis of asymmetric interactions between multicore architectures and the communication stack, culminating in three primary symptoms of such interactions;
2. Intra- and inter-node heuristics that detect these symptoms, and generate process-core affinity mappings that result in greater and more consistent performance;
3. The SyMMer library, an implementation of these heuristics along with a mechanism to apply

the resulting mappings, which we insert into MPICH2 [11] for the purpose of our evaluation;

4. A performance evaluation comparing SyMMer-enabled MPICH2 to a vanilla version across microbenchmarks and scientific applications, demonstrating that SyMMer-enabled MPICH2 can provide *more than a* $2\times$ *improvement* in communication time, and a 10-15% improvement in overall application performance.

The rest of the chapter is organized as follows. We describe the interactions between the network protocol stack and multicore architectures in Section 3.2. Section 3.3 describes our approach to intelligent handling of these interactions, as well as the design of the SyMMer library. Experimental results evaluating our approach is presented in Section 3.4. Discussion on alternative multicore architectures is presented in Sections 3.5. Section 3.6 presents a summary of the chapter.

3.2 Asymmetric Interactions in Multicore Systems

This section investigates the heterogeneity found in our example symmetric system, as well as the symptoms thereof. We primarily focus on the effects of the communication stack and high speed network interface, but find similar issues with compute peripherals such as GPUs in Chapter 5. Specifically we focus on the MPICH2 implementation of MPI using Linux TCP/IP sockets over 10 gigabit Ethernet, but other protocols that implement system notification and processing behave similarly.

3.2.1 Interactions with the Communication Stack

Logically, when an application sends data it formulates a message and hands it over to MPI to transmit. MPI buffers this data, appends a header, and passes it to the TCP stack for transmission. On the receiver side, the network adapter transfers received packets to the socket buffer, where they are assembled, checked for validity, and held until requested by the application. When the application calls an MPI receive operation, this data is copied out of the socket buffer into the application designated receive buffer.

Architecturally, on the other hand, there are many hidden side effects below the clean abstraction provided by the higher layers of the communication stack. In this section, we present the impact that these effects can have on a core's effective capability to perform computation and communication.

Processing Impact

When a packet arrives, the network adapter places the data in memory, and raises an interrupt to inform the communication stack that a message is ready for processing. For most system architectures, the processing core to which the interrupt is directed is either statically or randomly chosen using utilities such as *IRQ balance*. However, in both approaches, the *chosen core* is not guaranteed to be the core used by the process requesting the data. All of the processing done at the receiver adds overhead to a core that may not have requested it, charging the overhead to whichever process is running on that core.

This protocol processing computational load is *in addition to* whatever computation the application process is performing. Thus, as far as the application processes are concerned, the *chosen core* tends to have a reduced *effective* computational capability as compared to the remaining cores.

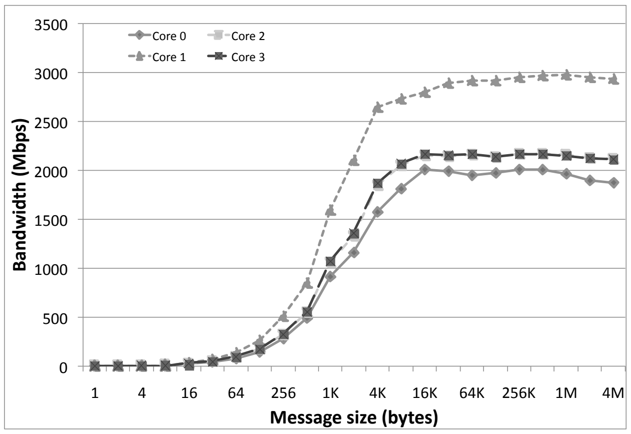


Figure 3.1: MPI bandwidth on Intel

Cache Transaction Impact

Aspects of protocol processing such as data copies and checksum-based data integrity require the communication stack to touch the data before handing it over to the application (through the socket buffer). For example, when the TCP/IP stack performs a checksum of this data, it has to fetch the data into its local cache. Once the checksum is complete the application has to fetch this data to its local cache. If the process resides on the same die as the protocol processing core, then the data may already be in one of its cache levels and can be quickly accessed. Instead, if the application process resides elsewhere, then the data must be fetched using a cache-to-cache transfer or loaded from main memory.

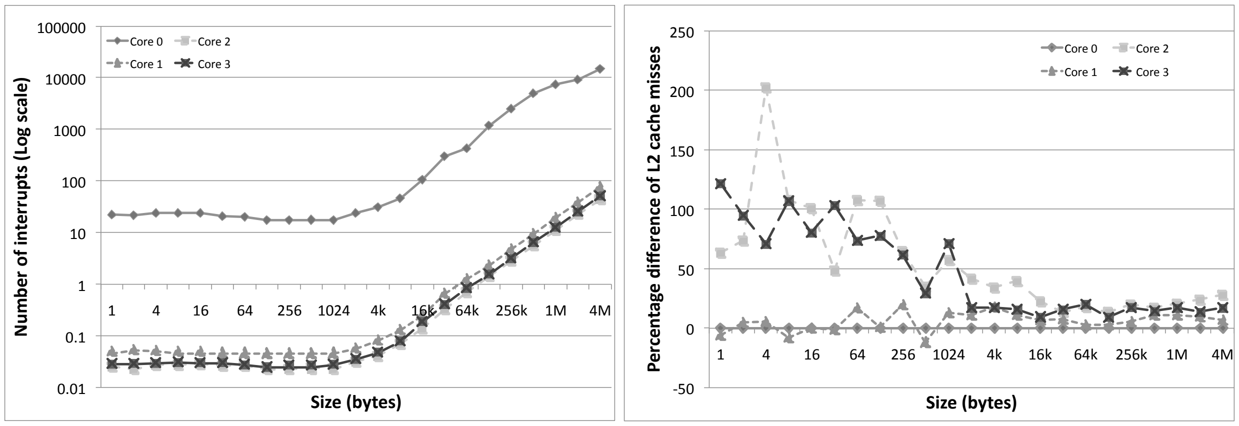


Figure 3.2: (a) Interrupts per message and (b) cache analysis

To demonstrate these effects (processing impact and cache transaction impact), we measured the communication bandwidth between two processes on two Intel dual-processor/dual-core machines, with the processes bound to different cores of the system. Figure 3.1 demonstrates these measurements. The interrupts (and hence the protocol processing) are always directed to core 0 in this test. As shown in the figure, when each process is bound to core 0, it is forced to compete with the protocol processing for cycles and suffers a performance penalty. On the other hand, when each process is bound to core 1, they do not conflict with protocol processing. Further, since core 0 receives the data for processing, the data is available locally on core 1's shared L2 cache. Thus, core 1 receives the benefits of in-cache access to its data through a *free ride* from core 0. The locally accessible data means that core 1 has the best performance for this configuration, increasing network bandwidth by nearly a gigabit per second over the other cores. Cores 2 and 3 do not face protocol processing overheads, nor do they have the benefit of increased cache hits. Thus, their performance is between that of cores 0 and 1. This behavior is confirmed in Figures 3.2 (a) and 3.2 (b), using performance counters for number of interrupts and cache misses occurring on each core.

3.2.2 Identifying the Symptoms of Asymmetric Interaction

Directly tracking interactions between the communication stack and the multicore architecture requires detailed monitoring of various aspects of the kernel and hardware as well as correlation between the various events. In order to keep runtime overhead low, we take an indirect approach to detecting these interactions by monitoring for abstract *symptoms* in application behavior triggered by known interactions. While a certain interaction can result in a symptom, the occurrence of the symptom does not necessarily mean that the interaction has taken place. That is, each symptom can have a number of *causes* that could have triggered it. While this does result in false positives, the benefits in computational complexity and overhead outweigh the detriments. The remainder of this section describes the symptoms that we track in order to detect asymmetric interactions in a system.

Symptom 1: Communication Idleness

If a core is busy performing protocol processing, the number of compute cycles that it can allocate to the application is lower than other cores, thus slowing down the process assigned to this core. A remote process communicating with this *slower* process would observe longer communication delays, and be idle for longer periods, as compared to other communicating pairs. This symptom is referred to as *communication idleness*. This symptom can also be caused by native imbalance in the application's communication and computation pattern, but in that case the application tends to benefit from re-mapping regardless.

Symptom 2: Out-of-Sync Communication

Communication middleware such as MPI perform internal buffering of data before communicating. Assuming both sender and receiver have equal computational capabilities, such a sender can completely transfer its data without resorting to buffering. Let us consider a case where process A sends data to process B and both processes compute for a long time. Then process B sends data to process A and again both processes compute for a long time. Now, suppose process B is *slower* than process A, and A sends data to B. In this case, process A's MPI library may buffer the data and continue since process B is not ready to receive more. From process A's perspective, the send has completed once the data has been handed over to the MPI library; thus, it moves on to perform its computation.

After its computation, when it tries to receive data from process B, it sees that the previous data that it attempted to send is still buffered and tries to send it out again. By now, B is ready to receive more data and the send is successful. After receiving the data, process B goes off to perform its computation, while process A waits to receive its data. At this point, both processes, and any other processes waiting on them, have been forced to wait through two additional compute cycles. This behavior is caused because, despite the fact that processes A and B are performing similar tasks, they are *out-of-sync* due to the difference between their effective computational capabilities.

Symptom 3: Cache Locality

When a core performs protocol processing, it fetches the data to its cache in order to perform the data integrity check. Thus, if the process that is waiting for the data shares a cache with the protocol processing core it will have the data without further delay. On the other hand, if the process is on a core that does not share cache with the protocol processing core, it will require a costly data transfer to obtain it. Thus, a process that transfers large amounts of data will gain a performance benefit from sharing a level of cache with the protocol processing core. In addition to this effect, we find that processes that share data locally also present this symptom. While MPI processes are not known for inter-process shared memory, the Nemesis implementation of MPICH2 employs shared memory for communication between local ranks, increasing the chance of locally shared data. For example, if one process frequently sends to or receives from another process on the same machine, the data will be transferred between their caches on a regular basis, and benefit from a shared cache.

3.2.3 Our Focus

As shown above, the effective capabilities of physically identical cores are frequently not the same. The result being that where a process is currently running can greatly affect its performance. The appropriate response to this issue is to map the processes to the correct cores. Traditionally the role of ensuring that processes are on the best possible processing unit has fallen to the system scheduler, but current schedulers do not take the *effective capability* of cores into account. Another option is to map the processes manually as done in previous work [72], but the time and difficulty of that method makes it impractical for many situations. For example, projects where the code base changes frequently, or applications where the work done per process changes frequently. As such, an automatic or dynamic approach is necessary. In the rest of this chapter we will describe and evaluate our solution, the dynamic mapping framework known as SyMMer.

3.3 Design and Implementation

This section describes our novel SyMMer library and its associated monitoring metrics. The SyMMer library, shown in Figure 3.3, is an interactive monitoring, information sharing, and analysis framework that can be tied into existing communication middleware such as MPI or potentially local parallel runtimes such as OpenMP. The library directly implements the monitoring, communication, and analysis components, while the metrics used for decision making are separately pluggable as we will describe in Section 3.3.1. Decoupling the components in this manner allows the design of metrics and the programming environment where they are used to be independent of one other.

Interaction Monitoring: The interaction monitoring component is responsible for monitoring system state. In particular it is meant to track system specific information (hardware interrupts, software signals), communication middleware specific information (MPI data buffering time and other internal stack overheads) and processor performance counters (cache misses, interrupts). This component utilizes existing libraries and the operating system for as much of the instrumen-

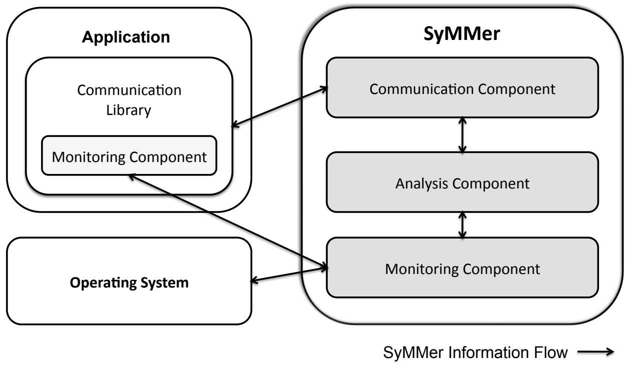


Figure 3.3: The SyMMer component architecture

tation as possible, while relying on built-in functionality for the rest. For example, processor performance counters are measured using the Performance Application Programming Interface (PAPI [1]) and system specific information through the `/proc` file-system. While MPI specific information can be monitored through the MPI-3 tools information interface [68], several of the MPI implementations did not support this interface fully during our investigation. Thus, we use built-in profiling functionality to obtain such information.

In order to minimize monitoring overhead, the monitoring component dynamically enables only the components that are required for the metrics being used. For example, if no metric makes use of processor performance counters, such information is not monitored.

Communication: The communication component is responsible for the exchange of state or data between different processes in the application. Several forms of communication are supported, including point-to-point sharing models (for sending data to a specific process), collective sharing models (for sending data to a group of processes) and bulletin board models (for publishing events that can be asynchronously read by other processes). For each of these models, both intra-node communication (between cores on the same machine) and inter-node communication (between cores on different machines) are provided. Inter-node communication is designed to avoid out-of-band communication by making use of added fields in the existing packet headers. Whenever a packet is sent, the sender adds the information that needs to be shared to the outgoing header. The receiver, on receiving the header, shares this information with other local processes using regular out-of-band intra-node communication. This approach has the advantage that any single inter-node communication can share information about all processes on the node. Intra-node communication, on the other hand, has been designed and optimized using shared memory without requiring locks or communication blocking of any kind. Our shared memory communication system provides a great deal more flexibility and reduces the overhead of our framework significantly.

Analysis: The information collected by the monitoring component in each process and shared with other processes is *raw*, in the sense that there is no correlation between the different pieces of information. Further, the monitored information is low-level data that needs to be processed and summarized into higher level and more compact information before it can be processed by the various metrics. The *information analysis* component performs all analysis and summarization of the data. This component also allows the various pluggable metrics to be arbitrarily prioritized for cases where an application may show multiple symptoms. Finally, each monitoring event has a certain degree of inaccuracy associated with it. As such, some monitoring events have more *data noise* than others. To handle such issues, the analysis component allows different monitors to define *confidence levels* for their monitored data. Depending on the number of events that are received, the analysis component can accept or discard events based on their confidence levels, using appropriate thresholds.

In addition to its duties in preparing data for the metrics to use, the analysis component takes action when a metric determines that it is required. Once the analyzed data identifies two processes that are currently scheduled on cores that are not best suited for them, but can potentially improve performance by swapping the processes between the cores, the analysis component is responsible for performing the swap as quickly and efficiently as possible. The communication component comes into play in a handshake phase used to minimize the time the processes spend on the same core during a swap. Swapping in SyMMer is accomplished using the get and to set affinity functions available on Linux to set each process to have an affinity with only one core, and when swapping simply changing the affinity of each process to their new target core. The design does not mandate this method, and in fact would benefit from the availability of an interface that would allow one to get and set the current core without affinity being set.

3.3.1 Metrics for Mapping Decisions

In this section, we discuss different metrics that can be plugged into the SyMMer library. Specifically, we focus on metrics that diagnose the symptoms noted in Section 3.2.2. Since our current reference implementation makes use of MPI, we describe these metrics in relation to MPI facilities, but they are equally applicable to other implementations and models. In all cases, the default mapping is assumed to be random with respect to the capabilities of the cores corresponding to the demands of the processes. While the scheduler generally puts processes on cores in order, the capabilities of each core per machine, the demands of the processes, and the order of launching the process is sufficiently variable for it to be considered random.

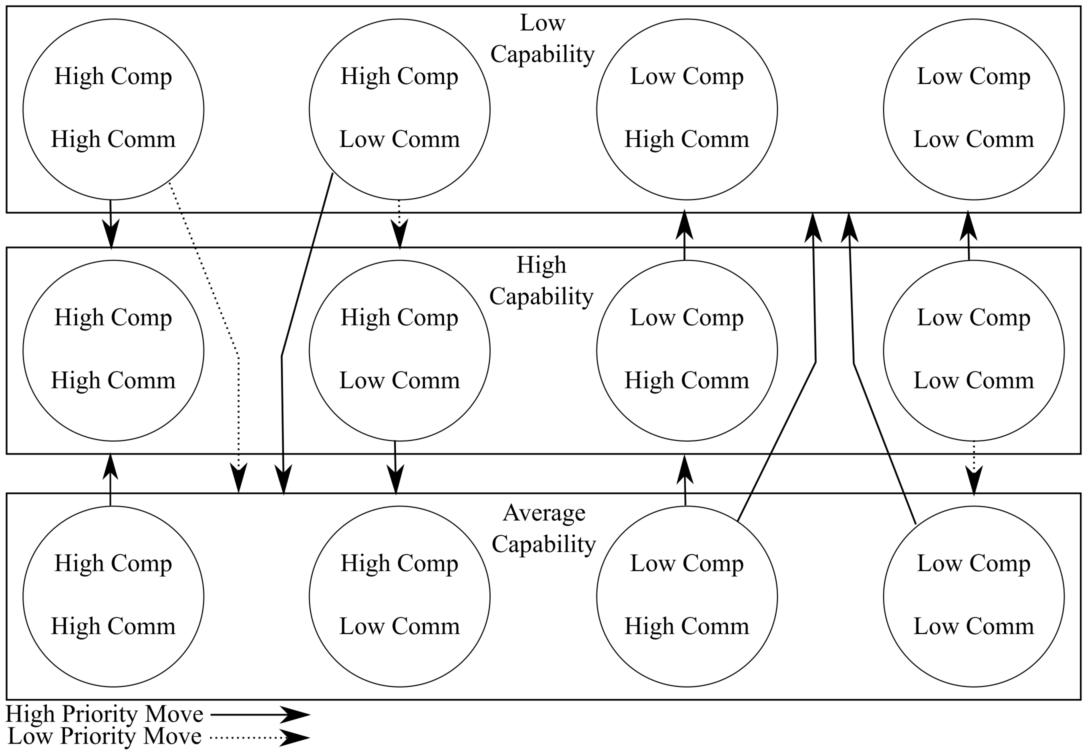


Figure 3.4: Process states: Each circle is a possible process behavior, each box a level of core capability, and each line a potential destination (based on the idleness metric)

Communication Idleness Metric

This metric is defined based on the *communication idleness* symptom defined in Section 3.2.2. The main idea of this metric is to calculate the ratio of the idle time (waiting for communication) and computation time of different processes. This metric utilizes the MPI monitoring capability of the SyMMer framework to determine this ratio. The idle time is measured as the time between the entry and exit of each blocking MPI call within MPI's progress engine. Similarly, the computation time is measured as the time between the exit and entry of each blocking MPI call. The computation time, thus represents the amount of computation done by the application process, assuming that the process does not block or wait on any other resource.

Hence, the computational idleness metric represents the idleness experienced by each process. For example, a process that has a high communication idleness can allow for other computations such as protocol processing. Processes with little idle time on the other hand cannot tolerate even a small amount of protocol processing without seeing material performance degradation. Comparison of this idleness factor between different processes provides an idea of which processes are more suited for sharing the protocol processing overhead. Clearly, the idleness metric needs to be compared only for processes running on the same node. Hence this metric only uses the intra-node communication channel discussed above.

Once the idleness is compared, if a process on a core other than the protocol processing core is experiencing a high idleness ratio, and the process on the protocol processing core is experiencing low or no idleness, this metric determines that they are suitable for a swap. Figure 3.4 illustrates this process by depicting each state a process may be in, and the core or cores to which it might switch, from that state. Once the swap is made the idleness of the two processes should be more similar, both to each other as well as to the average idleness of all local processes. While on the surface communication idleness appears similar to the cache locality metric, they differ substantially in that the locality metric tries to improve locality between local processes based on cache misses, whereas the idleness metric attempts to match the computation or communication demands of a process with a favorable core to balance the computational load.

Out-of-sync Communication Metric

The out-of-sync metric captures the amount of computation performed by a process while having unsent data buffered in its internal MPI buffers, and compares that with the wait time of other processes. As described in Section 3.2.2, this metric represents the case where unsent data followed by a long compute phase results in high wait times on the receive process. When the computation time (with buffered data) is above a threshold, a message is sent to all processes informing them of this symptom. Similarly, when the wait time of a process is above a threshold, this information is distributed as well. If communicating peer processes observe these conditions, a leader process is dynamically chosen, which then analyzes the data and computes the best mapping. The leader then sends the new mappings to all necessary processes, usually causing two of them to swap.

The mapping decision works to move each process to a core that is similar to the one on which its peer is running. In some cases a disparity between only two processes can cause all processes in an application to see out-of-sync communication, and fixing that one pair repairs this behavior for all others as well. This metric is unique among the three discussed in this chapter, in that it is inherently inter-node. In fact, it occurs between a minimum of two nodes, and all nodes that present the symptom make a decision that one or more nodes should change their mappings.

Cache Locality Metric

The cache locality metric utilizes the L2 cache misses monitored by the SyMMer library. This metric is specific to the processor architecture, and relies on the locality of cache between cores on the same die. If the number of cache misses for a process is sufficiently greater than those observed by another process on the same node, then these two processes can be swapped as long as the communicating process is moved *closer* to the core performing the protocol stack processing. In some cases it can even determine that the protocol processing core is the best location to place a process that has high enough cache misses, since that core will have no misses, the same as the other core on the same die, allowing two processes to be mapped in this manner rather than just one. Such a mapping is an example of prioritized metrics, in that in these cases, following the communication idleness metric would result in a below optimal result. Again, this metric only relies on intra-node communication as it is only used to switch processes to the respective cores within the same node.

3.4 Evaluation

In this section, we evaluate our implementation through multiple microbenchmarks in Section 3.4.1 and the GROMACS and LAMMPS applications and the FFTW Fourier Transform library in Section 3.4.2.

The testbed used for our evaluation consists of two Dell PowerEdge 2950 servers, each equipped

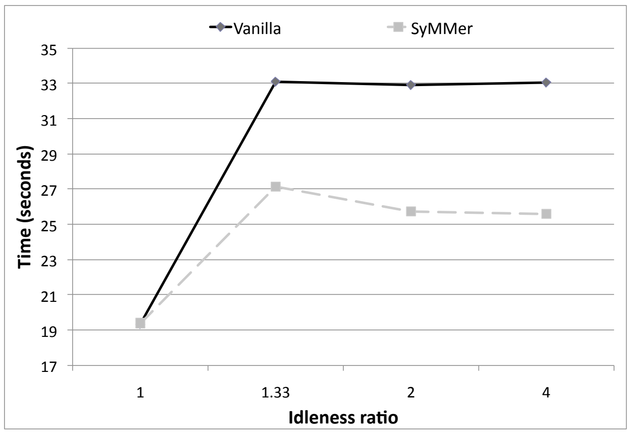


Figure 3.5: Communication idleness benchmark performance

with two dual-core Intel Xeon 2.66GHz processors. Each server has 4GB of 667MHz DDR2 SDRAM. Each processor has a 4MB L2 cache that is shared between the two cores. The machines run Fedora Core 6, with Linux kernel version 2.6.18 and are connected using NetEffect NE010 10-Gigabit Ethernet network adapters.

3.4.1 Microbenchmark Evaluation

In this section, we evaluate three microbenchmarks that were developed specifically to induce each of the symptoms described in Section 3.2.2 individually without being affected by the other interactions. In Section 3.4.2, we will evaluate the SyMMer library with full applications and scientific libraries.

Communication Idleness Benchmark

The communication idleness benchmark stresses the performance impact of delays due to irregular communication patterns. In this benchmark, processes communicate in pairs using MPI_Send and MPI_Recv, with each pair performing *different* amounts of computation between each communication step. Thus, a pair that is performing less computation spends more time in an idle state waiting for communication. Such processes are less impacted by having the protocol processing overhead on the same core as compared to other processes which spend more of their time doing computation.

For this benchmark, we define an idleness ratio to be the ratio between the computation done by the pair doing the most computation and the pair doing the least. This ratio represents the amount of computational irregularity in the benchmark. Thus an idleness ratio of *one* means that all processes in the benchmark perform the same amount of computation, while a value of *four* means that one communicating pair performs up to 4-times more computation than another.

In Figure 3.5, we plot the time taken for the benchmark to execute with various idleness ratios. We observe that both vanilla and SyMMer-enabled MPICH2 perform the same for an idleness ratio of *one*. Given that an idleness ratio of *one* represents a completely symmetric benchmark, and thus no process would be less affected by the overheads on core zero, SyMMer has no room for improvement. As the idleness ratio increases however, with vanilla MPICH2, the performance of the benchmark increasingly depends on the mapping of processes to cores. Such dependence makes it possible for SyMMer to use the disparity in computation to achieve a more optimal arrangement,

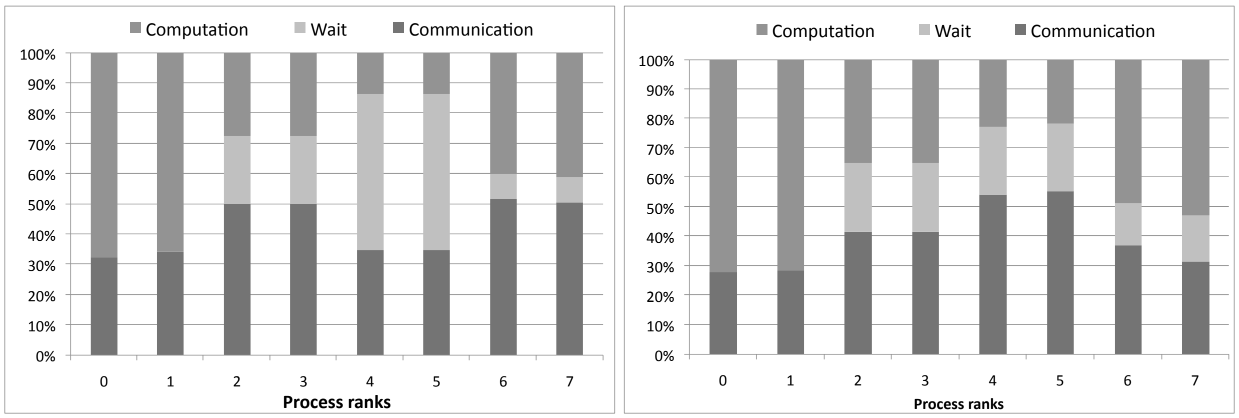


Figure 3.6: Communication idleness benchmark devision of time: (a) vanilla MPICH2 (b) SyMMer-enabled MPICH2

reducing runtime by up to 30%.

To analyze the behavior of this benchmark further, we show the distribution of time spent in computation, communication and waiting for communication in Figures 3.6(a) and 3.6(b) for vanilla MPICH2 and SyMMer-enabled MPICH2 respectively. For consistency both are based on the same initial mapping of processes to cores with an idleness ratio of *four*. Figure 3.6(a), shows that the wait times for the different processes are quite variable. Some processes spend a lot of time waiting for their peer process to respond, while others are overloaded with computation and protocol processing overhead. Figure 3.6(b), on the other hand, shows the distribution for SyMMer-enabled MPICH2. As shown in the figure, SyMMer keeps the wait time more even across processes, reducing application overhead without in any way altering the workload assigned to the processes. The benefit is accomplished entirely by changing the mapping of processes to cores.

Out-of-Sync Communication Benchmark

This benchmark emulates the behavior where two application processes perform a large synchronous data transfer and a large computation immediately thereafter. Because some of the user data may be buffered, usually due to a full buffer in a lower level communication layer, the processes may become *out-of-sync*. When the sending and receiving processes are not assigned to cores with equal effective capabilities, resulting in data being buffered at the sender node, we refer to them as *out-of-sync*. The send is delayed and results in the receiver process waiting for not only the amount of time it takes to send the data, but also the time needed to complete the remote computation.

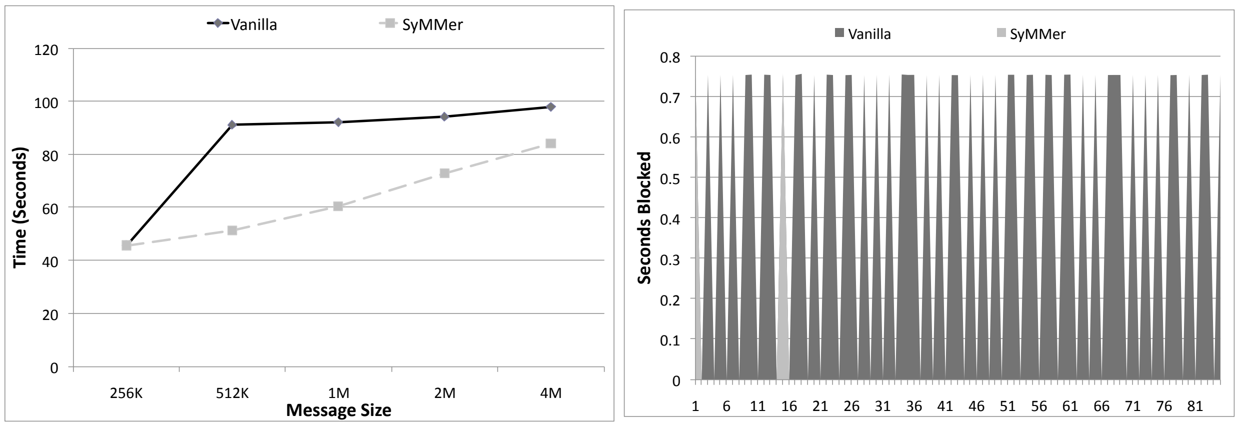


Figure 3.7: Out-of-sync communication benchmark: (a) performance (b) MPI data buffering time

Figure 3.7(a) shows the performance of the out-of-sync communication benchmark using vanilla MPICH2 and SyMMer-enabled MPICH2. It compares the total time to execute the benchmark with various message sizes used in the communication step. Similar to the communication idleness benchmark, SyMMer-enabled MPICH2 performs as well or better than vanilla MPICH2 in all cases. At message sizes up to 256 KB, both vanilla and SyMMer MPICH2 have the same performance, because the socket buffers can handle messages of up to 256 KB without requiring MPICH2 to buffer them. As such, small message sizes prevent out-of-sync behavior, which represents our base case and removes any opportunity for performance improvement. For message sizes above 512 KB, however, we observe that SyMMer-enabled MPICH2 consistently outperforms vanilla MPICH2 by up to 80%. As the message size continues to rise above 512 KB however, the performance gap between SyMMer and vanilla begins to close. MPI resists buffering larger messages whenever possible, diminishing the chance that an out-of-sync event will occur. A detailed analysis of this issue is left for future work.

The internal workings of the benchmark are demonstrated in Figure 3.7(b), which shows the times when data is buffered within the MPI library. As shown in the figure, the data buffering time is almost an order-of-magnitude less when using SyMMer. When an out-of-sync message occurs with SyMMer, the time taken is the same, but since SyMMer corrects the error in synchronization, its occurrence reduces. This reduction ultimately results in the application performance improvement demonstrated in Figure 3.9.

Cache Locality Benchmark

This benchmark stresses the cache locality of processes by performing the majority of the network communication in *certain* processes in the application. Thus, depending on the cores to which the communicating processes are mapped, they may present the cache locality symptom discussed in Section 3.2.2. Hence, when the communicating processes are not on the same processor die as the core performing protocol processing, they can potentially take a severe performance hit.

In our benchmark, processes communicate in pairs wherein two pairs of processes are engaged in heavy inter-node communication, while the other pairs perform computation and exchange data locally. We measure the performance as the time it takes to complete a certain number of iterations of the benchmark. Figure-3.8(a) compares the total execution time with a computational load factor, which is a measure of the amount of work per run. As the computational load factor increases, SyMMer is able to outperform vanilla MPICH2 by up to 29%.

We further analyze the number of L2 cache misses observed by each process. Figure 3.8(b) shows the total number of cache misses observed by the processes performing inter-node and intra-node communication respectively. We observe that the number of cache misses is significantly decreased for the inter-node communicating processes, and despite the migration and other overhead incurred, the cache misses fall for the intra-node communicating processes as well. The intra-node communicating processes gain two benefits from SyMMer that we did not initially anticipate. First, their locality with respect to the local processes with which they communicate improves. Second, moving them away from the die doing the inter-node processing reduces the amount of cache thrashing with which they have to contend. Thus SyMMer is not only able to improve the cache locality of the inter-node communicating processes, but also that of the intra-node communicating processes.

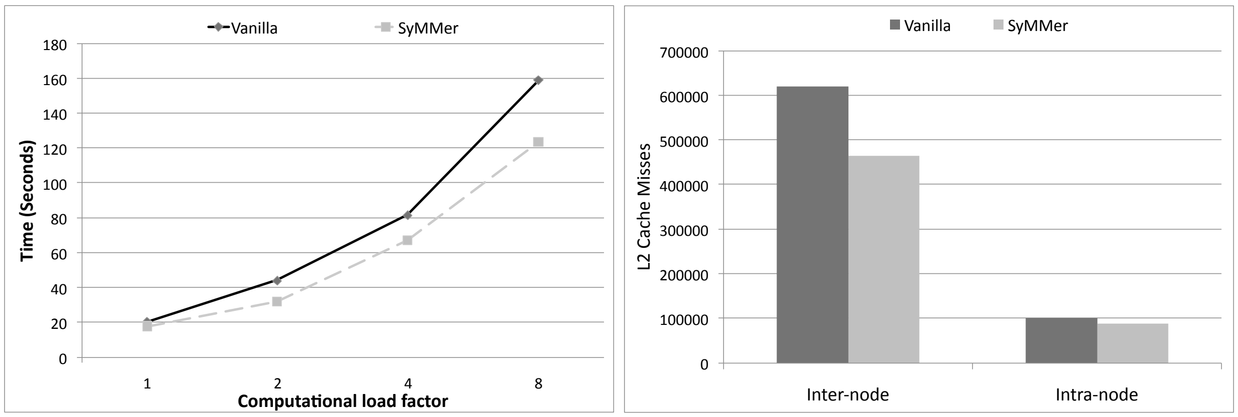


Figure 3.8: (a) Cache locality performance (b) cache miss analysis

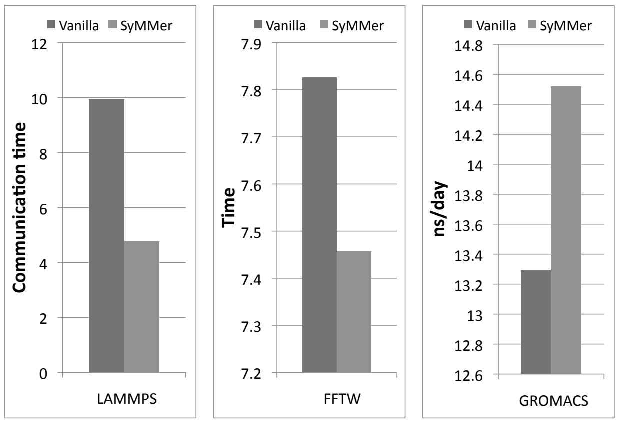


Figure 3.9: Performance evaluation: (A) LAMMPS (B) FFTW, lower is better (C) GROMACS, higher is better

3.4.2 Evaluating Applications and Scientific Libraries

In this section, we evaluate the performance of two molecular dynamics applications, GROMACS and LAMMPS, and the FFTW Fourier Transform library, which we discuss in greater detail in Section 2.7, and demonstrate the performance benefits achievable using SyMMer.

Figure 3.9 illustrates the performance achieved by SyMMer-enabled MPICH2 as compared to vanilla MPICH2 for GROMACS, LAMMPS and FFTW. As shown in the figure, SyMMer-enabled MPICH2 can remap processes to the right cores so as to maximize performance, resulting in a performance improvement across the board. For GROMACS, the overall application execution time is presented, in the form of its own internal performance measure ns/day, which shows an improvement of about 10-15%. For LAMMPS, communication overhead is presented since communication in LAMMPS tends not to scale with problem size making it a more stable measure, which shows nearly a *two-fold* improvement in performance. For FFTW, execution time is presented based on the internal average execution time provided by the benchmark, we noticed only about 3-5% performance difference in our experiments. The small communication volumes that are used for FFTW in our experiments only permit small gains. Given that SyMMer monitors for interactions between the communication protocol stack and the multicore architectures, small data volumes mean that such interaction would be small as well.

In summary, we see a noticeable improvement in performance with SyMMer-enabled MPICH2 for all three cases. Dynamic process-to-core mapping is a promising approach to minimize the impact of interactions of the communication protocol stack with the multicore architecture and allow us to improve application performance significantly in some cases.

3.5 Discussion on Alternative Multicore Architectures

Systems that employ NUMA memory are computationally quite similar to the Intel SMP system that we evaluate in this chapter, and as such one might think they would behave similarly. There are however, a few key differences that need to be addressed before they can utilize SyMMer-like process management libraries to full effect. Specifically, the memory access model that they use complicates process migration as compared to SMP systems. For example, on an SMP system, SyMMer could freely move any process to any core in the system with the only migration cost being re-populating the new cache from main memory. However, on a NUMA system, the memory for a process will have been allocated on its original local memory node. If the process is then migrated to a different die, and thus memory node, all of its memory access become remote, resulting in significant overhead. For example, in an evaluation that we performed with LAMMPS, process migration increased the non-local memory transfers by $17-31\times$ in some cases. While the lessons learned designing SyMMer and its architecture are still valid for NUMA systems, aspects such as page migration become important concerns in achieving good performance. Alternatively, altering the work assigned to a given process, or the location of the memory on which it operates, may be effective alternatives to process migration as means of dealing with asymmetric interactions.

Finally, for Network-on-Chip (NoC) architectures such as Intel Single Chip Cloud processors and Tilera Tile-64 processors, dynamic process-to-core mapping is yet more important; primarily owing to the huge number of cores that reside on the same die. For example, the Intel Single Chip Cloud processor holds 48 cores per die, while Tilera processors contain 64. In such cases, assigning a process to the wrong die, or potentially the wrong core in the on-die fabric, could lead to a significant amount of unnecessary communication overhead. A library such as SyMMer could dynamically *search* for the right assignments and use them to achieve better and more consistent performance.

3.6 Conclusions

In this chapter, we demonstrate that system resource contention and locality can cause cores to present highly distinct *effective capabilities* even in homogeneous systems. We further presented the design and evaluation of a novel systems software stack, known as the *Systems Mapping Manager (SyMMer)* library, which monitors these interactions and dynamically manages the mapping of processes onto processor cores. Our evaluation of the SyMMer library demonstrates nearly a *two-fold* improvement in communication time and 10-15% improvement in overall performance for various applications.

Chapter 4

Adaptive Scheduling with Splitter

4.1 Introduction

In order to address issues along the dimension of capability, this chapter investigates the creation of a heterogeneous task scheduler for Accelerated OpenMP. Targeting heterogeneous systems with existing programming models, such as OpenMP and OpenCL or pthreads and CUDA, currently requires a programmer either to program in at least two different parallel programming models, or to force-fit either the GPU or CPU into a model that wasn't designed for it. Multiple models require code replication, and maintaining two completely distinct implementations of a computational kernel is a difficult and error-prone proposition. That leaves us with using either OpenCL or accelerated OpenMP to complete the task.

OpenCL's greatest strength lies in its broad hardware support. In a way, though, that is also its greatest weakness. In order to target disparate hardware efficiently, the language is very low level, comes with a steep learning curve and many pitfalls related to performance across platforms and devices. OpenCL also suffers from an almost complete lack of bindings for non-C HPC languages such as Fortran. Given an existing OpenCL application, dividing an application across the devices in a system should be simple: divide the inputs and accumulate the outputs. Unfortunately, managing the data transfers, multiple CPU threads, and ensuring that the code functions correctly and runs quickly on different hardware remains a daunting task.

Accelerated OpenMP, in contrast, is designed to allow a user familiar with basic OpenMP programming to port their code to accelerators with relative ease. It offers a more digestible and familiar syntax, especially for Fortran programmers, while remaining capable of significant performance gains. When it comes to using both a CPU and an accelerator together however, the current state-of-the-art implementations offer little support. We propose the addition of new options to accelerated OpenMP, designed to split accelerated regions across available devices automatically.

Our ultimate goal is to enable OpenMP programmers to experiment with coscheduling, combining CPUs with GPUs, without having to re-create the work necessary to split their data and to load balance their computation. This approach requires the compiler and runtime system (1) to split regular OpenMP accelerator loop regions across compute devices and (2) to manage the distribution of inputs and outputs while preserving the semantics of the original region transparently. We investigate the creation of such a runtime system and the requirements to automate the process. Specifically we present a case study that uses a development version of Cray's GPU accelerated OpenMP. For the purpose of this chapter, we use accelerator and GPU interchangeably, although we could apply our approach to any platform that offers similar OpenMP accelerator extensions. We make the following contributions in this chapter:

- Extensions to OpenMP accelerator directives to support co-scheduling;
- Four novel scheduling policies for splitting work across CPUs and accelerators;
- An implementation of those extensions that is built on top of the OpenMP runtime and, thus, applicable to any implementation of the OpenMP accelerator directives;
- An evaluation that demonstrates our extensions significantly improve performance over only using an accelerator.

Our results for four programs that use the OpenMP accelerator directives demonstrates that our approach can produce as much as a two-fold performance improvement over using either the CPU or GPU alone.

The rest of the chapter is arranged as follows. Section 4.2 provides a background in accelerator programming and the OpenMP Accelerator Directives. Section 4.3 describes the design of our proposed heterogeneous scheduling extension. Details of our proof-of-concept implementation follow in Section 4.4. We present results and discussion in Section 4.5.

```
#pragma omp parallel for        \
        shared(in1,in2,out,pow)
for (i=0; i<end; i++){
    out[i] = in1[i]*in2[i];
    pow[i] = pow[i]*pow[i];
}
```

(a) Standard OpenMP

```
#pragma acc_region_loop                  \
        acc_copyin(in1[0:end],in2[0:end])\
        acc_copyout(out[0:end])          \
        acc_copy(pow[0:end])
for (i=0; i<end; i++){
    out[i] = in1[i] * in2[i];
    pow[i] = pow[i]*pow[i];
}
```

(b) Accelerated OpenMP

```
#pragma acc_region_loop                  \
        acc_copyin(in1[0:end],in2[0:end])\
        acc_copyout(out[0:end])          \
        acc_copy(pow[0:end])             \
        hetero(<cond>,                   \
               <iterations for CPU>)
for (i=0; i<end; i++){
    out[i] = in1[i] * in2[i];
    pow[i] = pow[i]*pow[i];
}
```

(c) Accelerated with hetero clause

```
#pragma acc_region_loop                     \
        acc_copyin(in1[0:end],in2[0:end])   \
        acc_copyout(out[0:end])             \
        acc_copy(pow[0:end])                \
        hetero(<cond>[,<scheduler>[,<ratio>\
               [,<div>]]])
for (i=0; i<end; i++){
    out[i] = in1[i] * in2[i];
    pow[i] = pow[i]*pow[i];
}
```

(d) Proposed hetero clause

Figure 4.1: OpenMP accelerator directive comparison

4.2 Background

This section provides specific background material with a focus on two items. First, we review the syntax and form of OpenMP for accelerators as of 2011. Second, we describe the previously proposed version of region splitting and task scheduling for this system. Further relevant background, including general accelerator programming information, is presented in Chapter 2.

The OpenMP accelerator directives are proposed extensions to OpenMP that parallelize OpenMP regions across GPUs or other accelerators. For the purpose of this work, we use the version proposed by James Beyer et al. [19]. Another set of directives with similar goals are available from PGI as the PGI accelerator directives [95]. Although our work builds directly on a working prototype of the former from Cray, our method is generally applicable and should perform similarly with the PGI version.

The extensions add three main concepts to OpenMP: accelerator regions, input/output identifiers, and array shaping. An accelerator region generates a team of threads on an accelerator to process the region, analogous to a parallel region. Input/Output identifiers specify how to transfer data in and out of a region with greater specificity than shared and private. Accelerators, such as GPUs, with on-board discrete memory require explicit memory movement, as discussed in Section 2.3. The identifiers support specification of that movement explicitly. Array shaping specifies the dimensions, range and stride of arrays. These shapes are passed with pointers to the input/output clauses to bound unbounded types in C or to transfer only the necessary part of arrays.

Figure 4.1a shows a loop parallelized for the CPU with OpenMP. Figure 4.1b is the same loop parallelized across a GPU with an `acc_region_loop` directive. We also add the `acc_copy()`, `acc_copyin()` and `acc_copyout()` clauses, which specify that values must be copied in and out, just in, or just out. Each clause accepts a list of variables or shaped arrays of the form `array[<start>:<end>:<stride>]`. These extensions preserve the clarity and syntax of OpenMP while allowing the use of local distributed memory accelerators.

The third code segment in Figure 4.1c includes a clause that was part of the draft standard for the OpenMP accelerator directives. This clause is of the form `hetero(<cond>,<width>)` where cond is a boolean expression, true to split, false to use only the accelerator, and width is the number of iterations to assign to the CPU. It does not provide for scheduling options however, and assumes that the application programmer will explicitly specify the number of loop iterations to run on the CPU, the others to be run on a single accelerator. Further, a more recent draft no longer includes the option. We expect that the option will be useful with some adjustment and increased runtime support; we propose our version in Section 4.3.

4.3 Design

This section presents the abstract design of our proposed system and its schedulers. First we describe the overall structure and then discuss the three classes of schedulers and their overall merits. The first class is `static`, analogous to but distinct from the `hetero` clause. The second supports dynamic scheduling with an intentional deviation from the traditional OpenMP dynamic scheduler inputs. Our third type of scheduler is two special case scheduling policies that combine the static and dynamic policies to handle common behavior of accelerated applications.

4.3.1 Overview

We do not propose to replace part of the existing software stack but rather to add a new one. As Figure 4.2 shows, we intend our work to be a new layer between the OpenMP accelerator directives and the existing CPU and GPU schedulers, which leverages those existing schedulers to handle the details of each device. We focus on assigning tasks to a compute device, which we define as an entity that can be targeted by a `parallel` or `acc` region, i.e., a single GPU or all CPU cores rather than an individual core. Since we work at the region layer, our design applies to any architecture for which an implementation of the accelerator directives is available.

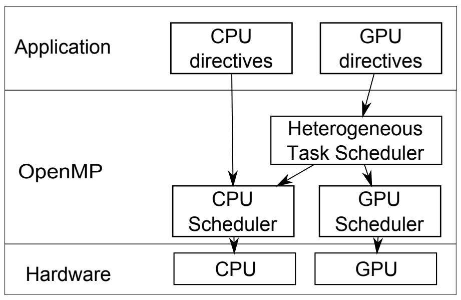


Figure 4.2: Our proposed software stack

Since we target heterogeneous resources, compute devices may have completely disparate performance characteristics. Standard OpenMP schedulers use the size of a chunk to split the work across cores. For example, given a loop of 500,000 iterations one might add `schedule(dynamic,500)` to their parallel loop, which would cause each thread to receive 500 iterations, compute those, and check for another chunk of 500 to compute. However, the optimal chunk size depends on the performance of the underlying devices and the cost to distribute new work to them. Given a CPU based system, chunks of size 500 may perform well, but assigning 500 iterations to an entire GPU will usually take so little time to execute that overhead dominates, wasting potential computation time. Conversely, a chunk large enough for the GPU can run so long on a CPU as to dominate the program execution time before it finishes the first chunk.

Our solution does not use chunks. Instead we specify a ratio that captures the amount of work that a CPU can complete in the time it takes for a GPU to finish 100 units. This schedule is essentially a form of unbalanced static scheduling, like those proposed by Ayguade et al. [14]. For example, if the CPU device (i.e., all CPU cores) completes 100 iterations in the time it takes for the GPU to complete 500 the ratio would be 17%. Alternatively, if the CPU is more suitable for a particular problem and completes 200 iterations in the time that it takes the GPU to handle 100 the ratio would be 67%. Thus, we specify the relationship between the compute units, rather than trying to find a single sensible unit of work to assign to both.

Our scheduler operates at the boundaries of a region rather than within it, except in special cases, much like the DVFS decisions made in Adagio [83], which provided the inspiration for this type of interpass dynamic approach. This choice is another concession to the overhead of GPU kernel launches: by making scheduling decisions only once each pass through a region we generate only one thread team per compute unit rather than having to recreate them repeatedly. It also allows us to synchronize memory at the beginning and end of the region and not between, in turn saving synchronization time. The user expects that all memory is consistent at the end of the region. The most basic example is that the output arrays on the CPU specified by the `acc_copy` and `acc_copyout` must be consistent with the output of running the full problem set on the GPU, but updates must also be pushed to GPU memory, failure to do either can cause unexpected side-effects.

4.3.2 Static Splitting

Static, which is the default scheduler, divides tasks at the beginning of the region. Each entry into the region runs one CPU team and one GPU team, using the underlying static schedulers for each. As noted above, we split based on a ratio. The CPU receives $i_c = i_t * r$ iterations where i_t is the total number of iterations requested and r is the ratio. the GPU receives the remainder $i_g = i_t - i_c$.

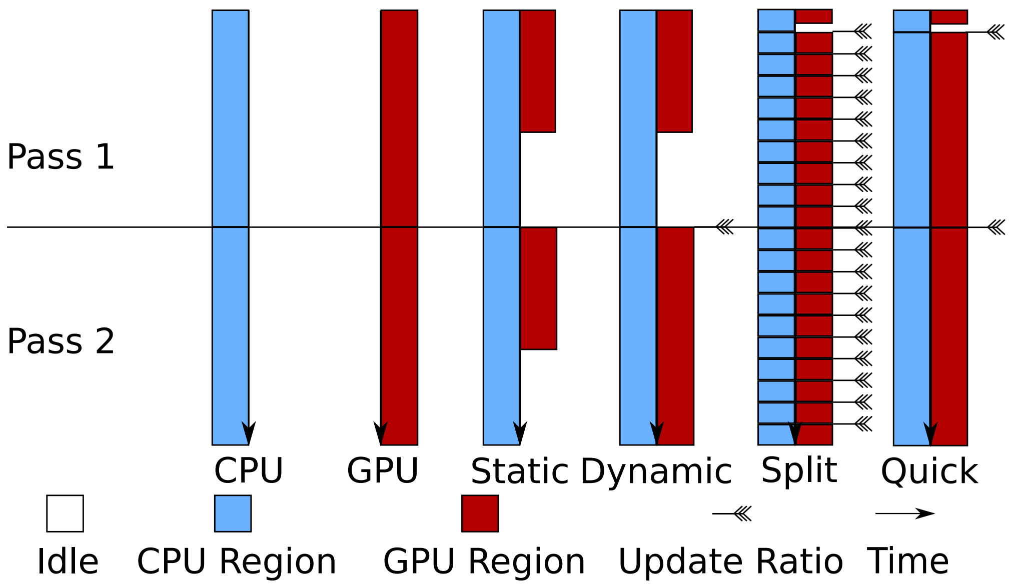


Figure 4.3: Scheduler behavior over time

The ratio argument is optional; its default value is a non-trivial problem. We compute the default ratio at runtime based on the compute resources found to be available, making the assumption that the workload is floating point computation bound. The goal is for the ratio to express the percentage of the total floating point work that the CPU device can perform in a unit of time. Unfortunately, most compute hardware does not expose a software API to query its peak flops directly, so we must approximate based on something more accessible.

We essentially need to know how many floating point operations each compute device can evaluate in a given unit of time. Further, we know that on a current generation GPU, each core can compute one floating point instruction per cycle, for now assuming single precision. The CPU is a more complicated. Each core on a CPU can compute anywhere from one to its SIMD width floating point instructions per cycle. We assume that floating point operations in the region are mostly vectorizable so the CPU can retire its full SIMD width in each cycle, which overestimates the

$I =$	total iterations in next pass (int)
$i_j =$	iterations on compute unit j in next pass (int)
$p_j =$	time/iteration for compute unit j from last pass
$n =$	number of compute devices
$t_j^+ =$	time over equal
$t_j^- =$	time under equal

Table 4.1: Variable definitions for the linear program

CPU somewhat. This overestimation helps to balance another assumption: both the CPU and GPU operate on the same frequency. The final equation is $r = c_c * 4/(c_c + c_g)$ where c_c is the number of CPU cores and c_g is the number of GPU cores. This default is portable since we can detect the compute resources available on any given system and adjust to them. We find that this simple model performs well for compute-bound floating point intensive applications, but not for memory bound ones, or highly conditional applications, as we discuss further in Section 4.5.

4.3.3 Dynamic Splitting

Similarly to our static scheduler, our dynamic scheduler deviates from the original OpenMP dynamic schedule policy. We make scheduling decisions only at the boundaries of accelerated regions. Thus, the dynamic scheduler assumes that the code will execute the parallel region several times. The first time, our approach executes the region as the static scheduler would. We measure the time taken to complete the assigned iterations on each compute unit. On all subsequent instances of the parallel region, we update the ratio based on these measurements.

Since we split at region boundaries rather than using a queue, we are subject to blocking time, during which one compute unit is waiting on the other to finish before they can pass the barrier at

$$min(\sum_{j=1}^{n-1} t_j^+ + t_j^-) \tag{4.1}$$

$$\sum_{j=1}^{n} i_j = I \tag{4.2}$$

$$i_2 * p_2 - i_1 * p_1 = t_1^+ - t_1^- \tag{4.3}$$

$$i_3 * p_3 - i_1 * p_1 = t_2^+ - t_2^- \tag{4.4}$$

$$\vdots$$

$$i_n * p_n - i_1 * p_1 = t_{n-1}^+ - t_{n-1}^- \tag{4.5}$$

Figure 4.4: Mixed-integer linear program to minimize runtime deviation

the end of the region. In order to minimize blocking time, we attempt to compute the ratio that causes the compute units to finish in as close to the same amount of time as possible. In order to predict the time for the next iteration, we assume that iterations take the same amount of time on average from one pass to the next. For the general case with an arbitrary number of compute units, we use a linear program for which Table 4.1 lists the necessary variables. Figure 4.4 presents our linear program with Equation 4.1 representing the objective function and the accompanying constraints in Equations 4.2 through 4.5.

Expressed in words, the linear program calculates the iteration counts with predicted times that are as close as possible to identical between all devices. The constraints specify that the sum of all assigned iterations must equal the total number of iterations and that all iteration counts must be integers. Since we often have exactly two compute devices, we also use a reduced form that is only accurate for two devices but can be solved more efficiently. The new ratio is computed such that $t'_c = t'_g$ where t'_c is the predicted new time for the CPU portion to finish and t'_g is the predicted time

$$\begin{aligned}
i'_c * p_c &= i'_g * p_g \\
i'_c * p_c &= (i'_t - i'_c) * p_g \\
i'_c &= ((i'_t - i'_c) * p_g)/p_c \\
i'_c &= ((i'_t - i'_c) * p_g)/p_c \\
i'_c + (i'_c * p_g)/p_c &= (i'_t * p_g)/p_c \\
(i'_c * p_c)/p_c + (i'_c * p_g)/p_c &= (i'_t * p_g)/p_c \\
i'_c * (p_g + p_c) &= i_t * p_g \\
i'_c &= (i_t * p_g)/(p_g + p_c) \qquad (4.6)
\end{aligned}$$

Figure 4.5: Reduced-form equation for split calculation on two devices

to finish the GPU portion. When expanded we eventually get Equation 4.6 in Figure 4.5, which can be solved in only a few instructions and produces a result within one iteration of the linear program for the common case.

4.3.4 Special Purpose

Our design and testing indicate that while our linear program is effective at deciding accurate distributions of work, it is not always appropriate to schedule only at the boundaries of a region. Thus, we created two specializations of the adaptive scheduler that offer different timing properties: *split* and *quick*.

Split

Our dynamic scheduling requires multiple executions of the parallel region to compute an improved ratio, which works well for applications that make multiple passes. However, some parallel regions are executed only a few times, or even just once. Split scheduling addresses these regions. Each pass through the region begins a loop that iterates *div* times with each iteration executing $totaltasks/div$ tasks. Thus, the runtime can adjust the ratio more frequently, and earlier, than with dynamic scheduling. More importantly, it can adjust the ratio in cases that dynamic scheduling cannot. The split schedule is analogous to the original OpenMP dynamic schedule since it specifies $totaliterations/chunksize$ instead of chunk size directly. It remains distinct however in that while it runs a number of chunks, they can be independently subdivided to avoid overloading, or underloading, a compute device. Increasing the number of passes however, and thus the number of synchronization steps, increases overhead, which is especially problematic with short regions and those unsuitable for GPU computation so it is unsuitable as a definitive replacement for dynamic.

Quick

Quick is a hybrid of the split and dynamic schedules. It executes a small section of the first pass of size $iterations/div$ just as split does, but the rest of that pass in one step of size $iterations - iterations/div$. It then switches to using the dynamic schedule for the rest of the run. It targets efficiently scheduling of applications with long running parallel regions that can be dominated by the first pass of the dynamic schedule when given a poorly chosen initial ratio. Quick is especially useful when such regions are executed repeatedly, making the split scheduler impractical due to its added overhead.

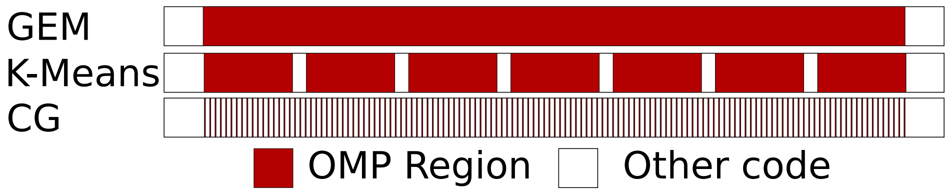


Figure 4.6: Computation patterns of evaluated benchmarks

4.3.5 Schedules

We have alluded to the types of application that each schedule targets. Figure 4.6 shows the computational patterns, in terms of OpenMP regions, of three applications that we evaluate in Section 4.5. Each of our three dynamic schedulers targets one of these three cases, not just for these applications but as a general pattern of use. The first application type, with its single huge region, is a clear choice for the split schedule. The quick schedule targets the second, which has slightly smaller sections in which the segments are too long to allow an entire pass with a bad static mapping but do not require splitting every region to achieve load balance. Finally, we have applications that use fine-grained regions. Any overhead dominates these regions, which do best with either quick or dynamic. Of course, we could use static for any of these cases as well, especially if we want to fine-tune the ratio manually.

4.4 Implementation

We implement our concurrent heterogeneous support as a library that uses the OpenMP accelerator directives to motivate its addition in lower levels. This library encapsulates our scheduling functionality. We manually translate applications with minimal effort to function as if our proposed clause and schedules were used. Our implementation would be easy to integrate into a compiler, which represents the design in Section 4.3. In the only significant difference from our design, our implementation currently only supports two devices at a time since our testing environment only offers two. We will implement the general case in future work. In addition to the implementation of the library, we also investigate what such loop splitting requires without underlying support.

4.4.1 The Splitter Library

In order to keep the implementation as general as possible, we design the library to be independent of the implementation of accelerated OpenMP. The library does not use accelerated regions or accelerated OpenMP functions or constructs, with the single exception of `omp_get_thread_limit()`, which we use to determine the number of available threads and to calculate the default static ratio. In the current version, we read the number of GPU cores from an environment variable, or assume it is 448, which is the number of cores in the NVIDIA Tesla C2050 GPU. While we would prefer to read the value from the underlying system, the OpenMP accelerator extensions do not include a function for this purpose.

The interface has six functions and a structure, as Figure 4.7 shows. The `split_init()` function initializes the library for a new region. It takes the arguments that would be given to the `hetero()` clause as well as the number of tasks to expect and it returns a structure to use with the splitter functions. After that, `split_next()` is evaluated at least once, populating the struc-

```
splitter * split_init(int size, split_type sched,
                      double *rat,int *div)
splitter * split_next(splitter * s, int size,
                      int iteration)
void       split_cpu_start(splitter *s);
void       split_cpu_end(splitter *s);
void       split_gpu_start(splitter *s);
void       split_gpu_end(splitter *s);
typedef struct splitter{
    int cts;   //CPU start iteration
    int cte;   //CPU end iteration
    int gts;   //GPU start iteration
    int gte;   //GPU end iteration
    int d_end; //div
    int d_ccs; //start of CPU output
    int d_cce; //end of CPU output
    int d_gcs; //start of GPU output
    int d_gce; //end of GPU output
}splitter;
```

Figure 4.7: Splitter API for basic, CPU and single GPU, case

ture with the assignments for each device. Each pass through the region, `split_next()` restarts these counters, unless invoked with monotonically increasing iteration values, which is used to implement the split scheduler as we discuss shortly. The other four functions are timing calls that inform the library of the beginning and end of each split region.

In order to avoid repeated data transfers to and from the GPU in a pass, the library sends the entire data set for the region to the GPU and retrieves the entire output whether or not it is all used. Although this choice is inefficient, it is more efficient than copying piecemeal as the split between CPU and GPU is adjusted. Lower level APIs in future could make this choice unnecessary. Because we copy back the entire region, we also must use a temporary array to receive either the output from the CPU or GPU, and merge that into the main output array after both have finished. Otherwise consistency could not be assured. Use of this array could also be avoided at a lower level in future work.

```
splitter * s = split_init(no, SPLIT_DYNAMIC, NULL, NULL);
int *m_c = (int*)malloc(sizeof(int)*no);
for(int d_it=0; d_it < s->d_end; d_it++)
{
    s = split_next(no, d_it);

#pragma omp parallel num_threads(2)
    {
        if(omp_get_thread_num()>0)
        {//CPU OpenMP code
            split_cpu_start(s);
#pragma omp parallel shared(fo,fc,m_c,s)              \
                num_threads(omp_get_thread_limit()-1) \
                firstprivate(no,ncl,nco) private(i)
            {
                #pragma omp for
                for (i=s->cts; i<s->cte; i++) {
                    m_c[i] = findc(no,ncl,nco,fo,fc,i);
                }
            }
            split_cpu_end(s);
        }else{//GPU OpenMP code
            split_gpu_start(s);
            int gts = s->gts, gte = s->gte;
#pragma omp acc_region_loop private(i)              \
                    firstprivate(nco,no,ncl,gts,gte)\
                    acc_copyin(fc[0:ncl*nco])       \
                    acc_copyout(m[0:no])            \
                    present(fo)     default(none)
            for (i=gts; i<gte; i++) {
                m[i] = findc(no,ncl,nco,fo,fc,i);
            }
            split_gpu_end(s);
        }
    }
}
memcpy(m+s->d_ccs,m_c+s->d_ccs,
       (s->d_cce-s->d_ccs)*sizeof(int));
free(m_c);
```

Figure 4.8: Manually transformed k-means kernel

Since we must merge the data, we attempt to merge as efficiently as possible. We accomplish this goal by having each compute device work from opposite ends of the iteration space toward the center. Thus, we divide the output only into two pieces regardless of any adjustments made during the run, unlike a simpler implementation that assigns chunks moving from one end to the other.

4.4.2 Using Splitter

```
#pragma omp acc_region_loop private(i)           \
        firstprivate(nco,no,ncl) default(none)   \
        acc_copyin(fc[0:ncl*nco]) present(fo)    \
        acc_copyout(m[0:no])
//      hetero(1,dynamic)
for (i=0; i<no; i++) {
    m[i]  = findc(no,ncl,nco,fo,fc,i);
}
```

Figure 4.9: Accelerated OpenMP k-means region

We present an example to illustrate the use of the library. The code in Figure 4.8 shows the manually translated version of the code of a k-means kernel in Figure 4.9, which includes a `hetero` clause so it would correspond to the manually transformed version. This kernel is part of the code for the k-means implementation that we evaluate in Section 4.5.

We divide the translated code in two ways. First, we divide the region into a CPU region and a GPU region, each in one branch of a conditional, with each given one thread from an outer parallel region. Thus, the master thread runs the GPU region, a requirement as all other threads crash when encountering GPU regions in this version of the compiler, and the other thread starts a new team that uses the remaining compute resources on the CPU. We transform the computational loop in each of the two regions to use the start and end values specified by the splitter library, and bound it on either side by calls used for timing.

The other way in which we split the code is through the outer loop. That loop allows the scheduler to split the region into multiple sub-regions, by specifying how many iterations it will pass, and thus how many times `split_next()` will be evaluated. At the end of the outer loop, the merging step is a simple `memcpy()` of the CPU calculated values from the extra array to the original output array.

While the manual transformation is conceptually simple, it is verbose. The original kernel is only 12 lines of code, including line wrapping, while the transformed version is 43. It also has two copies of the inner loop, forcing any change made in one to be replicated. For both of these reasons, even using a library such as ours to split a region at this level is tedious and error prone. Even so, being able to use all the compute resources available in a system is worthwhile.

4.5 Evaluation

This section presents an evaluation of our proof-of-concept runtime library. All codes were compiled with a development version of the Cray Compiling Environment (CCE) compiler version 8.0 using optimization flag -O3. All tests were run on a single node containing a 2.2Ghz 12-core AMD Opteron 6174 processor and one NVIDIA Tesla C2050 GPU. No other processes, aside from the standard daemons, were allowed to run during the tests. All CPU results, unless otherwise specified, use all 12 available cores, in the case of runs using both the CPU and the GPU concurrently one core is reserved to manage the GPU, the other eleven are assigned to computing. Parameters to the scheduler are defaults unless otherwise specified; ratio is as defined in Section 4.3.2 and div is 10.

To evaluate our proof-of-concept, we have implemented OpenMP Accelerator directives versions of four applications, each of which is discussed in greater detail in Section 2.7, including: GEM [8, 41, 44], k-means, CG [15] and Helmholtz. Each of the four presents a different pattern of execution, and different levels of fitness for GPU computing. In each case, the minimum transformation necessary to accelerate the code was used. In GEM, k-means, and Helmholtz the transformation entailed exactly two lines of code. CG required more changes because of some undefined behavior when mixing regular OpenMP threads with GPU regions in Fortran, which we solved by porting the computational kernel of CG to C before accelerating it. Optimizations could certainly be applied, and the performance of each benchmark would benefit, but we are evaluating the framework, not the specific benchmark, and so leave this for future work. In addition to the OpenMP accelerator directive modifications, we applied the transformations necessary to hook into our region splitting library as we described in Section 4.4

For each application we collect computation time, as defined by the time to compute a result excluding problem setup and I/O. All transfers to and from the GPU are included, as is scheduling time and extra work necessary to split and to reassemble data to preserve the memory semantics of the region. In addition to computation time, we collect the number of iterations scheduled on the CPU and GPU on each pass through the accelerated region, as well as the amount of time that each spent running those iterations. From this we calculate the blocking time on a given pass as the time one compute unit must wait for another to finish, or $max(time_{gpu}, time_{cpu}) - min(time_{gpu}, time_{cpu})$. Finally, we track the application's performance ratio, as we defined in Section 4.3.2.

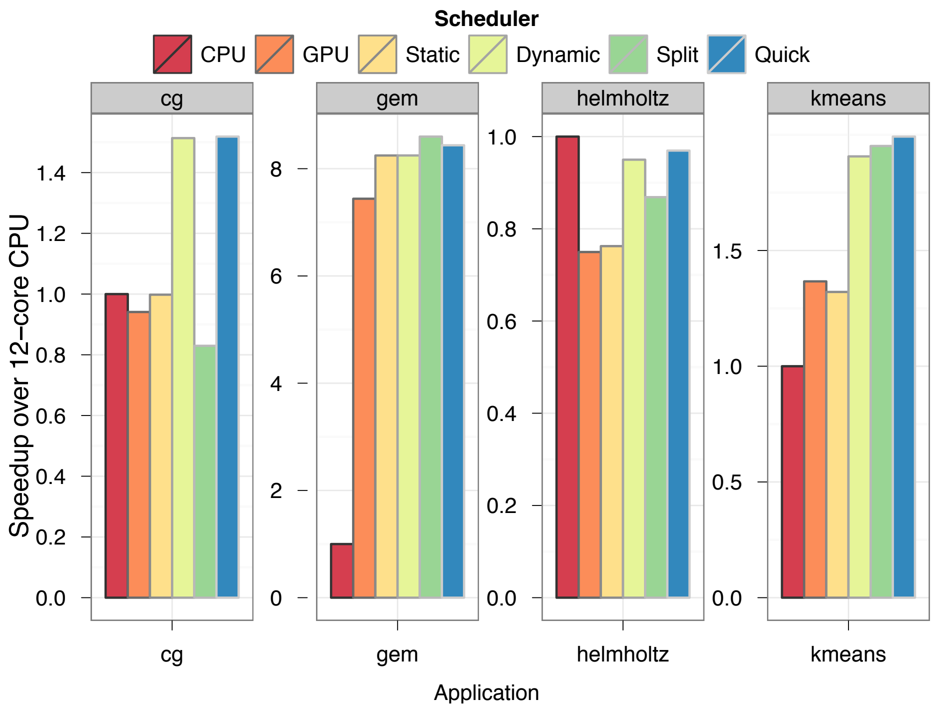


Figure 4.10: Speedup over 12-core CPU for each application across schedulers

4.5.1 K-Means

K-Means is a popular clustering algorithm that uses an iterative method. Each iteration has two stages; the first calculates the nearest cluster for all data points and the second moves each cluster to the center of the data points that identified it as nearest earlier that time step. As a converging algorithm, it does not have a set number of passes as an application, but a given problem does. In our case, the number of passes is generally low in relation to the kernel execution time, but varying the dataset can vary these parameters. The dataset used for our tests consists of 1,210,000 points each with nine observations and groups those points into 500 clusters, requiring seven iterations to converge to a solution.

Figure 4.10 shows the results for each of our benchmarks, with our k-means test on the far right-hand side. For this input set and implementation the CPU and GPU are relatively evenly matched. K-means, is bound by one of two operations, memory accesses as part of computing the distance to every cluster, or the conditionals that check whether that is the nearest cluster. Consequently, it is not compute bound on the GPU, and the default ratio of 0.098, which effectively gives the CPU 9.8% of the work, is far off from its native ratio of 0.396. Thus, the default assigns excess work to the GPU and the static scheduler performs particularly poorly. The dynamic scheduler performs better, but suffers from the first pass being run entirely with that significantly poor ratio. Split, despite the added overhead, outperforms dynamic by reducing the time that we run at the default ratio. This observation motivated our quick scheduler, which quickly makes the first adjustment while reducing overhead for the remainder of the run. In fact, quick produces an extra 10% performance improvement over split for this case.

K-means is unique among our four benchmarks in that its pattern of computation that Figure 4.6 shows can change significantly based on the input dataset. To study the effect of this change on the scheduler, we ran a variety of data sizes in Figure 4.11. These tests use varying size subsets of a 5 million point dataset with 10 observations per point and groups them into 100 clusters. The number of passes necessary for convergence, and length of a pass both vary as the data size varies. Despite the variability in the number and nature of the passes, the three dynamic schedulers show consistent behavior across all five reference sizes, although as the number of iterations for convergence grows, peaking in the three million point dataset, the split scheduler gets progressively worse, leaving the dynamic and quick schedulers as the clear options for k-means with quick being the most consistently fast across all tests.

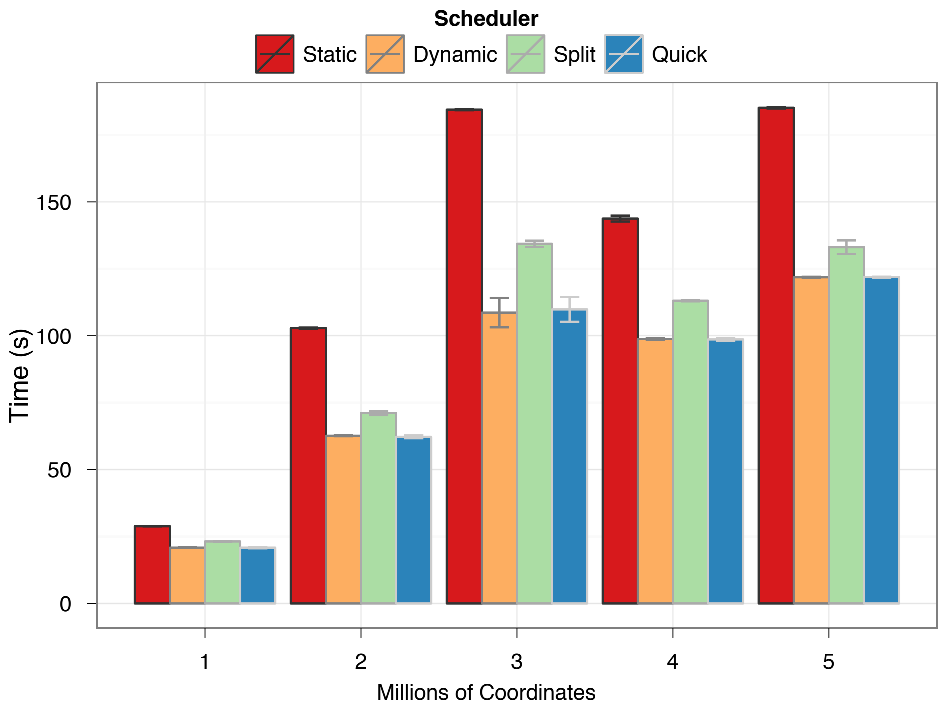


Figure 4.11: K-means performance on data sets in one million point increments

4.5.2 CG

CG is the NAS Parallel Benchmarks implementation of the conjugate gradient method. This method is used to solve systems of linear equations by iterative refinement. While CG requires a relatively small number of steps to converge, each full iteration consists of a set of smaller steps internally. Thus, the number of passes of the accelerated region is large, and each pass is short. We use the C class dataset for all of our tests, which requires 75 iterations to converge, and runs 1900 passes through the accelerated region. Figure 4.10 shows that, switching to the GPU alone does not speed up CG over using the 12 CPU cores. This result is expected since the data copying and kernel launch overhead inherent in the simple acceleration are significant. Even so, it almost matches 12 CPU cores, which in terms of performance per dollar is still worth it if both can be used together. The static scheduler sends insufficient work to the CPU, much as it does with k-means, although in this case, the performance is slightly improved regardless. Due to the very small pass size, the added overhead for split completely destroys the performance of the application, and the small extra overhead and early mis-prediction in quick cause it to be, on average, slightly lower performing than the dynamic scheduler.

4.5.3 GEM

GEM is a molecular modeling code developed by Fenley et al. [41] to study the electrostatic potential along the surface of biomolecules. It is a single step non-bonded force interaction problem that has a computational complexity of $a * v$ where a is the number of atoms in the biomolecule and v is the number of vertices, or points in the surface grid, to be computed. Since GEM is a single step problem, it runs large data sets through a single iteration of one region, as Figure 4.6 shows. For all GEM tests, we use the 2eu1 input, which consists of 109,802 atoms and 898,584 vertices.

The first striking feature of the GEM data that Figure 4.10 shows is that the GPU is significantly faster, about $7.5\times$, than all 12 CPU cores. This gain is consistent with prior studies [8]. However, including the 12 CPU cores, which are normally left idle, improves performance by approximately 10% over the GPU version. Also, the static scheduler performs well for GEM. The default ratio balances floating point performance and, thus, produces favorable results for floating point computation bound applications like GEM.

This application was the original impetus for the development of the split scheduler. Since it has

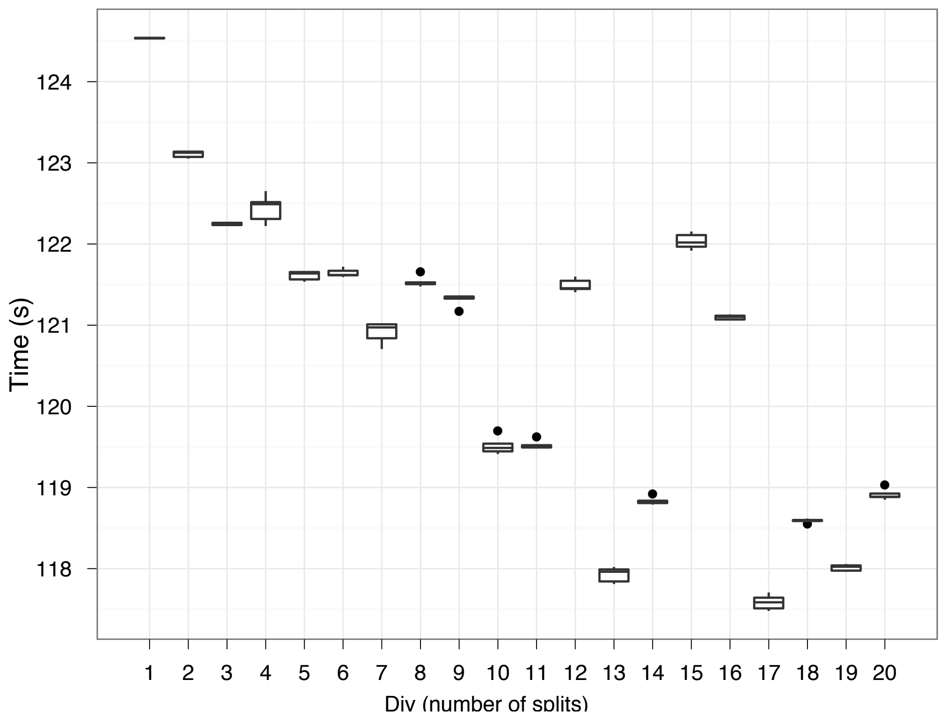


Figure 4.12: GEM performance at varying divs

only one very long running region, as Figure 4.6 depicts, the standard dynamic scheduler has no opportunity to adjust the ratio during the course of a run. We expected that the split scheduler would produce an improvement, but that the quick scheduler would provide even greater benefit due to its lower overhead and reduced synchronization. However, the split scheduler performs best, possibly because the quick scheduler runs a small chunk of a data and then extrapolates to the rest of the dataset, which could make it overly sensitive to overhead in thread team creation and kernel launching. In other applications, dynamic adjusts for to this cost after the second pass to hide it. However, GEM's single iteration provides no chance to fix the imbalance.

We also vary div with GEM to characterize its optimal split size. Figure 4.12 shows that performance increases until a div of ten. Three specific larger divs cause performance degradation.

We initially suspected that these tests exhibited system noise, but the results are repeatable. The degradations may be due to sub-optimal kernel launch decisions made by the underlying system similar to what happens when a poor block size decision is made directly in CUDA. In any event, the optimal choice for div is non-trivial, and presents an opportunity for future tuning.

4.5.4 Helmholtz

Helmholtz implements a solver for a discrete finite difference version of the Helmholtz equation, using the Jacobi iterative method. The implementation is similar in design to a map reduce in two dimensions that calculates the equation at all $n * m$ points in the space and reduces on the error residuals to test for convergence. The application has a single OpenMP region with a reduce clause, so the map and reduce are combined into one phase. The results for Helmholtz in Figure 4.10 are materially different from our other applications. Communication overhead completely dominates the computation whenever a region is offloaded to the GPU. Even with a significant problem size, none of the GPU enabled versions could keep up with using only the CPU. The dynamic schedulers detect the issue and effectively turn off the GPU by not sending it any work after the first few iterations – an average of 5 iterations. Table 4.2 shows the number of outer-loop iterations assigned to CPU and GPU in the first six iterations of a run with the dynamic scheduler. With the advent of accelerators that require little to no copying overhead, such as AMD Fusion, we expect that the accelerator could compete with the CPU, and thus it may be useful to enable this feature even for this application when they become available. We leave optimizing the number of iterations for the scheduler to determine that the GPU is detrimental for future work.

Device/pass	1	2	3	4	5	6
CPU	392	2793	3637	3891	4000	4000
GPU	3608	1207	363	109	0	0

Table 4.2: Outer-loop iterations assigned to CPU and GPU per pass in helmholtz

4.5.5 Overall

The optimal scheduler depends on the application and, in some cases, the size of the dataset. K-means benefits most from the quick scheduler, GEM the split, and CG the basic dynamic. Since each scheduler benefits at least one pattern, these four schedulers offer a good starting point for dynamic task splitting in heterogeneous systems. Alternatively, a user could compute the ratio for a given application or save the ratio that a dynamic scheduler computes to derive a low overhead static schedule that achieves accurate splits as Figure 4.13 shows. In a case with regions small enough to be highly sensitive to overhead, as in the case of CG, tuned static scheduling can be highly effective. However, for GEM and k-means, the dynamic policy performs better, implying that the ratio is adjusted during the run based on current conditions to produce a better ratio. In addition to the ability to compute a better scheduler at runtime, computing the ratios for all input data sets and applications, not to mention hardware platforms, will not always be feasible, so we foresee a continued need for runtime dynamic splitting.

In addition to the overall performance of each application, we collect the length of time spent blocking waiting on either accelerator. As we discussed in Section 4.3, our dynamic scheduler, and those that build on it, assign iterations in order to minimize blocking time. Figure 4.14 presents the total blocking time in each application scheduler combination. Overall these results follow

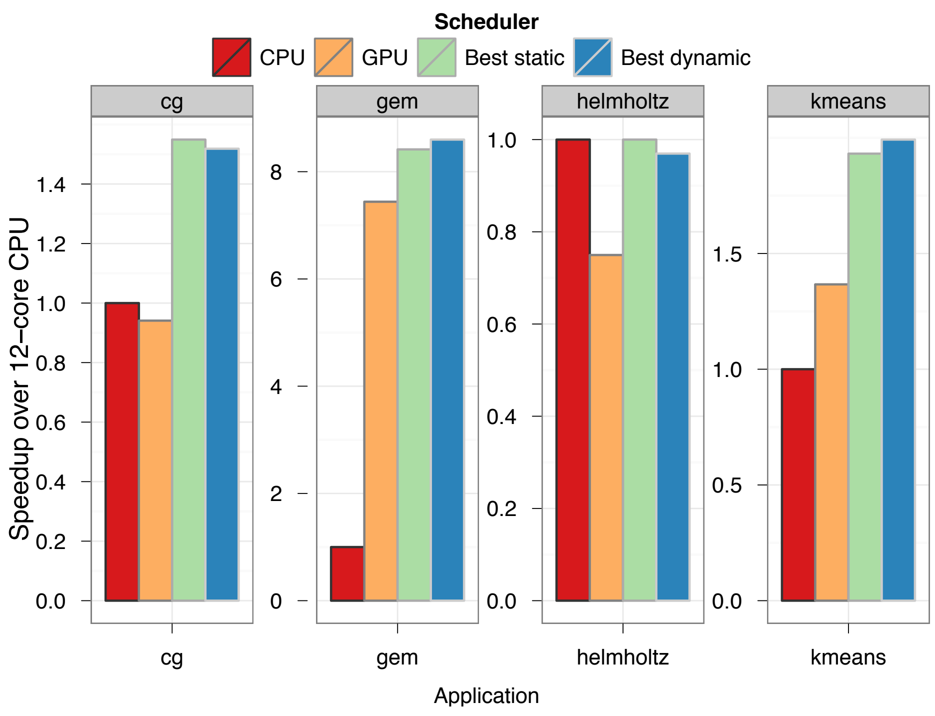


Figure 4.13: Speedup of best static and dynamic options over 12-core CPU

our expectations: the blocking time usually corresponds to overall execution time behavior. CG presents a notable exception in that the static scheduler, while not performing well, performs significantly better than the split scheduler. However the blocking time of the split scheduler makes it appear more efficient possibly due to an increase in the percentage of the accelerated region that transfers data and performs synchronization. As the regions become smaller, that overhead begins to dominate the execution time, slowing computation despite a balanced workload, as seen in the time per iteration on each device under the two schedulers. Under static, iterations average $0.39\mu s$ on the CPU and $0.26\mu s$ on the GPU. Under split with the default div of 10, iterations on the CPU average approximately the same time ($0.37\mu s$) but GPU iterations take much longer ($1.12\mu s$). The extra overhead increases the per iteration runtime by more than $4\times$. Thus, alternative metrics for

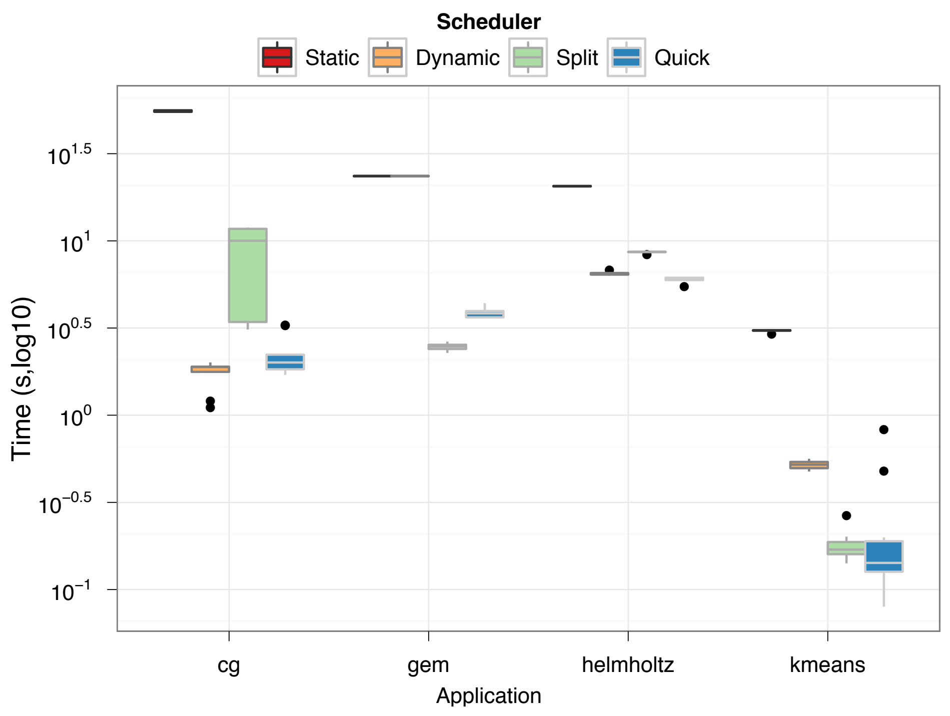


Figure 4.14: Total blocking time observed with each scheduler for all applications

the dynamic schedule may produce better results for this application.

4.6 Conclusions

In this chapter we have presented our design for an automatic heterogeneous task scheduler for Accelerated OpenMP. We make four major contributions: the design of the extension; four scheduling policies to handle a variety of application behaviors; our case study implementation in the splitter library; and our evaluation across four representative scientific codes. Despite certain drawbacks inherent in our library implementation approach, we have shown speedups of as much as $2\times$ over using the CPU or GPU alone. We clearly demonstrate the utility of a `hetero()` clause in Accelerated OpenMP.

Chapter 5

Task Adaptation with CoreTSAR

5.1 Introduction

This chapter continues our exploration of task scheduling and adaptation in response to heterogeneous hardware capabilities. The goal is to improve programmability, performance and performance portability of codes on these systems by creating a safe and efficient means of worksharing parallel loop regions across disparate devices. Specifically, we design an interface for worksharing across multiple address spaces, and thus arbitrary combinations of CPUs and GPUs, which may be automated by inclusion into Accelerated OpenMP. We call this interface CoreTSAR, and it is the continuation of our work on the Splitter library in Chapter 4.

While heterogeneous systems are becoming more popular, their programming models deter many potential users. Unlike adding more or faster CPUs, which work without code changes, programs must be explicitly updated to use GPUs and other accelerators. Rather than grapple with unfamiliar programming models, users often run their CPU-only code on accelerated resources, leaving a significant portion of the computing resources idle. Accelerated OpenMP [19], our term for a class of directive-based programming models including OpenMP for Accelerators [19] and the PGI accelerator model [95] among others, can ease this transition by allowing users to target accelerators with familiar OpenMP-style syntax. However, Accelerated OpenMP is not a panacea: current iterations help one *move* their computation to a *single* accelerator with straightforward syntax. Once so moved however, there is no way to work-share a loop across multiple devices without manually targeting each device.

In order to target, for example, a GPU and four CPU cores, a user must manually split their work, run that work on each separate device, and manually merge each result. Any load-balancing, coherency or runtime adaptation of any kind must be reimplemented by every user. Thus, while Accelerated OpenMP provides one of the greatest strengths of OpenMP, parallelizing serial code through annotation, it lacks another, the ability to scale and to load-balance work transparently on the hardware found at runtime.

Our work seeks to allow safe and efficient worksharing across devices in Accelerated OpenMP. We must address two primary concerns to offer such a construct. First, we must manage memory input and output across multiple address spaces without requiring alterations to the associated parallel loop. Second, we must divide work across devices with vastly different computational capabilities fairly and efficiently. The Task-Size Adapting Runtime (CoreTSAR) library automates the scheduling, load balancing, and cross-device data management necessary to implement this worksharing construct. This chapter presents the design and implementation of CoreTSAR and extended Accelerated OpenMP syntax to integrate its functionality. Specifically we make these contributions:

- Design and syntax of a multi-target worksharing construct for Accelerated OpenMP;
- The design, implementation and optimization of our scheduling and memory management library, CoreTSAR, that can be used with any Accelerated OpenMP compiler/runtime or with CUDA and CPU OpenMP directly;
- Seven adaptive scheduling policies spanning both a low-overhead but coarse-grained adaptive approach and a chunk-based fine-grain scheduling approach to distributing work.
- An evaluation across 19 benchmarks that demonstrates runtime scheduling can significantly improve performance while maintaining programmability.

The chapter is composed as follows. Section 5.2 offers motivation and background information. Section 5.3 describes the design of CoreTSAR including our task management concept, scheduling mechanisms, and memory management. Details on our implementation follow in Section 5.4. Section 5.5 present our results. Conclusions follow in Section 5.6.

5.2 Background and Motivation

As heterogeneous systems spread through the marketplace, so do programming models that target them. While a variety exist, most appear to fit into three categories, each of which is discussed in

```
void runGemm(T **a_a, T **b_a, T **c_a) {
  T *a = *a_a, *b = *b_a, *c = *c_a;
//OpenMP
#pragma omp parallel for
//Accelerated OpenMP
#pragma acc region for copy(c[0:N*N])   \
        copyin(a[0:N*N],b[0:N*N])
//Accelerated OpenMP + extension
#pragma acc region for part_copy(c[1:N][0:N])\
        copyin(b[0:N*N]) part_copyin(a[1:N][0:N])\
        hetero(1, all, adaptive)
  for (int i = 0; i < N; ++i) {
    for (int j = 0; j < N; ++j) {
      c[(i*N) + j] *= B;
      for (int k = 0; k < N; ++k) {
        c[(i*N)+j] += A * a[(i*N)+k] * b[(k*N)+j];
} } } }
```

```
__global__ void
cudag(T *a, T *b, T *c, T A, T B, int n) {
  uint i = blockIdx.x * blockDim.x + threadIdx.x;
  if(i < n) {
    for (int j = 0; j < n; ++j) {
      c[(i*N) + j] *= B;
      for (int k = 0; k < n; ++k) {
        c[(i*N)+j] += A * a[(i*N)+k] * b[(k*N)+j];
} } } }
void runGemm(T **a, T **b, T **c) {
  T *ca, *cb, *cc; dim3 dB, dG;
  size_t size = N*N*sizeof(T);
  dB.x = 64; dB.y = dB.z = 1;
  dG.x = (N/dB.x)+1; dG.y = dG.z = 1;
  cudaMalloc(&ca, size);
  cudaMalloc(&cb, size);
  cudaMalloc(&cc, size);
  cudaMemcpy(ca,*a,size,cudaMemcpyHostToDevice);
  cudaMemcpy(cb,*b,size,cudaMemcpyHostToDevice);
  cudaMemcpy(cc,*c,size,cudaMemcpyHostToDevice);
  cudag<<<dG,dB>>>(a, b, c, A, B, N);
  cudaMemcpy(*c,cc,size,cudaMemcpyDeviceToHost);
}
```

```
#pragma omp target device(smp,cuda)
void gemm(T *a, T *b, T *c, int i, T A, T B, int n);

#pragma omp target device(smp) copy_deps
#pragma omp task input ([n] a, [n*n] b) inout ([n] c)
void gemm(T *a, T *b, T *c, int i, T A, T B, int n) {
  for (int j = 0; j < n; ++j) {
    c[j] *= B;
    for (int k = 0; k < n; ++k) {
      c[j] += A * a[k] * b[(k*n)+j];
} } }

#pragma omp target device(cuda)
__global__ void
cudag(T *a, T *b, T *c, int i, T A, T B, int n)
{ unsigned int j = blockIdx.x * blockDim.x + threadIdx.x;
  if(j < n) {
    c[j] *= B;
    for (int k = 0; k < n; ++k) {
      c[j] += A * a[k] * b[(k*n)+j];
} } }

#pragma omp target device(cuda) copy_deps implements(gemm)
#pragma omp task input ([n] a, [n*n] b) inout ([n] c)
void gemm_gpu_wrap(T *a, T *b, T *c, int i, T A, T B, int n)
{
  __global__ void
  cudag(T *a, T *b, T *c, int i, T A, T B, int n);

  dim3 dB, dG;
  dB.x = 64; dB.y = dB.z = 1;
  dG.x = (n/dB.x)+1; dG.y = dG.z = 1;
  cudag<<<dG,dB>>>(a, b, c, i, A, B, n);
}

void runGemm(T **a, T **b, T **c) {
  for (int i = 0; i < N; ++i) {
    gemm(a[i], b[0], c[i], i, A, B, N);
  }
#pragma omp taskwait
}
```

Figure 5.1: A basic GEMM kernel as implemented in OpenMP variants (top left), CUDA (bottom left) and OmpSs (right).

more detail in Section 2.4:

1. Loop offload models;
2. Block and grid models;
3. Blocked task models.

Loop offload models include variants of Accelerated OpenMP [19], OpenACC [7] HMPP [35], PGI accelerator directives [95] and Intel OpenMP extensions for MIC. They extend an OpenMP-like annotated serial source model with data-movement declarations to offload work to a device with a distinct address space. The top left of Figure 5.1 shows a basic GEMM kernel implemented serially, with OpenMP, Accelerated OpenMP and our proposed Accelerated OpenMP extensions. With no pragmas, the loop runs serially, as one would expect. The OpenMP pragma on line 4 work-shares the loop across CPU cores. Accelerated OpenMP's pragma, on lines 6-7, adds explicit *in* copies of the `a` and `b` arrays and an *inout* copy of `c`. Each of these first two pragmas workshares the loop iterations across a single address space, either CPU cores or a single GPU. We discuss the third pragma below.

Block and grid models include CUDA [3] and OpenCL [6]. These low-level models specifically target GPU-like hardware by offloading blocks or groups of threads to an array of cores, each of which is a SIMD unit. Generally these cores share memory with one another but not directly with the CPU. The lower left of Figure 5.1 shows an example using CUDA. In addition to changing the array accesses, explicit memory allocation and copies are required to move data to and from the device. The loop is converted into a grid of threads, each of which execute a single iteration in the `cudag()` kernel, which must be called with the number of blocks and threads per block. As with the loop offload example, this code uses exactly one GPU.

Blocked task models include StarPU [13] and OmpSs [37]. They specify tasks, and their dependencies, in terms of blocks of data and, sometimes, other tasks. The right-hand code-block in Figure 5.1 uses OmpSs to implement the GEMM kernel with load-balancing across CPUs and GPUs, so it contains both CPU and CUDA kernels both aliased to the `gemm()` function by the compiler. Each call to `gemm` is given the start address of the block, in this case row, to process. Since the task size is fixed, each task must represent enough work to occupy all compute units on a GPU long enough to amortize the overhead of scheduling it. However, each task must also be small enough not to overload a single CPU core. The runtime does not combine tasks or otherwise adjust task granularity. Blocked task code will target all CPU cores and GPUs dynamically at runtime however, for a significant gain in performance portability. The downside is that it requires code replication, explicitly providing CPU and GPU versions and explicit splitting of the available work into blocks.

Each of these approaches has advantages and disadvantages. The CUDA code is highly efficient on the GPU and offers maximum control over it. The loop offload version requires the least change from serial or OpenMP code, but offers less control. Blocked task models offer control through direct use of the other models, and the option of automatic load-balancing across all compute resources. Unfortunately they also require the greatest departure from the original code, and may require a redesign of the application's data layout. We need a programming model that offers the full performance of block and grid models, the flexibility of blocked task models and programmability of loop offload models. Our proposed extensions, along with our prototype library implementation, brings us one step closer to this goal by adding worksharing across devices to Accelerated OpenMP without requiring a specific task granularity from the user. The third pragma, in the upper left of Figure 5.1 lines 9-10, is how our proposed extension would workshare the GEMM loop across an arbitrary number and type of supported devices. Thus, it offers more flexibility in the region than even blocked-task models, while maintaining the serial loop as written.

5.3 Design

This section presents the design of our proposed extension, schedulers and memory management infrastructure. CoreTSAR safely divides annotated, un-blocked, serial loops, as used in many traditional OpenMP applications, and schedules them across heterogeneous resources. We add a clause to Accelerated OpenMP that is similar to the `schedule()` clause. The OpenMP programming model imposes the following constraints on our design:

1. Use existing, unchanged code in the Accelerated OpenMP loop region;
2. Treat the accelerated loop as a group of related tasks that are defined by the loop code and the region directive including its associated clauses;
3. Maintain data consistency outside of the region and do not alter data accesses in the existing loop body although we can extend the data copy clauses of the region.

By following these constraints we preserve programmability while adding significant new functionality.

5.3.1 The Proposed Extension

The CoreTSAR interface consists of two parts, which Figure 5.2 depicts. The `hetero()` clause specifies how to schedule the region and which classes of device should be considered. For memory management, we add the `part_copy()` clauses to provide the runtime with sufficient information to partition input and output data for the region safely.

```
//items in {} are optional
#pragma acc region \
  hetero(<cond>{,<devs>{,<sched.>{,<ratio>{,<div>}}}})\
  part_copy{in/out}(<var>{(<size>}[<cond>:<num>{:<halo>}])\
  persist(<var>)
#pragma acc depersist(<var>)
```

hetero() inputs	
`<cond>`	Boolean, true=coschedule, false=ignore
`<devices>`	Allowable devices (cpu/gpu/all)
`<scheduler>`	Scheduler to use for this region
`<ratio>`	Initial split between CPU and GPU.
`<div>`	How many times to divide the iteration space

part_copy() and {de}persist() inputs	
`<var>`	Variable to copy.
`<size>`	Size of each "item" in the array/matrix.
`<cond>`	Whether this dimension should be copied.
`<num>`	Number of items in this dimension.
`<halo>`	Number of halo elements required.

Figure 5.2: Our proposed extension

Our clauses are permitted on the accelerator directive or on any top level accelerator loop construct. Unlike normal copy clauses, `part_copy` is not allowed on data clauses as it only has meaning when directly associated with a loop. We still support data regions, but only for cases where complete replication of the input/output is desired, as opposed to only those data regions that are required. We define the properties of our clauses in greater detail in the following sections.

5.3.2 Scheduling

In order to workshare the iterations in a given region efficiently, we must offer appropriate scheduling policies. Each policy in CoreTSAR treats the iterations within a loop region as a group of related tasks, which allows us to select the scheduling granularity adaptively. For example, CoreTSAR can break a region with 10,000 iterations into four chunks or a thousand or any number less than 10,000 for scheduling, without modification or user intervention. This capability is critical for efficiently scheduling across heterogeneous systems, as a single grain size is rarely appropriate for all available devices.

Existing OpenMP schedules for CPUs divide the iteration space either evenly across processors statically or into *chunks* that are assigned dynamically. The static version is simple, efficient to implement, and consistent, but does not load-balance. Alternatively, OpenMP's chunk based schedules (*dynamic* and *guided*) load-balance well in homogeneous architectures. However, they suffer high overhead due to synchronization at each work-request and especially as a result of the lack of data locality their dynamic algorithms cause. In heterogeneous systems they would also incur repeated kernel launches and data transfers. We dealt with these issues in our initial work with CoreTSAR [87], by designing a set of adaptive schedules that statically divide the work within each pass through a region, but load-balances across passes. This scheme proved effective, but it does not tolerate imbalanced workloads well, whereas chunk-based schemes can. To address that case, and broaden our evaluation, we have developed a number of new chunk-based designs as well as a hybrid of the two approaches.

Static/Adaptive Schedulers

Our static and adaptive schedulers, based on those presented in Section 4.3, predict the time that each device will take to complete an iteration in the next pass, and generate a single task for each device sized such that all finish the region as close to simultaneously as possible. These schedulers make two assumptions: the average runtime of an iteration in the region is constant on a device; and subsequent passes through the region have the same performance ratio as the previous pass. Also, our schedulers begin with a default time per iteration for each device until we have gathered runtime data. This default is either a user-defined value, one saved from a previous run, or one based on the formula $1/(deviceSIMDWidth/baseDeviceSIMDWidth)$. While we do not claim that this formula accurately models the relative performance of devices, in practice we have found it to be accurate for dense floating point kernels. We leave further exploration of default values as future work.

As the linear program proposed as part of Splitter in Equation 4.1 yields an "optimal" solution based on our scheduling model, we first implemented CoreTSAR with exactly that linear program. In order to ensure the solve itself is efficient, CoreTSAR employs the lp_solve [18] library, an optimized linear program solver with the capability to refine an existing solved tableau into the solution for a new set of inputs. This incremental approach greatly reduces overhead for scheduling since each pass tends to have similar inputs to the one before.

Figure 5.3 represents the time spent in CoreTSAR scheduling 1,900 passes through a region, or 19,000 scheduling iterations with the split scheduler. Despite the use of an optimized implementation of an integer linear solver, the original linear model has exponential time complexity as the number of GPU devices increases. In the worst case, the split schedule with four GPUs, the scheduling takes nearly $3\times$ longer than the 40-second compute phase. An overhead of that magnitude is unacceptable, especially for applications with many short regions.

Two issues reduce the solver's performance. First, its input is made up of widely distributed values, which leads to poor numerical stability that slows convergence and frequent floating-point error

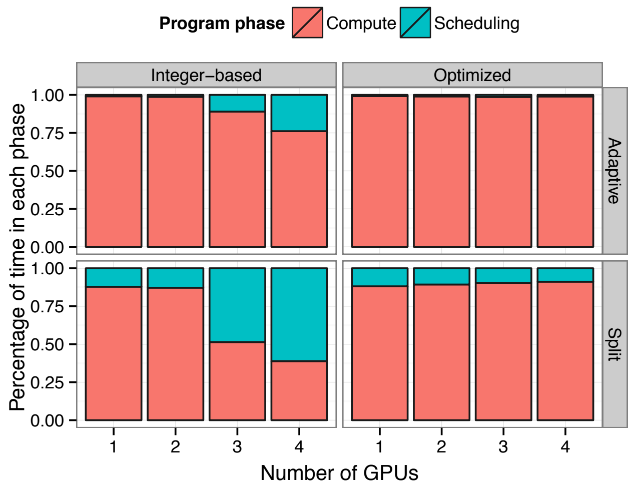


Figure 5.3: Percentage of time spent on computation and scheduling for a benchmark running 1,900 passes with and without the optimized linear program.

corrections. Second, all of the outputs require integer values. This requires the solver to refine an optimal solution into an optimal *integer* solution across all of those values, significantly increasing computational complexity.

The linear program in Figure 5.4 uses the time per iteration value for each device to calculate the fraction of the total available iterations that should be allocated to each device. In words, the program finds the fractions of work that result in the minimum deviation between predicted execution times. By removing the integer requirement on the output, we reduce the complexity of the solve, and also improve numerical stability in the floating point portion by providing numerically smaller input values. As Figure 5.3 shows, the new linear program minimizes the overhead of computing the schedule compared to the original.

Using this linear program, over our mixed-integer version, has a downside however. The result no longer represents the number of iterations to assign to each device. Instead, the runtime multiplies

I =total iterations
i_j =iterations for compute unit (CU) j
f_j =fraction of iterations for CU j
p_j =recent time/iteration for CU j
n =number of CUs
$t_j^{+/-}$ =time over/under equal

$$min(\sum_{j=1}^{n-1} t_j^+ + t_j^-) \quad (5.1)$$

$$\sum_{j=0}^{n} f_j = 1 \quad (5.2)$$

$$f_2 * p_2 - f_1 * p_1 = t_1^+ - t_1^- \quad (5.3)$$

$$f_3 * p_3 - f_1 * p_1 = t_2^+ - t_2^- \quad (5.4)$$

$$\vdots$$

$$f_n * p_n - f_1 * p_1 = t_{n-1}^+ - t_{n-1}^- \quad (5.5)$$

Figure 5.4: Our adaptive scheduler's deviation minimization algorithm as a linear program, variables at left

each f_j by I and assigns the nearest lower integer. This approximation can fail to assign up to n iterations. Since most applications execute thousands of iterations, the small deviation is usually within the error threshold of our measurements. To ensure that rare corner cases are handled, CoreTSAR assigns these iterations based on a heuristic that by default adds the iterations in round-robin across devices in descending order of TPI. When a device has been assigned less than I/n iterations however, it is passed over. The extra condition prevents cases where an incapable device might otherwise be assigned tan iteration when it should get none, or two when one is the only appropriate response. In practice, cases where the assignment order and distribution noticeably affect performance is rare, but assigning the extra iterations in this manner handles those cases as well.

Our *static* schedule applies this linear program to the default, or supplied, values once at the beginning of the first pass through a region, then reuses the result thereafter. The adaptive schedules (*adaptive*, *split* and *quick*) use a first pass with the static schedule as a training phase. The first time that we encounter a CoreTSAR region, we assign work based on the static schedule and then measure the times on each device. For each following scheduling decision, we use measured times per iteration in the linear program, converging on a more efficient schedule. Our design intentionally includes all recurring data transfer and similar overheads required to execute an iteration on a particular device, naturally incorporating data transfer and launch overheads.

Adaptive trains on the first full pass through the region, then adapts at the beginning of each subsequent pass. *Split* is designed to adapt within regions that either cannot tolerate a full pass with a poor schedule, or only run once per application run. *Split* breaks each pass into several evenly split subpasses, based on the `div` input. It treats each subpass as the same as a full pass with *adaptive*. While *split* provides better, and earlier, load-balancing for some applications, it increases overhead in each pass. *Quick* balances between *split* and *adaptive* by executing a small subpass for its first training phase, similarly to *split*. It then schedules and runs all iterations remaining in the first pass, and uses the adaptive schedule for all subsequent passes. This schedule suits applications that cannot tolerate a full pass using the static schedule or the overhead of extra scheduling steps in every pass.

Chunk Schedulers

Chunk schedulers are exemplified by the OpenMP *dynamic* schedule, in which a chunk size specifies the number of iterations assigned to each thread each time it requests work. These schedulers effectively use a work queue approach, which offers natural load balancing. While it is a classic load balancing approach, it is most effective when used with homogeneous computing resources with fast synchronization mechanisms, which is not the environment that CoreTSAR targets. Thus, we present variations on this method for hybrid systems.

Specifically, we design three new schedules (*chunk*, *chunk static* and *chunk dynamic*). *Chunk* serves as our baseline chunk scheduler, and is functionally identically to OpenMP's *dynamic* schedule. *Chunk static* scales the chunk size for each device based on the same scheme used in the *static* schedule above. Thus, larger chunks are provided to devices that complete their iterations faster, with the goal of each chunk running for the same length of time regardless of the target device. Finally, *chunk dynamic* begins in the same way as *chunk static* then dynamically adapts the chunk size for each device based on their performance. Unlike the adaptive schedulers, it does not employ the linear program to determine the new chunk size since the chunk schedulers do not have a natural barrier point where all times have been updated. Instead, it employs an annealing approach that computes a weighted average of the time per iteration for each device, and attempts to reduce the time per iteration by increasing or decreasing the chunk sizes. For example, if the time per iteration on a device decreases with an increased chunk size, *chunk dynamic* again increases that chunk size. In this design, each device is independent and does not block on the others in order to adapt.

Hybrid Scheduler

In addition to the schedulers that are strictly chunk or ratio based, we also investigate a *hybrid chunk* schedule that begins as a chunk dynamic schedule and after the first pass transitions into the adaptive scheduler. *Chunk dynamic* adapts and load balances quickly during the first pass while refining the split. However, after that first pass, it incurs unnecessary overhead in contention and memory transfers, where adaptive excels. Using *chunk dynamic* in place of *static* for the training phase of *adaptive* naturally fuses the advantages of both schedulers.

5.3.3 Memory Management

In order to maintain memory coherency across address spaces while dynamically splitting the region, CoreTSAR requires information about the memory access pattern of each iteration of the loop. The primary design goal of our memory manager is to support unblocked input and output data naturally. Thus preserving the data layout of the serial code. Our interface covers a wide range of cases while a method to specify any possible association is an ongoing topic of research, including our future work. As we discuss in Section 5.4, our prototype library implements the memory management for all examples evaluated in Section 5.5.

For each variable the `part_copy()`, or partial copy, interface requires at least a variable name, one dimension to copy, and the number of items in that dimension. Given the clause `part_copyin(a[1:N])`, `a[i]` will be copied, where `i` is the current loop iteration, to the device that executes that iteration. If a range of `i` from $5 - 500$ is assigned to a device, `a[5-500]` will be copied. Thus, the partial copy is associated with the loop iteration(s) of the loop being split.

Figure 5.5 displays two simple examples of patterns produced by our memory-association syntax. The top example specifies that a 10×10 matrix is being registered to the region, and the iterator will be associated with the column dimension, assuming C ordering, since the column dimension's condition is true. The lower example selects the row dimension instead, and additionally specifies

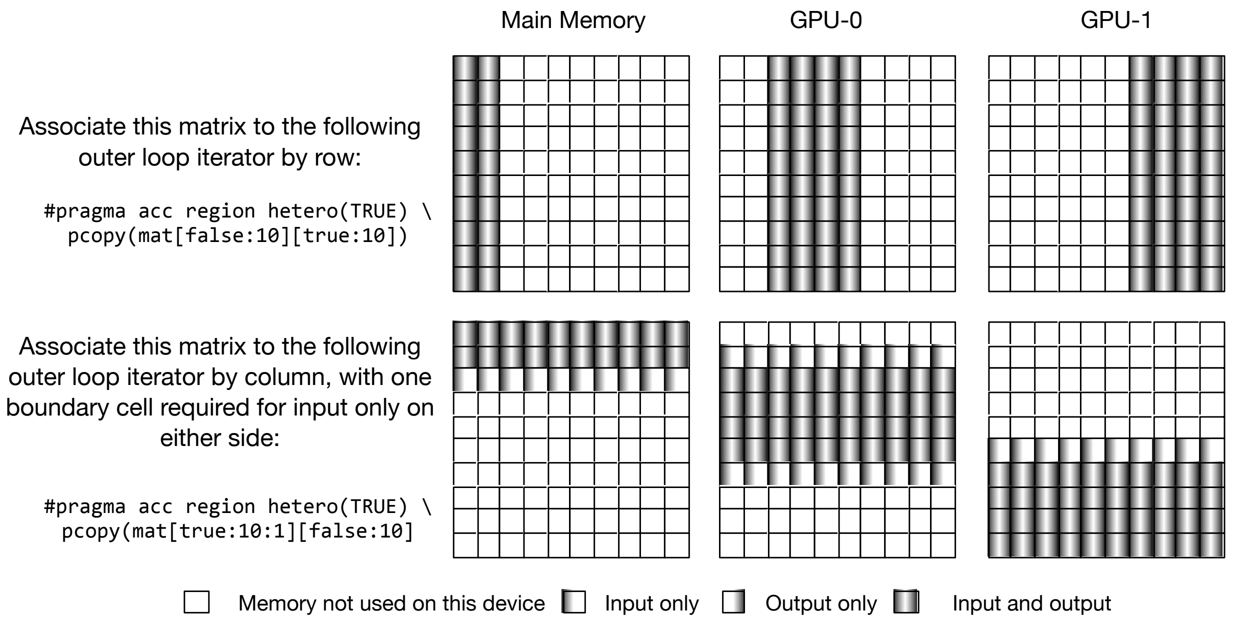

Main Memory
GPU-0
GPU-1
Associate this matrix to the following outer loop iterator by row:
#pragma acc region hetero(TRUE) \
pcopy(mat[false:10][true:10])
Associate this matrix to the following outer loop iterator by column, with one boundary cell required for input only on either side:
#pragma acc region hetero(TRUE) \
pcopy(mat[true:10:1][false:10]
Memory not used on this device
Input only
Output only
Input and output

Figure 5.5: Example memory association patterns, assuming a pass in which two iterations are assigned to the CPU device, and four each to two GPUs.

that one halo element is required on either side of that dimension. This pattern is typical of stencil-type computations, where halo values are required as input, but are not updated by the device reading them, having the halo argument makes supporting such associations natural.

Our interface does not directly support random access output, reverse indices or indirect indices. For input, these can all be supported at the cost of additional overhead by copying the entire data set, since the input process is non-destructive.

5.3.4 Example Usage

Figure 5.6 shows how to use our proposed interface to implement the example in Figure 5.1. All options, including those that use default values, are specified for `part_copyin(a...)` and

```
void runGemm(T **a_a, T **b_a, T **c_a) {
  T *a = *a_a, *b = *b_a, *c = *c_a;
#pragma acc region for part_copy(c[1:N][0:N])        \
        part_copyin(a[1:N:0][0:N:0]) copyin(b[0:N*N])\
        hetero(1, all, adaptive, default, 10)
  for (int i = 0; i < N; i++) {
    for (int j = 0; j < N; ++j) {
      c[(i * N) + j] *= B;
      for (int k = 0; k < N; ++k) {
        c[(i*N) + j] += A * a[(i*N) + k] * b[(k*N) + j];
      }
    }
  }
}
```

Figure 5.6: Our proposed extension to accelerated OpenMP.

`hetero(...)` in the example. The minimum necessary to specify the copy correctly are used for `part_copy(c...)`. In this example, the loop will always be split across devices using the adaptive scheduler with the default ratio and a div of 10. The copies specify that the `a` array is two-dimensional, of size N by N, made up of elements of size `sizeof(T)`, and that iteration i requires row i of `a` but not column i. The `c` copy specifies the same as for `a` except that it should be copied both in and out. The traditional `copyin()` clause from accelerated OpenMP is used for `b` since all participating devices need access to the full region. Complete output, in the form of `copyout()`, is not allowed however because it does not provide enough information to resolve the changes made by each device. We may investigate this in future work.

5.4 Implementation

We implement CoreTSAR as a C library on top of Accelerated OpenMP, tested with PGI Accelerator and Cray's Accelerated OpenMP. Our evaluation in this chapter focuses on PGI Accelerator, so our examples use its directive format. This section discusses our implementation including its portability, API and our memory manager as well as some necessary deviations from the design discussed in Section 5.3.

5.4.1 GPU Back-off

Some applications are not amenable to being run on GPUs, or at least the GPUs present in some systems. While iterations of a region may benefit greatly from running on a powerful compute GPU such as an NVIDIA Tesla C2075, they may perform poorly on a consumer-grade NVIDIA GeForce GT 520. CoreTSAR's base design assumes that trading one CPU core to control a GPU will always be a gain in performance. While often a safe assumption, many applications do violate it. As a result, we have added handling for applications that run more efficiently using one CPU core than using that core to control a GPU. In order to maintain portability across disparate accelerator and CPU capabilities, CoreTSAR implements GPU Back-off support in all adapting schedulers.

The back-off system is implemented differently for each of the two main scheduler types. In the adaptive schedulers CoreTSAR converts a GPU offload thread into a CPU thread when the GPU has a higher time per iteration than the slowest CPU core for a configurable number of passes (default is two). We use multiple iterations since under certain circumstances, such as loading large persistent datasets for the first time, or an inappropriate initial amount of work, a device can be erroneously classified as slower than the CPUs. With the chunk schedulers, we base the decision on whether a given GPU completes fewer iterations than the slowest CPU core during each pass of a configurable number of passes. This difference compensates for the sometimes highly variable

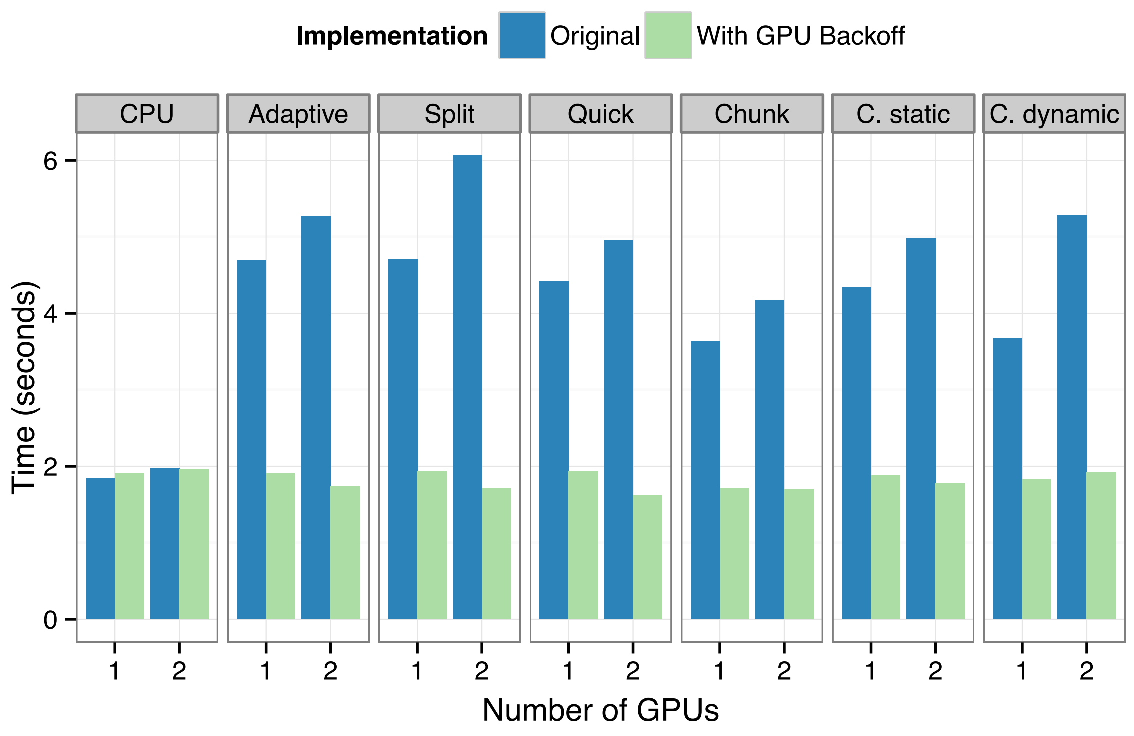


Figure 5.7: Helmholtz with and without GPU backoff support

time per iteration when bootstrapping chunk schedulers across initial data copies, which can cause false conversions with the adaptive back-off scheme.

Figure 5.7 displays the results for Helmholtz without GPU back-off in the *Original* series and with it in *With GPU back-off* series. While the original result is clearly inferior to the CPU performance, at left, the version with back-off maintains within 5-10% slow-down. Overall the *With GPU back-off* version can be as much as $3\times$ faster than *Original*. Further, having released the GPUs, it uses fewer resources and less power. Regardless of the number of GPUs, CoreTSAR decides to stop using all of them. While we do not achieve a speedup, this result is positive, bounding the loss imposed by a unsuitable application to a reasonable level. We discuss the effects of this extension on other benchmarks in Section 5.5.

5.4.2 Memory Management

The existing memory interface of Accelerated OpenMP is insufficient to express the relationships necessary to handle certain kinds of memory association. While Accelerated OpenMP does natively support copying a subset of an array, it does not support copying multiple subsets of one array, nor does it support non-contiguous rectangular sections such as a subset of the columns of a 2D array.

In order to support our desired memory association interface, CoreTSAR implements its own memory manager, using the `deviceptr()` clause to pass CoreTSAR managed memory into Accelerated OpenMP regions. We offer a straightforward syntax by which users specify the data required by a given iteration. Given that information, CoreTSAR automatically copies the ranges of data required by whatever iterations are assigned to a given device for that pass. When possible the memory manager uses pinned memory to accelerate copies, as well as asynchronous copies to and from the device in order to overlap them with scheduling and synchronization.

Currently, the CoreTSAR memory manager handles a restricted set of partial copies. In addition to the straightforward one-to-one relations, CoreTSAR also supports stencils through padding, and row, column or planar associations on two and three dimensional matrices. In order to support reductions we provide an API inspired by user-defined reductions in OpenMP 4.0. We discuss the details further in Section 5.4.4. While only a subset of the possible cases, these mappings are sufficient to implement all benchmarks evaluated in this chapter.

5.4.3 Data Packing and Padding

Our original implementation of the memory interface [87] had a material weakness. That version of CoreTSAR allocated the full size of each memory region on each device in order to preserve offset accuracy. In other words, any input or output array/matrix supplied to CoreTSAR was allocated in full in all participating address spaces. Managing subset allocation and access without invalidating offsets and iterator values is a difficult problem, especially in languages like C.

We have redesigned CoreTSAR to support three kinds of regions depending on how the data is mapped. The first, and most simple, case is a one dimensional partial array or two dimensional array that is associated by rows. Since all of the resulting subsets are contiguous, the runtime provides an offset pointer that can be indexed by the original offsets without issue. No further action or overhead is required for this case, and a significant amount of storage on accelerators can be saved. The second case is where a two or more -dimensional array is associated by columns. CoreTSAR can pack these, but must have control over the calculation of offsets into the resulting matrix. As such, we handle this case in our translator for contiguous arrays accessed with `array[i][j]` style syntax, but currently do not support dynamically-allocated C arrays accessed with the `array[i*row_size + j]` syntax, though these can be supported by directly using the C API functions. Third, associations can use both the row and column associations, resulting in a region resembling a plus-sign being assigned to each device. Since these require the full range in both rows and columns, even though they may not need the corners, CoreTSAR is forced to allocate the full size of such arrays.

By allowing data regions to be packed, CoreTSAR gains two extra capabilities beyond reducing memory usage on target devices. The packing functionality allows any chunk-based scheme to place a low bound on memory use by selecting a small chunk size. This allows large data sets on the host to be streamed through accelerators without enough memory to hold even their assigned sub-part of the problem. When used in this mode however, CoreTSAR becomes similar to a blocked-task system, including the increased task management and data-transfer overhead that implies.

Perhaps more importantly however, the capability to adjust indexing, as described for column-wise associated multi-dimensional arrays, allows regions not only to be shrunk to save space, but also padded for alignment. As is well known, memory alignment is important for the performance of SIMD computations and coalesced memory accesses are important to the performance of GPU kernels. Given the ability to pad rows beyond the data assigned to each device, or even rows of data that are mis-aligned by the user, CoreTSAR can ensure that each row is aligned for most efficient access on each target. We implement this optimization by ensuring that the length of each row in a matrix is a multiple of the target device's SIMD width. While in some cases this choice is more strict than required, it is consistently sufficient to ensure reasonable alignment.

Figure 5.8 shows the effect that even small changes in row-width can have on performance without padding, and how our auto-padding helps. The figure represents the performance of a general matrix multiplication kernel when run with row and column lengths ranging from 8,192 to 8,223 elements in increments of one, specifically the x-axis values are the number of elements over 8,192 in each row. On our primary evaluation system, with four NVIDIA Tesla C2070 GPUs, a square matrix of size 8192×8192 runs in 7.9 seconds. Increasing that size by only one element on each side more than doubles that runtime to 19.5 seconds. In fact, every odd-numbered increase in size takes approximately the same increased time, while each power of two increase does better up to 16, or a total of 8208, which performs the same as the original 8192. Another system with a pair of NVIDIA Tesla K20x GPUs shows a nearly identical stepped pattern as well. The difference in performance is somewhat smaller, ranging from 10.9 seconds at the zero and 16 positions and 15.2 at the odd offsets. The L2 read request performance counters provide a partial explanation for the wide range in performance. When padding is enabled, the difference in L2 requests with a multiple of 16 row length is consistently within 10%; for any odd length it balloons to 80% more L2 accesses for the un-padded version. This increase is due to a greater number of reads being required to accommodate the mis-aligned read requests of each warp on the GPUs, increasing contention and lowering cache efficiency overall. On the right-hand-side of the plot, you can see that CoreTSAR's automatic padding smooths out these issues. Also, when dividing a data-set column-wise, this padding support can ensure alignment even when the appropriate amount of work to be assigned is not a multiple of the target device's native SIMD width, an important consideration for several of our benchmarks.

5.4.4 CoreTSAR API and Usage

This section describes the low-level API to the CoreTSAR library in detail.

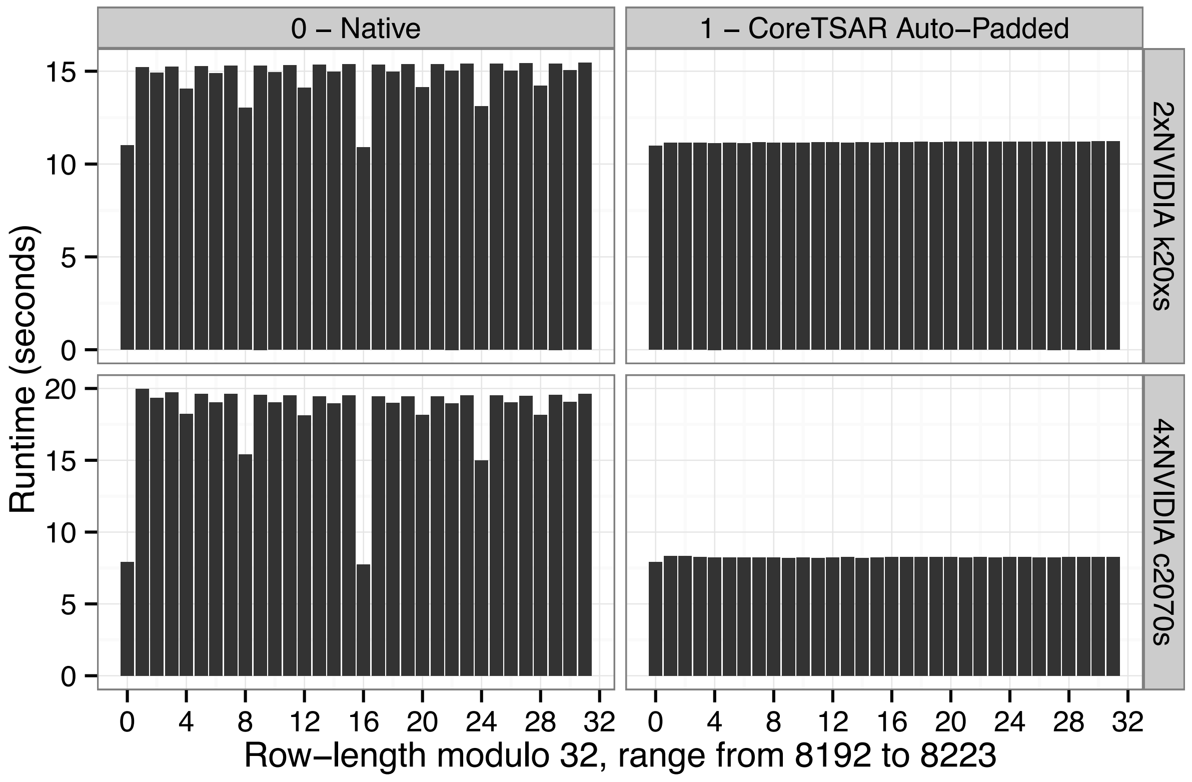


Figure 5.8: Runtime of GEMM kernel on square matrices, statically scheduled across GPUs only with and without auto-padding

ctsar_init Initializes an instance of the CoreTSAR runtime, one such instance should be used for each region that is to be separately scheduled. The parameters allow a user to set the default scheduler, allowed devices, the default time per iteration for each accelerator as an array of doubles (NULL for defaults), and how finely the split and quick schedulers should divide regions (NULL for default).

ctsar_next Computes the division of work for the region associated with `c` based on `size` total iterations. This function is also responsible, updating appropriate memory regions on each target device and starting timers to evaluate each device's performance.

ctsar_loop In order to support *split*, *quick* and the chunk schedulers, CoreTSAR must reevaluate the loop with each thread repeatedly. The `ctsar_loop` function serves as the condition for a do/while loop surrounding each region. In addition to managing repeats, the loop function is responsible for synchronization, GPU back-off support, copying data back from all devices, completing reductions, and calculating performance statistics at the end of each pass.

ctsar_reg_mem{**_2d**} These functions register a host buffer with CoreTSAR. The full version takes a pointer to CPU memory, the size of an element of the input data, the number of element in each row/column, the number of halo elements required, and a flag option that allows the user to control copy direction and type. The non-2D version is shorthand for 1D arrays. The return value is a pointer to the memory assigned to the calling thread, which may or may not be identical to the original pointer.

The flags value controls whether memory is copied in or out or both, as well as whether to copy persistently, partially by rows or partially by columns and whether padding is to be allowed, if it is, an extra output parameter for the new row size is required. Partial copies are integral to the correct functioning of CoreTSAR as they make automatic merging of output possible. They also improve performance of input operations. The 2D interface supports all specifications discussed in Section 5.3, except that it does not handle matrices with dimensionality higher than 2.

Regardless of the flags, CoreTSAR allocates an appropriate size buffer on the device associated with the calling thread. If the region is set to persistent, data is immediately and asynchronously copied from the CPU array into the newly allocated memory, where it resides until it is explicitly removed with a call to `ctsar_unreg_mem()`.

ctsar_unreg_mem De-registers the pointer from the region instance, frees the memory that stores the state of the data, and frees persistent regions.

ctsar_retarget_mem Re-target allows a user to specify that the region already allocated for pointer `old` should be used to store the data pointed to by `new` on all devices. A typical use is to swap buffers for double buffering, although it can also be used to implement blocked data transfers by re-targeting a pointer to the new start pointer before entering a region.

ctsar_reg_reduc This function registers a reduction. Because each memory space will have its own reduction result, CoreTSAR must safely initialize the temporary variables in each memory space and combine those results into a meaningful final value. The identity pointer points to an appropriate initial value to use on each device. For example, in a sum the identity would usually be 0, in a product 1, and so on. The `item_size` specifies the size of the elements to be reduced. The `reduc` argument is function pointer that should accept two void pointer arguments, the first of which is both a value to be reduced and the output, the second is another value to be reduced. This function is called repeatedly to accumulate the final value as each device finishes execution. For simple reductions, the body of the function can be as simple as `*((int*)a)+=*((int*)b)`.

CSTART/CEND Macros used to retrieve the start and end values to use for iteration in the loop region.

Figure 5.9 presents an example using this interface to implement the extension as presented in Figure 5.6. In this example, CoreTSAR is initialized with the adaptive scheduler, default ratio, and div of 10. The parallel do-while loop allows our library to reevaluate the code region as necessary

```
void runGemm(T **a_a, T **b_a, T **c_a) {
  ctsar * s = NULL; int div = 10;
  ctsar_init(&s,N,CTSAR_ADAPTIVE,CTSAR_DEV_ALL,NULL,&div);
#pragma omp parallel default(shared)
  do{
    T *a = ctsar_reg_mem(s, a_a[0], sizeof(T)*N, N,
            CTSAR_MEM_PARTIAL | CTSAR_MEM_INPUT);
    T *b = ctsar_reg_mem(s, b_a[0], sizeof(T)*N, N,
            CTSAR_MEM_INPUT);
    T *c = ctsar_reg_mem(s, c_a[0], sizeof(T)*N, N,
            CTSAR_MEM_PARTIAL | CTSAR_MEM_INOUT);
    ctsar_next(s,N);
#pragma acc region for deviceptr(a,b,c) independent \
        if(ctsar_get_type(s) == CTSAR_DEV_GPU)
    for (int i = CSTART(s); i < CEND(s); ++i) {
      for (int j = 0; j < N; ++j) {
        c[(i * NJ) + j] *= B;
        for (int k = 0; k < N; ++k)
          c[(i*N) + j] += A * a[(i*N) + k] * b[(k*N) + j];
    } }
  }while(ctsar_loop(s));
}
```

Figure 5.9: CoreTSAR library version of GEMM

by looping with the `ctsar_loop(s)` call until done. The data regions are registered, as partial input, complete input, and partial input/output, and the appropriate pointers for those data regions on each device are returned into the local copies of pointers `a`, `b` and `c`. The `ctsar_next()` call calculates the number of iterations to be completed in this pass by each device. Once it is complete, the `CSTART()` and `CEND()` macros return the appropriate iterator range values for the device that evaluates them. This syntax can either be used manually, or generated by our python/libclang-based source-to-source translator.

While the code is extended significantly around the loop, we do not replicate or alter any code in the loop body. The Accelerated OpenMP `if()` clause determines if a thread runs on a GPU or CPU core. If the device is a CPU, the loop is run serially on the associated core completing its assigned iterations. If it is a GPU-controlling thread, the `acc region` directive work-shares the assigned iterations across the associated GPU. All codes used in our evaluation are implemented

System name	CPU Model	CPU Cores/die	CPU Dies	CPU RAM (MB)	GPU Model	GPU Cards	GPU Cores	GPU RAM (GB)
amdlow3	E3300	2	1	2,012	Tesla C2050	1	448	3
armor1	E5405	4	2	3,964	GeForce GT 520	1	48	1
dna2	i5-2400	4	1	7,923	GeForce GTX 280	1	240	1
escaflowne	X5550	4	2	24,154	Tesla C2070	4	448	6
hokiespeed	E6545	6	2	24,154	Tesla M2050	2	448	3

Table 5.1: Test system specifications, all CPUs and GPUs are made by Intel and NVIDIA respectively

in this fashion.

5.5 Evaluation

This section evaluates the CoreTSAR library. We compiled all benchmarks with the PGI Accelerator compiler compiler suite version 12.9. Optimization flags are `-acc -ta=nvidia -O3 -mp=allcores`. Table 5.1 lists our test platforms. Unless otherwise specified, tests were run on escaflowne. In tests with GPUs enabled, one CPU core is used to control each selected GPU and does not do computation. We use default scheduler parameters unless otherwise specified, with the initial split calculated at runtime based on the available resources and a div of 10. We include all scheduling overhead, GPU data transfer time, and synchronization time in all measurements.

Reported times and speedups include all activity that the original OpenMP CPU code did not require, such as library initialization, scheduling, and memory movement. We do not include application IO or problem setup that is shared between CPU, GPU and scheduled versions. We also record the time for each device to complete its assigned iterations, from which we can compute the time that devices wait for others to complete, the time spent to calculate the split for the next pass and, as a subset of that, the time to solve the linear program. Finally, we track the time per iteration for each device, as described in Section 5.3.

5.5.1 Benchmarks

We use four applications and the PolyBench/GPU [46] benchmark suite in our evaluation of CoreTSAR, all of which are described in greater detail in Section 2.7. CG [15] is a direct port of the NAS conjugate gradient benchmark. GEM [8] is a molecular modeling application for the study of the electrostatic potential along the surface of a macromolecule that has been extensively studied for GPU optimization [32]. Helmholtz is a discrete finite difference code that uses the Jacobi iterative method to solve the Helmholtz equation. K-means is a popular iterative clustering method. Our implementations of the 15 PolyBench/GPU benchmarks execute each computational kernel 10 times to mimic use in an iterative scientific application more closely. Tests at 5 and 15 kernel executions yield similar relative results. Since we are evaluating scheduling behavior, and not computational kernel performance, we made minimal changes in porting each benchmark. As such, our computational kernels are not optimized for the GPU other than by the compiler. Nonetheless, CoreTSAR can easily support optimized implementations through the same syntax.

5.5.2 Input Parameters

As mentioned above, we use the default values for our tests unless otherwise specified. However, chunk size does not have an obvious default. Figure 5.10 illustrates the performance for the basic chunk scheduler across chunk sizes for each benchmark using one GPU. We do not report chunk sizes in terms of absolute iterations, which has little meaning across benchmarks. Instead, we compare by the number of chunks into which the region is partitioned. The performance of some applications varies little based on chunk size. Others, such as CORR and COVAR, have a range of as much as $3\times$. These ranges shift or even reverse in some cases as the number of GPUs or scheduler changes, creating even more variability. Due to the sensitivity to this parameter, all subsequent results for chunk-based schedulers use the best chunk size for that benchmark, scheduler, and GPU count combination.

5.5.3 CoreTSAR Performance

We begin with an evaluation of the overall speedup achieved for benchmarks across schedulers and GPU counts on escaflowne, as Figure 5.11 depicts. All plots are based on the speedup over a chunk-based CPU schedule equivalent to OpenMP's dynamic schedule across the 8 CPU cores. We can group these results roughly into three groups of behavior: those that scale to all four GPUs; those that benefit from GPUs but do not scale to more than one; and GPU averse applications.

GPU Amenable Applications

Eleven benchmarks scale to four GPUs on escaflowne, resulting in between $3.5\times$ and nearly $200\times$ speedup. First GEM, GEMM, kmeans, SYRK, SYR2K, TWOMM and THREEMM scale nearly linearly from one to four GPUs, missing linear only because of the use of one CPU core for the addition of each GPU. Slightly off of linear are CORR and COVAR, which gain performance at

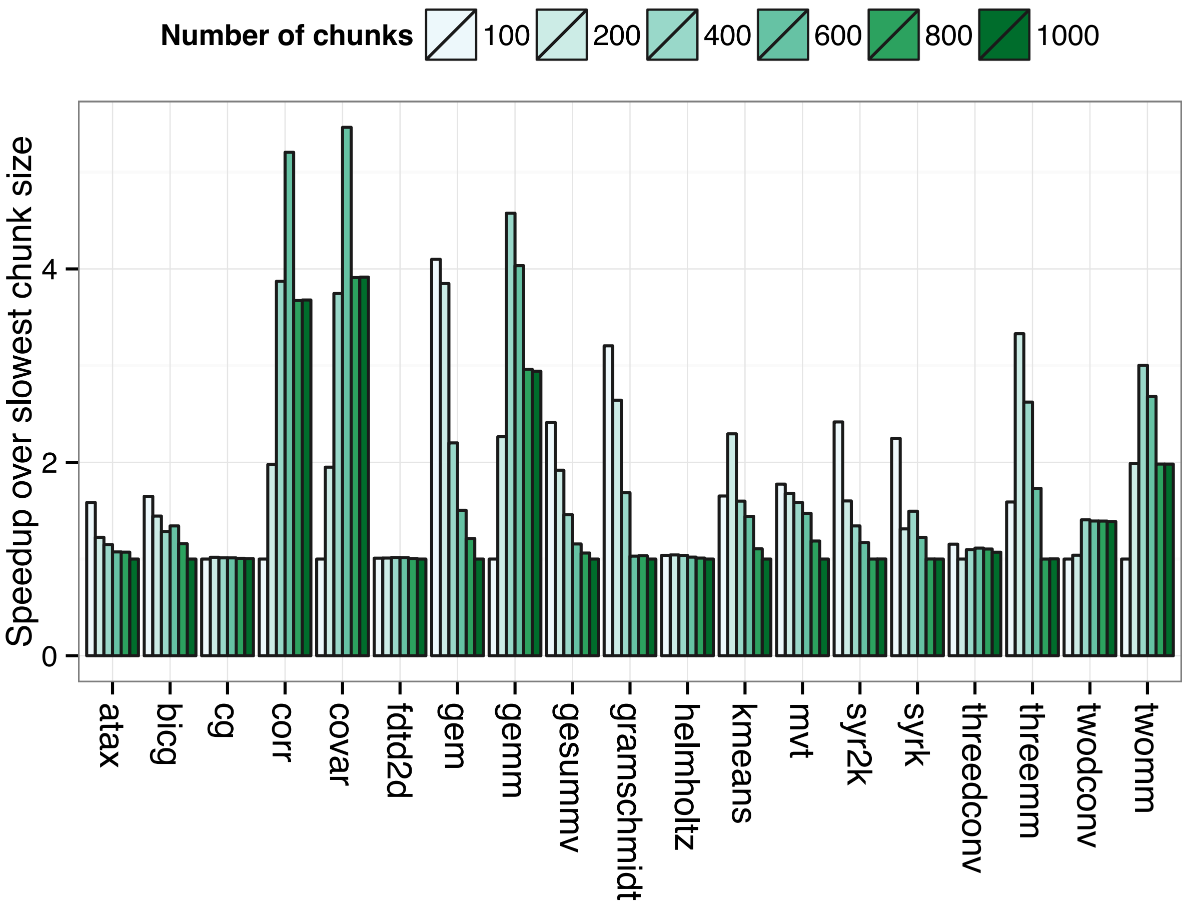


Figure 5.10: Performance across chunk sizes for each benchmark with the basic Chunk scheduler

approximately one quarter of that rate, but consistently up to all four GPUs. Also in this group are ATAX, BICG, and MVT, which clearly taper off after two GPUs, since these benchmarks do not have enough work available at this problem size, to occupy all four GPUs fully. Further, we cannot increase the problem size without overflowing the GPU memory due to the way CoreTSAR's memory model currently handles mappings. In another peculiarity of these three benchmarks, the chunk scheduler performs almost identically to the CPUs. While all three reap significant

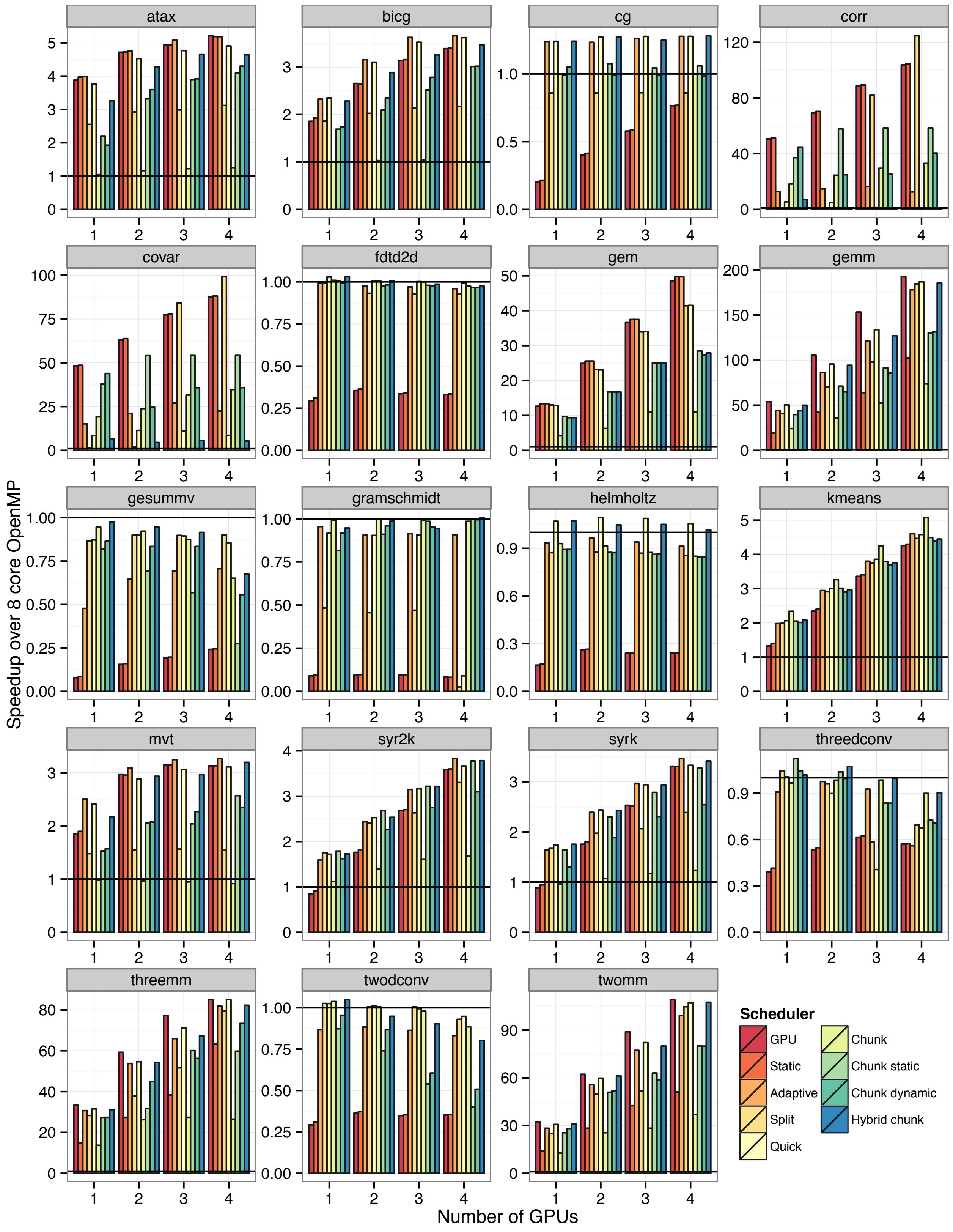


Figure 5.11: Performance across schedulers and number of GPUs for all benchmarks, normalized to CPU OpenMP across 8 cores.

performance benefits when run on GPUs, they are the only benchmarks that use column-wise partial copies. The overhead of column-wise copies for each chunk apparently causes the runtime to deactivate all GPU threads for the basic chunk scheduler.

In terms of individual benchmark behavior, GEMM achieves the most speedup, which occurs with the static GPU-only configuration. While this schedule is not an adaptive, it is still facilitated by CoreTSAR, and for extremely GPU suitable applications can outperform the adaptive schedules. The CORR and COVAR benchmarks superficially behave similarly, but for a different reason. In their imbalanced workloads, each iteration i does $n - i$ units of work. Thus, they violate the assumption of the adaptive schedulers that the average work per iteration is constant. We expected one or more of the chunk schedulers would perform best in this scenario, but both CORR and COVAR are highly sensitive to overhead, and cannot tolerate the additional launches and copies of the chunk schedulers. Thus, the static schedulers (GPU and static) perform best in most cases. In the four GPU case for each, however, the split scheduler surges ahead. Split *stops using the CPU cores* and schedules across the GPUs in the three and four GPU cases. Our linear program does not handle varying time per iteration with heterogeneous hardware, but given relatively homogeneous hardware it handles the heterogeneous iterations much better. Using only GPUs, no CPU cores, with the Adaptive schedule achieves a further 10-20% performance improvement over the next best schedule in each case for CORR and COVAR.

Also unexpectedly, kmeans performs best with the basic chunk scheduler. With a precisely selected chunk value kmeans does quite well but, as Figure 5.10 shows, its performance varies by as much as 50% across chunk sizes we tested. The adaptive schedulers are more robust in that they do not require users to search the input space in order to find a reasonable initial parameter.

Overall CoreTSAR scales well to at least four GPUs without loop body or memory layout changes for GPU-amenable applications. Further, each scheduler is stronger for certain tasks than others, and the adaptive scheduler is the best overall choice, even with the best chunk sizes for chunk schedulers. It remains stable, and within approximately 10% of optimal for all amenable benchmarks with homogeneous iterations. For heterogeneous iterations, static and chunk are better options.

GPU Averse Applications

These are applications that *do not* run well on GPUs. Some are so sensitive to it that running any part of the job on a GPU causes slowdown. These are included to evaluate CoreTSAR's response to regions that should not use GPUs, or to running normally amenable applications on a system where the accelerator is particularly slow. While Jacobi solvers in general, and Helmholtz solvers in particular, are not GPU averse as a class, the implementation of Helmholtz that we evaluate is. Our Helmholtz is a generic CPU OpenMP version that runs correctly but slowly when compiled for the GPU. It never performs better by using a GPU for any work. This category also includes three PolyBench/GPU benchmarks (FDTD2D, GESUMMV and GRAMSCHMIDT). Each runs slower on a GPU than on one CPU or runs many passes, accentuating copy overhead.

In each case, schedulers that run more often, and thus convert the GPU threads to CPU threads faster, incur less performance loss. For the same reason, GPU-averse benchmarks that run many small passes perform better. For example, GESUMMV suffers more than the others by running 10 passes rather than 50 or thousands. For each of these benchmarks, the ability to convert GPU control threads to CPU threads is crucial. Without GPU backoff support, the total runtime of Helmholtz more than doubles for both adaptive and chunk based schedules, and as much as triples for the split schedule.

Three other benchmarks (CG, THREEDCONV, and TWODCONV) fall into this category, but only marginally. Each can benefit from the first GPU. GPUs complete iterations faster than CPUs for these benchmarks, but they only have enough work to saturate a single GPU, or face increasing data transfer overhead as more are added. CG passes through the region enough times (1,900) that all but one GPU are converted to CPU threads very early in the computation, so it achieves roughly constant performance from one to four GPUs. The convolution codes do not run long enough to hide the overhead of extra GPUs enabled in the first few passes and show degrading performance.

5.5.4 Adaptation Across Machines

We now evaluate CoreTSAR's performance across several disparate systems. All systems run the same OS image and execute identical binaries for all tests. Table 5.1 lists the hardware in each system in detail. Of particular interest are the GPU-centric system amdlow3, which contains a dual-core Intel Celeron processor and NVIDIA C2050 GPU, and the CPU-centric system armor1 with two quad-core Intel Xeon cores and a low power NVIDIA GT 520 GPU.

As some of our benchmarks require more memory than the smaller GPUs posses, we selected a

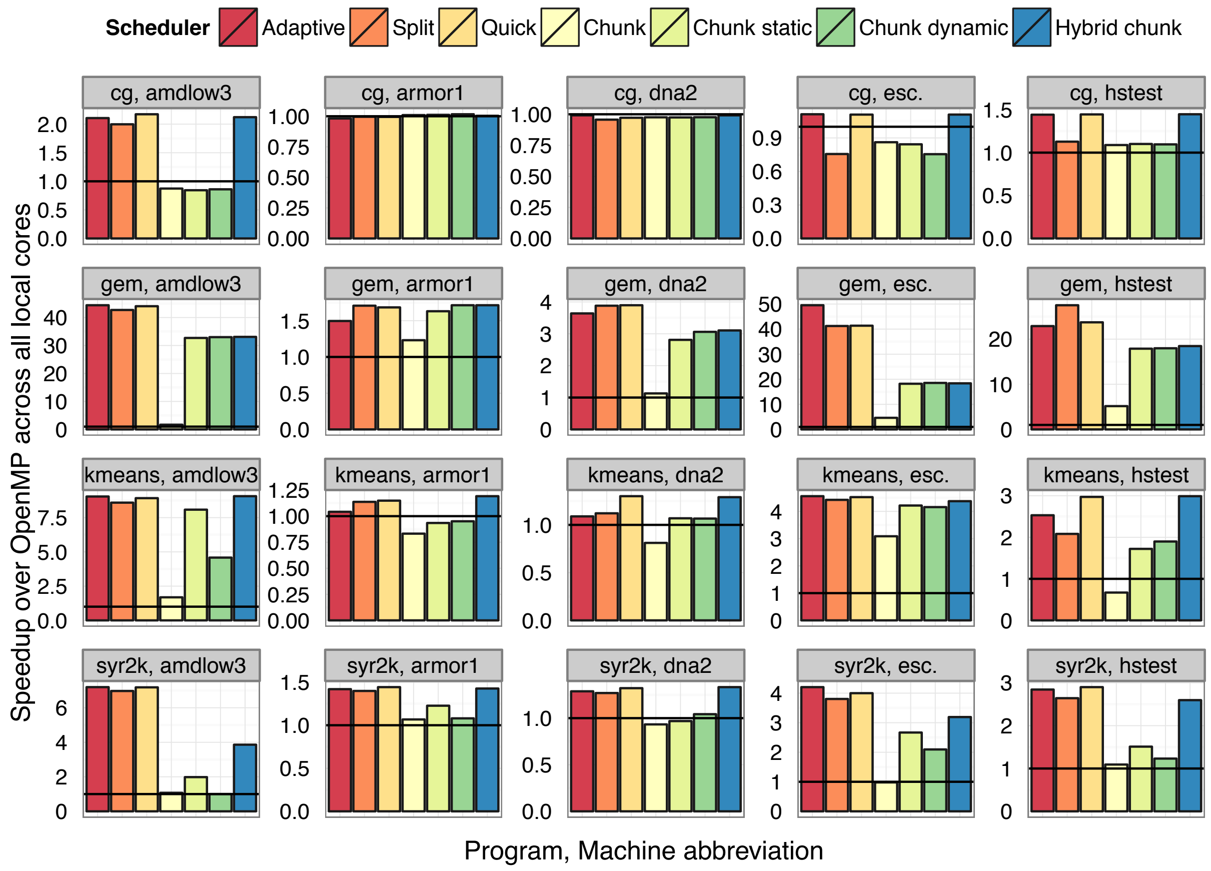


Figure 5.12: CoreTSAR speedup across systems

representative subset (CG, GEM, kmeans, and SYR2K) with problem sizes that fit onto all evaluated GPUs. Figure 5.12 shows results for these benchmarks across all five test platforms. The most prominent feature of the results across systems is the significant change in overall speedup. In particular, amdlow3 exhibits consistently high speedups using the GPU, partly due to the extreme imbalance between its Intel Celeron processor and NVIDIA C2050 GPU. Even CG shows material speedups on amdlow3, as much as $2\times$. More importantly, even though speedup and overall performance shift across the various systems for each benchmark, the distribution of performance by scheduler is similar. Thus, the right CoreTSAR scheduling algorithm is more related to the ap-

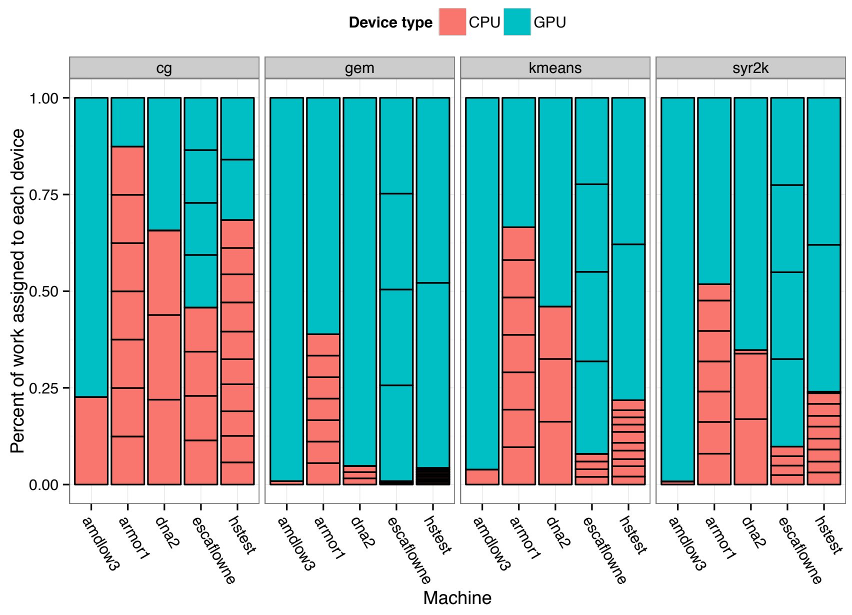


Figure 5.13: Work distribution across devices in each system, where a single device is a single black box, with the Adaptive scheduler

plication than the hardware. Allowing the scheduler to be determined once per region, rather than once per machine. Further, these results show that the default adaptive scheduler is effective across hardware configurations, with only GEM as an issue, as a result of its single iteration. GEM's strong performance on the other devices also showcases the portability of our computed default division of work, which for that application is consistently near the best.

We investigate the actual division of work between different processing elements in Figure 5.13. Each benchmark has a distinct pattern, with CG using mostly CPU cores and GEM using the GPU almost entirely. While the distribution of overall performance in Figure 5.12 shows that the Adaptive scheduler often achieves high performance, the work division shows how differently each machine behaves, and between them how effectively CoreTSAR hides these discrepancies from the user.

5.5.5 Comparison with Blocked Task Schedulers

In order to compare CoreTSAR's scheduling with a state of the art heterogeneous task scheduler, we employ those designed to support blocked task models. Specifically we port three benchmarks (GEMM, kmeans, and Helmholtz) to two freely available implementations of this type of model, OmpSs and StarPU. As with Accelerated OpenMP and CoreTSAR, we use the most straightforward port possible, transforming only the loop regions that CoreTSAR targets. For example, Figure 5.1 lists a literal transcription of the GEMM implementation on OmpSs, calling functions defined in Figure 5.1.

In order to provide an accurate comparison, the CoreTSAR codes evaluated here use the CUDA and C functions created for OmpSs and StarPU rather than using Accelerated OpenMP. In fact these functions were compiled into a single object file with nvcc that was then linked with the CoreTSAR, OmpSs and StarPU scheduling code, thus all three are scheduling over *identical* compute kernels. The OmpSs version was run with the versioning-stack scheduler, to support alternative implementations, as well as flags to allow prefetching and overlapping of data transfers for benchmarks where these offered speedup (slowdown was observed in one case). The StarPU implementations used the "dmda" scheduler with the history-based performance model, trained on at least ten runs before results were collected.

GEMM and Helmholtz run each row of the main outer loop as an individual task. The outer loop

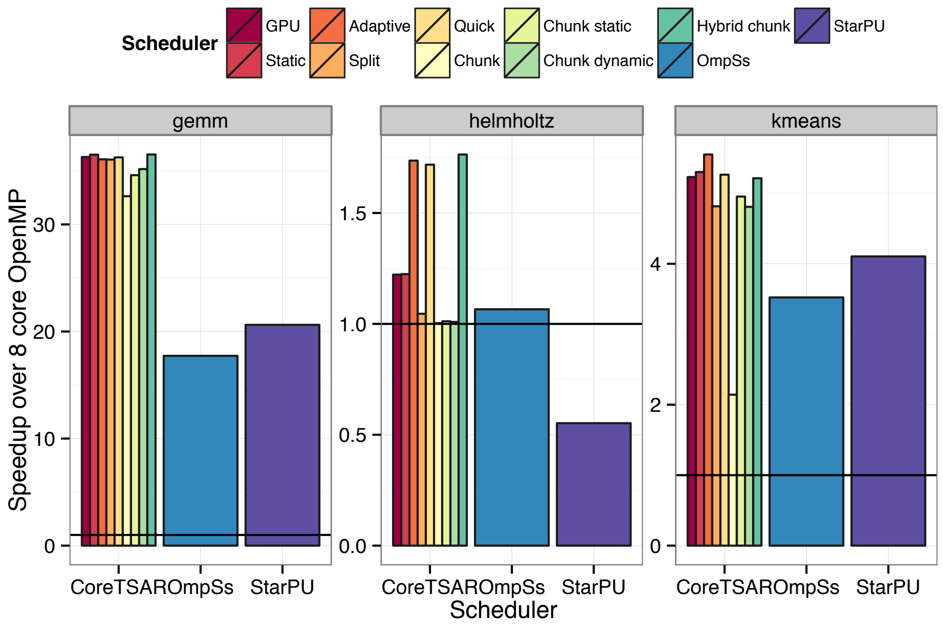


Figure 5.14: Comparison of CoreTSAR with OmpSs and StarPU.

for kmeans is fine-grained, so we block it into chunks of 1000 iterations for OmpSs and StarPU, and also use 1000 iteration chunks as the default for CoreTSAR's *chunk* schedulers although we allow it to adapt at runtime where capable. Each only copies the data necessary for a given task. For example, we only request the three rows necessary for a given Helmholtz row. We also disable CoreTSAR's persistent memory support, since OmpSs does not provide an equivalent feature, though StarPU does.

Figure 5.14 presents the speedup results, calculated as speedup over all 8 CPU cores with the OpenMP dynamic schedule, with OmpSs and StarPU on the far right. While unrelated to the performance comparison, Helmholts shows a performance benefit using GPUs in this case. In truth, the CPU version compiled with gcc is significantly slower ($9\times$) than the version evaluated earlier, while the CUDA and OpenACC versions perform similarly.

Each of StarPU and OmpSs are block schedulers, operating much like our Chunk scheduler, and so we expect that they would perform similarly. The expectation holds holds for Helmholtz, wherein OmpSs performs almost identically to CoreTSAR's Chunk scheduler with StarPU trailing by roughly 50%. In kmeans and GEMM each performs quite differently, with OmpSs and StarPU outperforming Chunk on kmeans and being heavily outperformed by it in GEMM.

While the computation and data transfers are nearly identical between the schedulers, the performance of CoreTSAR using one of the granularity adapting schedulers is consistently higher due to reduced overheads. Since CoreTSAR never explicitly creates the individual tasks, it never pays the cost to allocate or to initialize them, only paying for the aggregate tasks it runs. This benefit is especially noticeable in GEMM where CoreTSAR is $3\times$ faster than OmpSs and $2\times$ faster than StarPU scheduling the same work. Given the ability to adapt task granularity at runtime, all three would yield similar performance. It may be worthwhile to consider adding CoreTSAR, or a similar task-splitting design, to each of OmpSs and StarPU to reduce overhead for this type of computation.

5.6 Conclusion

This chapter has presented the design and implementation of Task-Size Adapting Runtime (CoreTSAR). We make four primary contributions: the design of our scheduler for adaptive scheduling across arbitrary numbers of heterogeneous devices; an implementation and optimization of that design; the design and evaluation of seven adaptive scheduling policies; and our evaluation across four scientific codes, 15 benchmark kernels and a side-by-side comparison with OmpSs and StarPU. We achieve speedups as high as $3.74\times$ over the best performance that uses all cores and a single GPU. When compared to the original CPU performance on 8 cores, we achieve as much as $180\times$ for one benchmark. Further, we present an extension to our memory management system that transparently aligns matrices during mapping, improving performance in some cases by as much as $2.5\times$. These results clearly demonstrate the benefits to be gained from runtime adaptation of task sizes and motivate the addition of a co-scheduling interface, such as the `hetero()` clause that we propose, to Accelerated OpenMP.

Chapter 6

Locality Mapping with AffinityTSAR

6.1 Introduction

Our previous chapters discuss the methods we use to address aspects of the dimensions of locality and capability in isolation. This chapter presents our efforts to address both of these spaces as a coherent whole, combining approaches from both CoreTSAR and SyMMer into a unified model and extension.

Given a system with four memory nodes, associated with four cores each, and four GPUs, each with a distinct address space and tens to hundreds of cores programmed like a BSP [93] cluster, some of which cannot directly inter-communicate due to multiple PCI-E bridges, what would you call that system? Perhaps a better question is, what would a programmer from ten years ago have called it? Systems such as this are effectively clusters of miniature constellations (in a node). The aggregate complexity of this *workstation* is akin to programming constellation-based supercomputers prior to the advent of multicore processors.

One of the biggest challenges in programming these miniature constellations is managing the locality and distribution of application data throughout the system. In the case of GPUs, the challenge comes in the form of requiring data to be either replicated locally on the GPU or pinned to a physical memory address on the host and accessed at far greater cost. CPUs, on the other hand, can access any system-level memory coherently, but not all of it efficiently. If allocation and initialization are done without regard to the NUMA layout of the system, the extra contention and loss of memory bandwidth can be extreme. Neither of these issues is insurmountable, but each requires significant programmer effort and non-trivial expertise. Worse yet, any static solution to the problem reduces the performance portability of the application, due to explicit ties to memory or coprocessor layouts and types.

We propose to address these issues with a single coherent extension of the data-mapping facilities of OpenMP 4.0 or similar models (such as OpenACC). The extension specifies associations between data and computation along with multi-device worksharing, allowing the runtime system to migrate, to replicate, or even to re-shape the data required by a given region without modification or invalidation of compute code. These new facilities can mitigate, or even correct, NUMA locality issues while simultaneously allowing automated worksharing and load-balancing of regions across *multiple* coprocessor devices with potentially independent address spaces.

Our primary contributions are as follows:

- The design of a memory-association interface extension for OpenMP
- A set of runtime optimizations enabled by the extension, including memory migration in NUMA systems, data sub-setting, replication and re-shaping as well as uniform treatment of CPU and GPU target resources
- A prototype implementation of our extension as a C++ runtime library, called AffinityTSAR
- A thorough evaluation of the extension's performance across CPU and GPU resources under different memory policies and layouts

The chapter is composed as follows. Section 6.2 discusses background and related work. Section 6.3 describes the design of our extension, including the interfaces for data association and work partitioning, scheduling mechanisms, and memory management. Discussion of our prototype runtime library, AffinityTSAR, follows in Section 6.4. Section 6.5 presents and discusses our evaluation of the prototype, and Section 6.6 concludes and discusses future directions of our work.

6.2 Background and Motivation

In this section, we briefly present modern heterogeneous node architectures and how data distribution can affect performance in this environment. We further discuss some current and historical approaches to the problem.

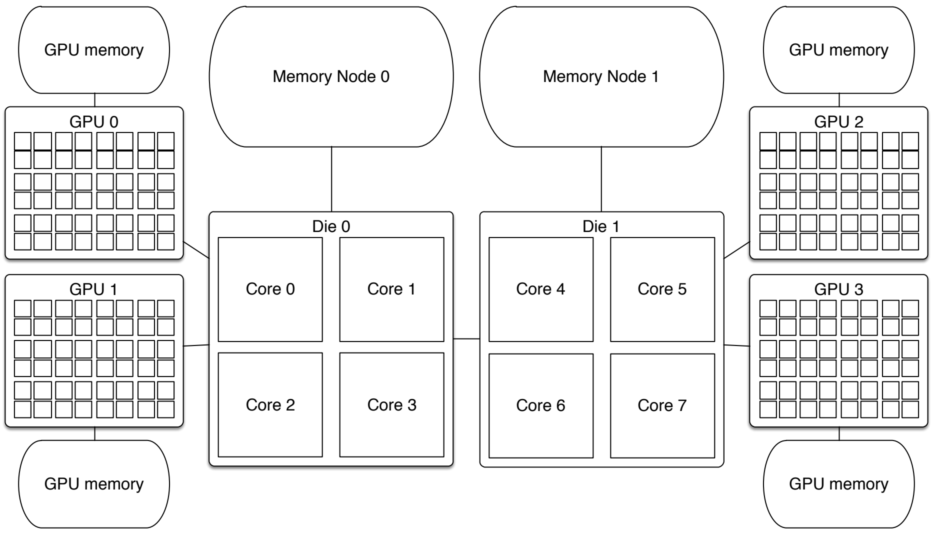


Figure 6.1: Layout of a test system, dual quad-core CPUs and four GPUs

From/To	Node 0	Node 1	GPU 0	GPU 1	GPU 2	GPU 3
Node 0	12,407	8,704	3,851	3,855	3,785	3,758
Node 1	8,963	17,920	3,795	3,771	4,032	4,096
GPU 0	3,460	2,926	97,469	4,890	N/A	N/A
GPU 1	3,460	2,922	4,890	97,619	N/A	N/A
GPU 2	2,833	3,971	N/A	N/A	97,630	4,890
GPU 3	2,820	4,108	N/A	N/A	4,890	97,636
Interleaved	15,639	14,298	3,454	3,238	3,429	3,457

Table 6.1: Memory bandwidth in MB/s from each memory node to each CPU/GPU node

6.2.1 A Modern Heterogeneous Node Architecture

Heterogeneous systems, especially those heterogeneous in memory access latency, have been a known issue for many years. That said, the complexity of a single node of a distributed memory system has classically been relatively low, where each node consists of a number of homogeneous cores with only one or two coherent memory nodes and a relatively small difference in access latency (though even in these cases significant heterogeneity in performance can be observed [85]). Today, however, distributed memory clusters are being built of nodes that are, in and of themselves, composed of multiple distributed memory nodes of different architectures and access requirements. Consider the layout of an example node shown in Figure 6.1, one that we will use for evaluation later in Section 6.5. The system contains two Intel Xeon CPU dies, each with four cores and independent memory controllers connected across a QPI link. While these Intel Xeon CPU dies by themselves form a modestly complex, cache-coherent, NUMA system, they are not the only computational devices in the system. Each of the dies is also connected to an independent PCI-E bridge connected to two NVIDIA Tesla C2070 GPUs.

Considering this node as one would a NUMA design, Figure 6.1 shows the memory bandwidth between each set of memories in the system. First, note that the local memory bandwidth of node 0 is approximately 30% lower than the local memory bandwidth of node 1. This is the result of an imbalance in the number of memory DIMMs in each memory node, node 0 has one less than node 1 and loses bandwidth as a result. For the same reason, the performance of interleaving memory across both memory nodes is limited by memory node 0 and is lower overall than using memory node 1 alone from the local cores.

Looking at the GPU bandwidths, the story gets even more muddled. Transfers between two memory regions on the same GPU operate at nearly 100GB/s, dwarfing anything else. Transfers between two GPUs by direct peer-to-peer copy operate at only 5GB/s, faster than transfers to or from either host memory but slower than transfers between two segments of host memory. These numbers are especially interesting since, in a system where the GPUs are split across two PCI-E bridges, not all GPUs can directly communicate with all others. The peer-to-peer transfers are only possible between GPUs on the same bridge, requiring transfers back to the host and subsequently out to the other GPU to get the same result. Finally, the transfer performance from a GPU to or from system memory is affected about as much by locality as the CPU bandwidth is by the same effect.

In effect, this *node* is a constellation composed of two four-core nodes connected to two independent pairs of BSP clusters in the form of the GPUs. Clusters of nodes like this are best conceived of as distributed memory systems, composed of shared memory systems containing other smaller distributed memory systems. To handle nodes of this complexity requires applications that can be distributed at multiple levels of a hierarchy, both inside and outside of a node, placing tantamount importance on memory layout, locality, and load-balancing to achieve high performance.

6.2.2 Memory Association and Distribution

Many approaches have been taken to handle data distribution in distributed and shared memory systems. We categorize the majority of existing models in terms of three groups: distributed-array models, shared-memory affinity models, and task-associative models. These groups are discussed in greater depth in Section 2.5. We then describe the approach that we take with our extension.

Distributed-array models: This category includes a variety of models that specify the distribution of arrays across devices or nodes of a system and express computations in terms of these aggregate data types. Examples include HPF [64] and Co-Array Fortran [76] and their descendants XcalableMP [77] and Chapel [31]. In each case arrays are defined as an aggregate construct that spans one or more, potentially distributed, memory regions. That definition includes annotations on how that data is to be distributed. To use HPF as an example, the `DISTRIBUTE` directive specifies data distribution in terms of arrays that can be copied either entirely by block or cyclically in each dimension. These arrays can also be aligned or have "templates" applied to ensure that data that tend to be accessed together are nearby to one another. Given the distribution of the data, computational tasks are launched such that they are bound to the location of the data, often with the same syntax or mapping construct used to align the data to devices.

These models enable data locality with computation and possess the advantage of handling both inter-node and intra-node data distribution together. Because the data distribution is explicitly split across distinct memories already, adding GPUs to these models is a natural extension as well. However, because the data is distributed explicitly across devices and computation is bound to the data after the fact, load-balancing is hindered or even prevented entirely by the data distribution. Although Chapel investigated dynamic domain mappings to address this issue, the binding of data is static for the most part and results in a static binding of computation to that data. For homogeneous CPU-based systems, this issue is relatively minor. For heterogeneous systems where the performance of different devices is difficult to approximate beforehand, it is far more serious.

Shared-memory affinity models: On the other end of the spectrum, shared-memory affinity models extend shared-memory programming models like OpenMP with soft memory mappings to improve memory locality within the shared space. These models do not define explicit mappings of data to devices, but rather associate data to threads. Some examples of this include ForestGOMP [25, 26] and UPMLib [74]. Even some approaches explored in the Linux kernel for automatic page migration could fall into this category, mapping data to threads by tracking remote-access faults in the system. This approach solves the issue of load-balancing, freeing the system to map computation and migrate data as appropriate. The downside is that because the mappings between data and computation or between data and devices occurs at the level of hints, migrations, and complete buffers rather than at the level of individual elements, the mappings are insufficient to incorporate distributed memory accelerators naturally or to allow multi-device worksharing.

Task-associative models: Gaining popularity with the rise of task-parallel models and accelerators, this group includes models like OmpSs [27, 28, 37] and StarPU [12]. Rather than binding data to devices (or using hints and migration), task-associative models define each unit of work as a "task" with an explicit set of input and output data regions on which to operate. With that specification, extension to accelerators is simple because the data requirements are clearly expressed as part of the model. Furthermore, load-balancing is possible by distributing fine-grained tasks across available target devices. However, these models eliminate worksharing across parallel loop regions and instead express all parallelism as tasks. As a consequence, their performance depends highly on the granularity of the tasks that they are given. If the user defines tasks that are of inappropriately small, or large, size for any of the devices being targeted, performance suffers. In addition, because the dependencies are specified at task granularity, the runtime system possesses no mechanism by which to alter that granularity dynamically, even when it is both possible and appropriate.

Our approach: OpenMP 4.0 [78] and related efforts, such as OpenACC [7], extend the classically shared-memory model of OpenMP with mapping constructs that are capable of targeting local distributed memory devices. However, they also must specify data movement explicitly from the host shared-memory space to a region in a specific target device. Our previous work, CoreTSAR [87], presented an extension to support multi-device coscheduling and worksharing for accelerated OpenMP and a minimal memory association syntax to support the distribution of data and work across distinct address spaces. CoreTSAR lacks a general-purpose association interface, support for data-distribution scopes outside of data-parallel regions, and support for managing data locality. Leveraging CoreTSAR's extension and scheduling system as a starting point, we propose to create a more general memory-association syntax for OpenMP to enable both worksharing of data-parallel regions across distinct address spaces and dynamic mapping of memory in hierarchical shared-memory spaces as a single unified extension.

6.3 Design

Here we present the design of our model and the extension we propose to OpenMP to enable it. Specifically, we present the design a memory-association interface that will simultaneously tackle the issues of locality in NUMA architectures and memory management for accelerators. Both issues stem from the same root, the distribution and locality of data in a complex memory hierarchy, allowing us to treat them simultaneously by providing more comprehensive information on memory access to the runtime system and scheduler. The existing design and model of OpenMP imposes a number of constraints on the design of our extension and runtime system, specifically we must:

1. Maintain source-code correctness, whether the directives are honored or not; elision must

result in a correct serial program;

2. Prevent the distribution or scheduling of work from altering the correctness of an application given correctly applied directives;

3. Maintain data consistency in system memory at the boundaries of partitioned regions;

4. Limit the need to alter the body of a region.

Following these rules allows us to provide an extension that fits with existing OpenMP practice while enabling significant new capabilities.

Our proposed extension consists of two main parts: the partitioning clauses and the data-association clauses. While the data association system is the primary contribution of our work, it depends on the design of our partitioning interface to function, so we discuss the partitioning clauses and model first.

6.3.1 Partitioning

The partitioning interface is comprised of the two clauses and the directive listed in Figure 6.2. Rather than adding a new directive, we find it more natural to introduce new clauses for use with the existing `parallel` and `target` directives.

The `devices` clause specifies the devices to be targeted by the region, currently supporting CPU, GPU or ANY options for type along with an optional list of device IDs to use for the region, allowing for subsets of the available devices of the given type. By default the system will attempt to use all devices of the given type available in the current OpenMP "place," which may be a subset of the available resources created by a parallel region with the `proc_bind(spread)` clause. When `devices` is specified on a parallel directive the parallel region will create one *host thread* per specified device, set the device Internal Control Variable (ICV) of each thread to the device assigned to it, and create a data scope on that device. Specifying target devices on a non-target construct like `parallel` is not immediately an intuitive decision, but it eases the creation of per-device threads and simplifies the management of data by preserving the existing assertion in OpenMP 4.0 that a given symbol will have at most two meanings per thread in a given scope, a host value and a device value. When `devices` is specified for a `target` directive, one thread per device is created as with the parallel case, but each immediately invokes the body on its assigned target device, so the host threads are never user-visible.

The `partition` clause defines how, and what, work is to be partitioned among the available devices. When a partition clause is bound to a combined loop construct, specifically a `target teams distribute parallel for` directive or a `parallel for` directive it requires only the schedule argument. As with a traditional work-shared loop, the range and stride of iterations is derived from the associated loop definition and directly partitioned by privatizing the iterator variable.

```
// Partitioning clauses
partition([schedule[: loop-expr]])
devices(type[, id-list])

// Binding directive
#pragma omp bind_partition(partition-id)
```

Figure 6.2: The partitioning constructs

Partition can also be bound to a bare `parallel` or `target` directive, but requires the `range-expr` specification in this case. The form of the specification is:

```
    part-id=start; part-id cmp. end; part-id+=stride
```

This expression should conform to the OpenMP requirements for countable loops, and has the additional requirement that `start` and `end` must be variables rather than expressions or literals. As with the loop-bound case, the range and stride of the loop expression are used to determine the range of work items to be partitioned across devices. The main difference from the loop-bound case is that instead of running the body of the construct once for each iteration specified by the expression, it will only be run once per "task," where a task may be comprised of an arbitrary number of iterations between one and the number specified by the expression. The values of the `start` and `end` variables provided in the loop expression are privatized by the runtime and replaced with appropriate start and end values expressing the range of the task assigned to the current device. Parallel-bound `partition` directives may also be nested to specify multi-level partitioning of work and data. While nesting is not required, using nesting allows a natural mapping of levels of computation to levels of the system hierarchy. This in turn increases the information available to the runtime and compiler for optimization of work partitioning and data movement.

The `bind_partition` directive is used to inform the runtime that the associated loop is being partitioned by the range associated with `part-id`. This is never actually required, since the range values can be used directly, but allows the runtime to gather more accurate timing information and potentially perform additional optimizations. In particular, `bind_partition` is useful when there are multiple loops within a region that are split by barriers or reductions, rendering timing across the full region useless for scheduling purposes. By adding the `bind_partition` directive to one or more of these loops, the timing is taken specifically from them, and used in place of full-region timing.

The scheduler specified with the `partition` construct may be any of runtime, static, dynamic, or adaptive. Runtime and dynamic work in an identical manner to the schedules of the same name defined for use with the existing OpenMP schedule clause. The static schedule is extended to partition statically across devices, but to do so based on their peak performance rather than guaranteeing an equal split. Finally the adaptive schedule is based on our prior work on CoreTSAR [86, 87], where it was originally named "dynamic," and partitions the range based on its prediction of the average runtime of an iteration on each device. Given that prediction, adaptive assigns work to all devices such that the deviation in runtimes between devices will be minimized. We have previously found that the adaptive schedule is a good low-overhead alternative to dynamic, but it also has another advantage. In order to maintain the semantics of a parallel region with a partition construct, the body of the region must run exactly once per thread. This requirement arises from the fact that the region may contain barriers, reductions or other constructs that prevent the body of the region from being re-evaluated safely. In this environment, chunk-based schedulers cannot be used, but since adaptive only needs to run the region body once, even while load-balancing, it is a minor issue.

6.3.2 Memory Association

Our association interface directly extends the existing map clause by adding the option of different association types, the option to specify an association to a partition ID and adding the concept of "padding" around a partition. We also propose new clauses for the update directive, and a basic set of API functions to support manual handling of complex data structures where necessary.

```
// Extended map clause
map(map-type[, association-type]: list)
  map-type: to | from | tofrom
  association-type: array | segmented | indexed
  array var: var[var-spec][[var-spec]][...]
    var-spec:  [array-sec.[, part-id[, stride-var]]]
    array-sec: [pad-pre|][start]:[end][|pad-post]
  segment var: var[array-sec, segment-array[, part-id]]
  indexed var: var[array-sec, index-array[, part-id]]

// New update clauses
padding_to(list)
padding_from(list)
padding_update_nearby(list)
// API functions
void * omp_target_allocate(size_t size);
void omp_target_memcpy_to(void*  dst,
                          size_t dst_offset,
                          void*  src,
                          size_t src_offset,
                          size_t size);
void omp_target_memcpy_from(void*  dst,
                            size_t dst_offset,
                            void*  src,
                            size_t src_offset,
                            size_t size);
```

Figure 6.3: The association constructs

The map clause maintains all of its original binding properties, and all existing map clauses will continue to work as expected with the default association type "array," portrayed on the left of Figure 6.4. The array type maps rectangular sub-regions of multi-dimensional arrays, allowing one dimension to be partitioned per level. The new additions to this type are the optional part-id, padding and stride values. If a `part-id` is specified, then the mapping is not replicated on all devices but divided among them based on the partitioned range associated with the ID. A partitioned mapping is roughly equivalent to using a `partition` clause on a parallel region, then a `map` clause in each of the children with the range values specified as the start and end values for the range to map to or from the device. For multi-dimensional arrays of known shape, such as C statically sized arrays of the form `int a[n][m]`, the transformation can be handled automatically by a compiler, but dynamically-allocated unidimensional arrays are also frequently used as multi-dimensional arrays in C. Unfortunately, there is no point of indirection where a compiler can safely transform accesses to these structures. The optional `stride` parameter, which must be a non-constant variable as with the start and end parameters to the partition clause, handles that case. It takes the distance between two elements in the dimension on which it is specified. For example, given a dynamically allocated array representing a 2D matrix with rows of length 10, the row stride would be 10, and the column stride would be one. Specifying the stride for any dimensions that require it allows the compiler and runtime to privatize the variable when the associated data is re-shaped, correcting indexing while maintaining the correct serial elision of the program as well.

Padding pre and post values can be used to specify extra required input on either side of a range in order to support computations such as stencils that require values as input that are not expected to be copied as output. Specifying padding in this manner also gives the runtime system the information necessary to perform efficient padding exchange by pulling the appropriate amount of data from each device back to the host to serve as padding for nearby devices with padding_from and copying that data back to the appropriate target devices with padding_to. The special clause

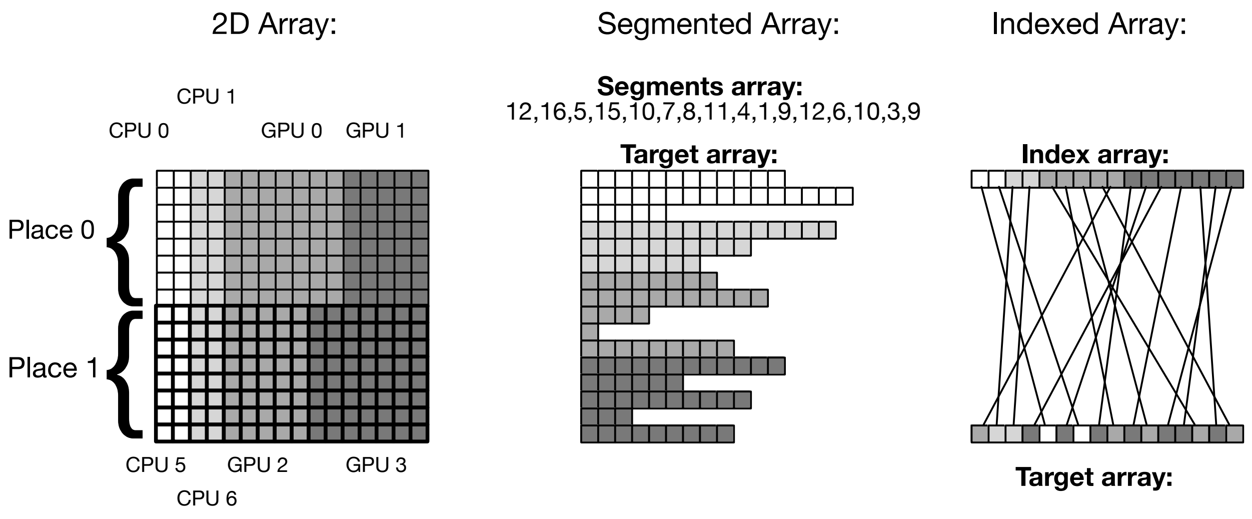


Figure 6.4: The three pre-defined partitioning types

padding_update_nearby is also proposed to allow direct exchange of padding between devices without requiring the data to be copied back to the host as an intermediary.

In addition to the default array type, we also propose two new association types to handle data with indirect shapes. The first of these we refer to as a segmented array, center in Figure 6.4, wherein an array is broken into a set of segments indexed by a second array such that, logically, element 5 of segmented array `a` is composed of the range of values `a[segments[5]]` through `a[segments[6]-1]`. A common example of this layout is a compressed sparse row matrix, where row indices are stored in a secondary array, but the pattern appears in a number of applications. The indexed type is also based on indirection, but accesses an array of the form `a[index[i]]` resulting in a potentially arbitrary layout of input elements. In either case, both the target array and the segment or index array are partitioned, and the segment or index array may be re-written by the runtime to allow for correct partitioning of the base array. This mechanism could also be used to flatten the accesses to indexed types completely given compiler support, though we have not tested it that way.

Finally, some arrangements of data are irregular, require deep copies, complex output handling or similar that cannot be fit into the general associations. For these, we propose the addition of a basic set of API calls that are sufficient to implement manual transformations for any desired distribution. The `omp_target_allocate()` function is self explanatory, but the `memcpy` prototypes are non-standard. They are split into two functions in order to handle systems that cannot determine, based on address, what direction the copy needs to proceed in, thus requiring an explicit specification of whether to copy to or from the target device. The offsets are included for a similar reason, specifically to support partial copies to and from devices that use opaque handles to represent buffers on the host, such as OpenCL.

6.3.3 Example

The GEMM code in Figure 6.5 illustrates our overall approach. The first construct on line 1 is a parallel partition construct that does not target any non-CPU device, rather it uses the places interface clause `proc_bind(spread)` to create one thread per "place," which would usually be a memory node or socket, and divide the cores of the system among them. We propose to incorporate accelerators into the places hierarchy such that they are subset as well when `proc_bind(spread)` is used. Note also that the A and C matrices are partitioned by columns, while B is not partitioned. Partitioning an inner loop in an outer construct may be counter-intuitive at first. That behavior is desirable however, since partitioning both loops by binding directly to them would run the inner loop as `i_end - i_start` separate target regions and cause a great deal of unnecessary overhead.

The inner construct on line 6 is much more dense than the first, partitioning the B and C matrices by row, and otherwise preserving the existing shape of all arrays. Having specified the shape of each matrix in the outer parallel region, there is no need to re-specify it, simplifying the inner mapping. These nested directives divide the columns across sockets or memory nodes, and rows across cores and GPUs in or near that socket. The nested specification is also sufficient to allow the runtime system to migrate memory to appropriate sockets dynamically, to replicate or to pack subsets, and even to block loops with a combination of a chunk scheduler and replication.

```
1   float A[i_size][j_size], B[i_size][j_size];
2   float *C = (float*)malloc(sizeof(C[0])*i_size*j_size);
3   int C_stride = j_size;

4   #pragma omp parallel proc_bind(spread)                  \
5     num_threads(omp_get_num_places())                     \
6     partition(adaptive: j_id=j_start; j_id<j_end; ++j_id)\
7     map(to: A[0:z_size][:,j_id], B[0:x_start][0:z_size]) \
8     map(tofrom: C[0:x_start][:,j_id])
9   {
10  #pragma omp target teams distribute parallel for      \
11    devices(OMP_TYPE_ALL,*) map(to: A[:][:], B[:,i][:])\
12    partition(adaptive) map(tofrom: C[:,i,C_stride][:])
13    for (int i = i_start; i < i_end; ++i) {
14  #pragma omp bind_partition(j_id) // Optional
15      for (int j = j_start; j < j_end; ++j) {
16        float sum = 0.0;
17        for (int k = 0; k < k_size; ++k) {
18          sum += A[k][j] * B[i][k];
19        }
20        C[i * C_stride + j] = sum;
21      }
22    }
23  }
```

Figure 6.5: A GEMM kernel extended with our memory-association syntax

6.4 Affinity-aware Task-Size Adapting Runtime (AffinityTSAR)

AffinityTSAR is a prototype runtime library implementing support for the major features of our proposed extension on top of PGI's OpenACC and OpenMP. This section discusses the major components and usage of AffinityTSAR, and the specific policies and optimizations enabled by the extended memory association interface.

6.4.1 Locality and Threading

System topology and locality of memory and devices is one of the most critical factors considered in AffinityTSAR. In order to extract the basic information we employ the hwloc [24] library for system topology discovery. GPUs are also placed in the hierarchy by requesting their PCI addresses through the CUDA interface, and locating the corresponding PCI-E device through hwloc, allowing us to map CUDA device IDs reliably to physical cards and associate them with the appropriate memory node. All cores and GPUs in the system are probed and arranged into a hierarchy inside of AffinityTSAR, wherein each memory node is treated as a "place" in OpenMP parlance, and cores and GPUs are members of these places.

In creating this topology, one of the first questions to arise was how to handle the management of GPUs. Many frameworks and systems reserve cores for the purpose of managing each device, but not all. In order to determine which option AffinityTSAR should take, we test it for ourselves. AffinityTSAR supports both options by either scheduling both a CPU thread and a GPU thread to the same core, or reserving the core entirely for the GPU thread and removing the corresponding

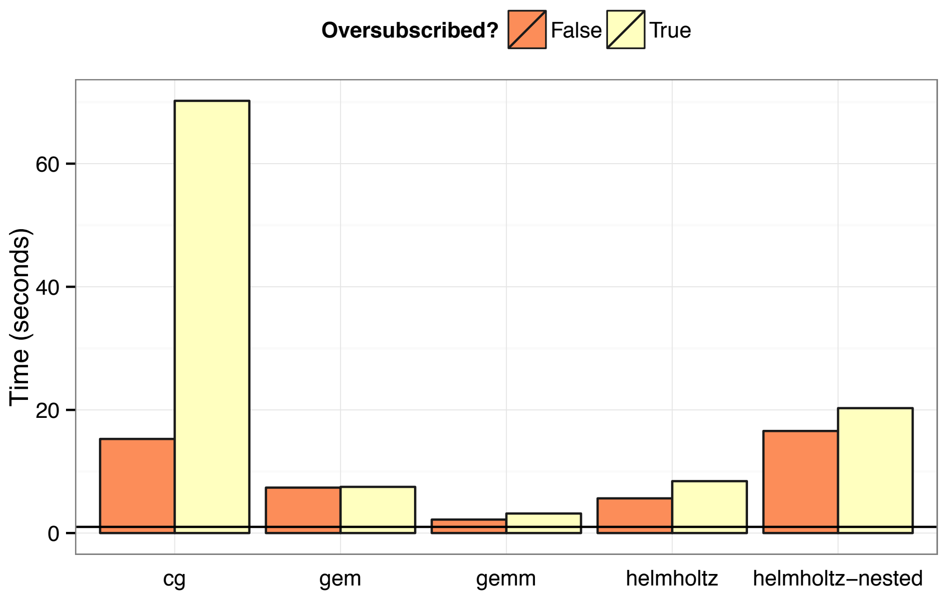


Figure 6.6: Performance of the system with and without oversubscription of GPU control cores.

CPU thread from the topology. Figure 6.6 portrays some of the results that we found when evaluating the choice of whether to load the GPU control cores. The effect on performance is proportional to the number of passes taken by the benchmark being tested, which is why CG with 1,900 passes shows the most effect, but in all cases we have tested, doing computation on the GPU control core produced a net loss of performance between 1% and up to over $3\times$.

6.4.2 Memory Management and Re-shaping

The implementation of the AffinityTSAR memory management system uses an interface directly derived from the designed syntax. The only material difference is that the array type always requires a stride parameter in the library implementation. Aside from that, all three types take the same arguments as in the abstract design and produce a base pointer, offset, length and stride in each dimension for each variable. These details can all be reasonably hidden by a compiler translation layer, but are necessary to allow for arbitrary transformation of the shape of the allocation in which the data is placed.

Each is also designed to support repeated partitioning such that every layer can be a direct reference to the memory space represented by its parent, a subset of the parent using the original memory with offsets, or a subset of the parent by replication. Subsequent children can also have any of these relationships with the current level, allowing things like replication of a data-set onto all sockets then sharing subsets by reference with all threads in the socket.

Given these hooks, we implement a number of memory handling policies for both GPU and CPU devices. For GPUs the main benefit is to be able to allocate only the space necessary for the input subset, possibly with padding to preserve coalesced accesses. A secondary benefit is that since the *shape* of the data is known in advance, it is often possible to handle sub-regions with the efficient 2D copy APIs rather than a slew of smaller contiguous copy invocations.

For CPUs, the range is more varied, and focuses on a set of memory management policies that can be set at each level of partitioning. The most basic is coherent, do nothing to the memory mapping, just assume the operating system will take care of it. Next up the chain is migrate, which determines which pages will be worked on by each selected device, and where they are, and migrates pages to the target if possible. Migrate will only migrate full pages, and only when a registered region is being partitioned at the current level. As a result, migrate is best used for socket-level or coarse grained splits. Migrate-interleaved is a composite policy that combines the behavior of the migrate policy with the extra option to interleave, through explicit user-level re-distribution of pages, memory regions that are not subset and thus cannot be migrated to specific targets. For random-access or globally shared inputs interleaving can be the best option. Finally, there is the replicate option. Replicate treats each device as though it had no coherent access to CPU memory, giving it a local copy of every input, though local copies may be shared for input-only regions. While replicate sounds extreme, and expensive, there are cases where the resulting dense sub-sections result in more efficient accesses than the original, and the replication cost is more than made up for by improvements in access efficiency and reduced contention.

6.5 Results and Evaluation

This section presents an evaluation of our prototype runtime library, AffinityTSAR, in terms of two general factors. The first is the utility of the memory association information for correcting NUMA memory imbalances, and the second is its use as a coscheduling and load-balancing system.

6.5.1 Experimental Setup

We used two systems for our evaluation, both with non-uniform memory and a GPU, but with materially different properties. They are each detailed in Table 6.2. System 1 is the system portrayed earlier in Figure 6.1, consisting of two sockets with four cores each, and four NVIDIA Tesla C2070 GPUs with two attached to each socket. This system is also notable for having imbalanced bandwidth available between NUMA nodes, since node 0 has one less RAM DIMM installed than node 1, further increasing the heterogeneity of the memory system. System 2 contains only one

System	CPU Model	Cores/ die	Mem. Nodes	CPU Dies	CPU RAM	GPU Model	GPU Cards
1	Intel X5550	4	2	2	20gb	Tesla C2070	4
2	AMD 6272	16	4	2	64gb	GTX Titan	1

Table 6.2: Test system specifications

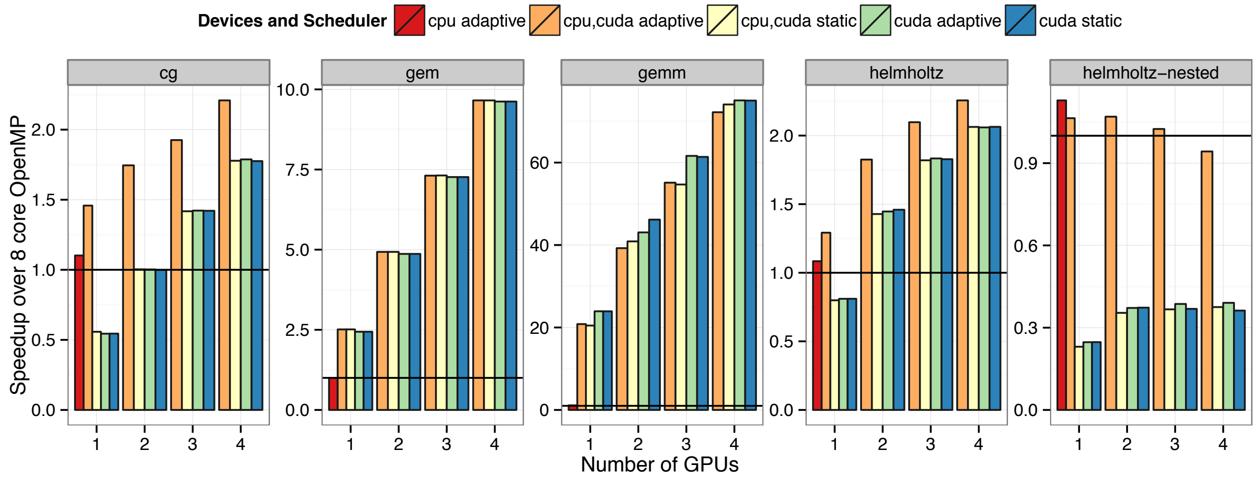


Figure 6.7: Speedup across schedulers and number of GPUs for all computational benchmarks, normalized to 8-core CPU OpenMP performance.

GPU, an NVIDIA GTX Titan, but has two NUMA nodes per socket, for a total of four. The second system is relatively similar to the common GPU cluster design, with one GPU and multiple CPU sockets, and increases nested scheduling complexity since only one of the four NUMA-node groupings is associated with a GPU.

Both systems run a Linux 3.2 kernel with a Debian-based distribution. All benchmarks, described in further detail below, are implemented with OpenACC 1.0 directives[1] and OpenMP, and extended with AffinityTSAR memory and coscheduling facilities. AffinityTSAR and all benchmarks are compiled with the PGI accelerator compilers version 14.1 linked to CUDA 5.5 and directly targeting compute capabilities 2.0 and 3.5. All test results represent a minimum of five runs. We

[1]We use OpenACC here in place of OpenMP 4.0 directives as a result of availability of compilers, AffinityTSAR has no explicit dependency on OpenACC.

include four computational benchmarks in our evaluation, which are discussed in greater detail in Section 2.7: CG, the conjugate gradient benchmark from the NAS parallel benchmarks; GEM, a visualization application for molecular modeling; GEMM, a generalized matrix multiplication kernel from PolyBench/GPU; and Helmholtz, a Jacobi iterative implementation of the Helmholtz equation.

In addition to these, and in order to isolate the effects of the memory policies from issues of coscheduling, we also include an evaluation of the Stream [67] benchmark. No modifications have been made to the bodies of the tuned Stream loops, and this benchmark was intentionally left CPU-only. The loops are simply surrounded by AffinityTSAR constructs used to allow for memory management, migration and replication, allowing us to test our NUMA management policies more directly.

6.5.2 Coscheduling and Performance

To provide a baseline for performance with AffinityTSAR, we start with an evaluation of the overall speedups and scaling achieved for the computational benchmarks with and without scheduling across CPUs and GPUs on System 1. These results are presented in Figure 6.7. These results use the default memory policy and allow multi-level nested scheduling for the benchmarks that express multiple levels of parallelism (GEM, GEMM, and Helmholtz-nested).

Both GEM and GEMM scale almost linearly from one to four GPUs with either scheduler. They are different however in their preference of devices. GEM gains a small amount of performance, on the order of 1-2%, from using CPU cores coscheduled with the GPUs. GEMM, in contrast, actually loses performance by attempting to schedule any work on the CPU cores. The delay in GEMM is partially an artifact of the scheduling algorithm starting from scratch at the beginning of each run, such that the first pass loses performance on the CPU cores with an imbalanced schedule. We may investigate cross-run history in future work to alleviate this issue.

In contrast to GEM and GEMM the two Helmholtz benchmarks gain a significant amount from having the CPU cores available for coscheduling. In fact, the nested version of Helmholtz consistently loses performance whenever it attempts to make use of even *one* GPU. We include this version of the benchmark because it illustrates a common issue with expanding serial or single-GPU codes to a multi-GPU design.

Helmholtz is an iterative stencil code that implements the Helmholtz equation, as such the partitioned region is run repeatedly across buffers of the same data to incrementally refine the solution. The Helmholtz-nested code is structured such that at the beginning of each outer iteration the workload is re-balanced across devices, a 2D region of the input space is copied in, computed on, and the output is merged back into system memory. Unfortunately, this practice is common, and one that was necessitated by the lack of a parallel partition construct in some of our previous work. While it results in giving the runtime the most opportunities to re-schedule, the copy overhead is excessive. The un-nested version, in contrast, uses a parallel partition construct to partition the space once, then runs a number of iterations (20 in this case) *with the same partitioning* before re-balancing again. As the combined figure shows, the overall performance improvement by reducing the frequency of redistributions is significant, reaching over $2\times$ performance where the nested version only loses it.

The major reason for this difference is the reduction in the amount of data-transfer overhead from two factors, reducing the frequency of re-balancing and handling the boundaries of the stencil with padding update constructs rather than a full merge back to the host. To illustrate the effect of spending more iterations doing the cheap boundary transfers rather than re-balancing, Figure 6.8 presents the time to run the Helmholtz benchmark with different numbers of iterations between re-balancing phases as split between computation, input and output time. On this plot, the Helmholtz-nested benchmark would be equivalent to the line at one iteration, where the computation time is less than one eighth of the overall execution time of the benchmark. By partitioning less frequently, and using the efficient padding-transfer interface rather than merging the entire set between iterations, the input and output time drop steadily as more iterations are done between re-scheduling passes until at 20 and higher the data transfer time drops to less than half that of the computation time, settling finally at a tenth when all iterations are run with only padding transfers.

CG behaves somewhat like the optimized version of Helmholtz, though for a somewhat different reason. CG also uses a parallel partition region to amortize the cost of re-balancing, but it does so across four different loops, each of which passes partial results and reduced values to the next. This is a major advantage of the data scoping effect of the parallel partition construct over loop-bound partitions, allowing a great deal of separate computations to proceed, with synchronization, on the same data partitions. As with Helmholtz, the CPUs contribute a great deal of the overall performance of CG, reaching a high of $2.2\times$ speedup over 8-core CPU static with the adaptive schedule. This situation also shows the significant downside of a static schedule for applications

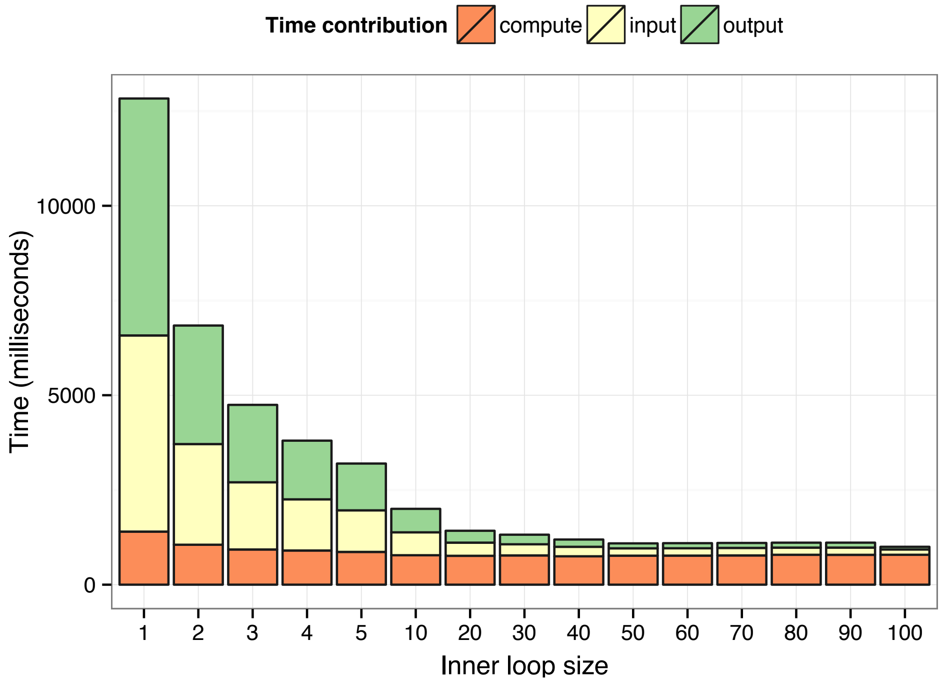


Figure 6.8: Distribution of time for the Helmholtz benchmark across inner-loop sizes.

of this type, where the CPUs end up underutilized throughout the run.

6.5.3 NUMA Policies

As we have discussed earlier in this chapter, coscheduling is but one of the goals of our extension, we also see it as an effective way to increase the effectiveness of the runtime system at mitigating NUMA affinity issues. In order to evaluate that effect, we have tested a number of different policies for managing memory regions assigned to devices by the system. Specifically the policies are coherent maintains the OS default; interleave distributes all allocations across all memory nodes in round robin order at page granularity; migrate uses the memory layout specified by the extension to migrate memory to the local space at the entry into the region; migrate interleaved extends

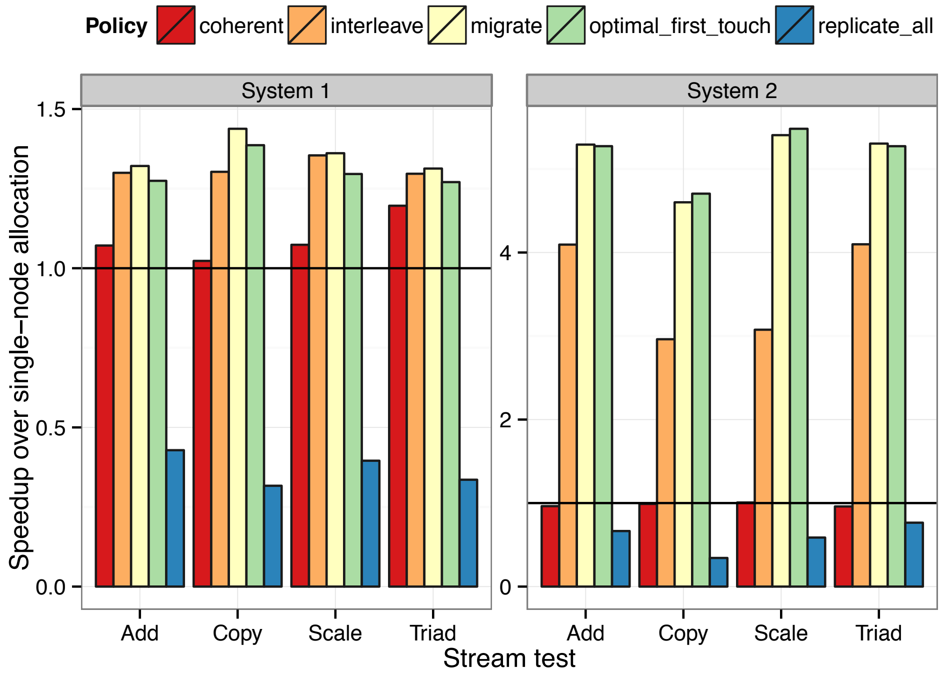


Figure 6.9: Stream benchmark average bandwidth across different NUMA management policies.

the migrate policy to interleave the pages of memory regions that are not partitioned; finally the replicate all policy makes local copies of all partitions for each core.

The Stream benchmark was deigned to exploit the Linux default "first-touch" memory affinity policy by allocating memory, then performing initialization with an OpenMP parallel loop. That way, each page is placed close to the thread that is responsible for working with it, a theoretically optimal layout assuming that all memory nodes have equal bandwidth. Often, the worst way to handle allocations is to initialize all of the memory in a single thread, placing all allocations in one memory node. In order to test our memory management policies, we set up Stream to allocate and initialize all buffers on a single memory node, then use AffinityTSAR for each of its tests. Figure 6.9 presents our results as increase in bandwidth over the single-node baseline, as well as a measure of the explicit first-touch behavior originally built into Stream.

The default “coherent” policy does not move, migrate or otherwise alter any of the allocations, and thus should be the same as the single-node baseline. On System 2, coherent behaves as expected, but on System 1 it is actually faster than the baseline by some amount for all tests. The small improvement with coherent on System 1 is not actually due to the memory policy, but rather to the default binding of threads to cores that AffinityTSAR performs. The interleave option is a material improvement over the baseline in all cases. Adding in migration makes a small difference on System 1, up to 10% on copy, but is very significant for System 2, which has a more complex memory structure, approaching a $5\times$ speedup for the Copy benchmark over the baseline and 25% over the interleaved policy. Where the first-touch policy is occasionally slower than the migrate policy or even the interleave policy on System 1, the differences are due to an imbalance in the performance of each of the two memory node on System 1. Allocating even amounts to each node is not an optimal result in that case. The difference between migrate and first-touch on System 2 is within the margin of error of our measurements. Finally, replicate performs the worst in all stream tests.

CPU-side replication proves far more useful in the case of our CPU-only GEMM benchmark tests, represented in Figure 6.10. When socket partitioning is enabled, meaning partitioning both by sockets and devices in a socket such that all matrices are partitioned in at least one dimension, migrate and migrate interleaved both make a positive difference. With the interleaving effect, migrate-interleaved achieves almost a 50% improvement in performance. More surprising however is that using the replicate all policy with only a one-way partitioning achieves $4\times$ speedup over

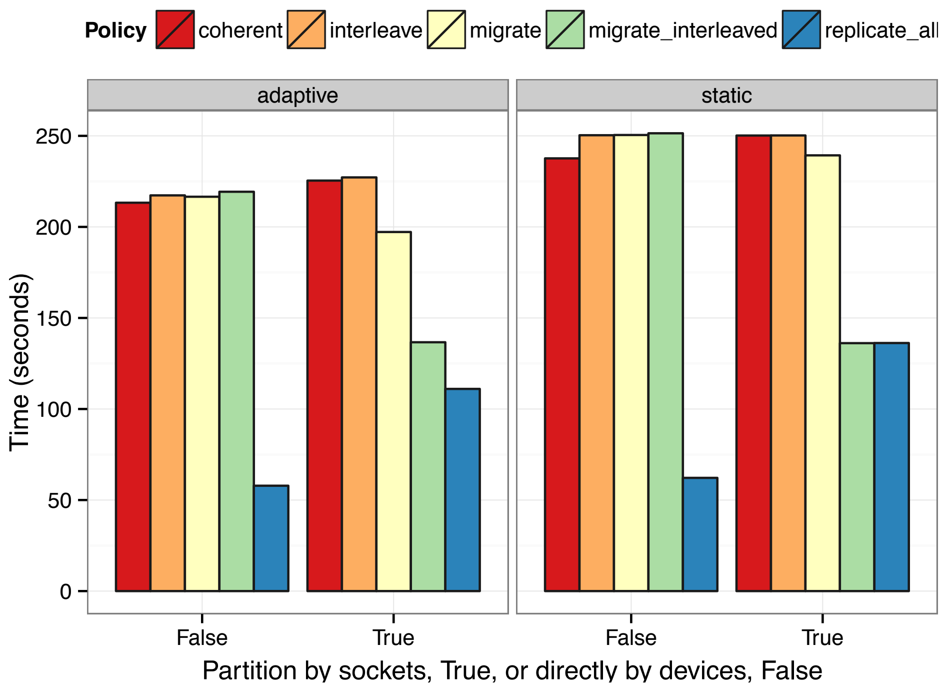


Figure 6.10: Matrix multiply performance on CPUs only across different NUMA management policies.

the coherent case. In effect, this policy results in a tiled loop, and better caching and contention behavior. We did not expect replicate to be an improvement of that magnitude, but our extension presents a natural place to alter parameters such as these for auto-tuning or testing to discover unexpected performance benefits such as this one.

6.6 Conclusion

In this chapter we present the design, and prototype implementation, of a unified extension to manage multiple accelerator devices as well as NUMA affinity in OpenMP. We make four primary contributions: the concept and design of our new memory association interface; the design of a set of runtime optimizations and transformations enabled by that extension; an implementation of the runtime components of the system; and our evaluation of the prototype in terms of both coscheduling performance as well as NUMA affinity correction. Our prototype system achieves as high as a $50\times$ speedup over eight core CPU with four GPUs, and we show a nearly $2\times$ speedup for a previously averse benchmark as well. When applied to mitigating NUMA affinity issues, we also see improvements of as much as 40% in the bandwidth of the Stream benchmark, and greater than $3\times$ performance improvement in the performance of dense matrix multiplication on the CPU with appropriate policies. Given these results, our extension will greatly improve the adaptability and performance portability of codes in the ever more complex nodes of supercomputers as we reach toward exascale.

The next phase of our work in this direction is to investigate the expansion of AffinityTSAR beyond the bounds of BSP-style execution and scheduling patterns. While they are common in scientific applications, and form the basis of parts of AffinityTSAR, some applications do not map well to this model. Perhaps more importantly, some *parts* of a wide range of applications do not, and we wish to investigate ways to integrate AffinityTSAR's design with a task-based or preferably work-stealing system to handle more complex work patterns. In order to do this efficiently, we still wish to avoid the explicit creation of fixed-size tasks, and instead investigate the creation of structures that allow disparate devices to retrieve portions of work for themselves with low overhead. As a first step toward that goal, we need high-throughput concurrent structures that can support the extreme concurrency of the environment, which we will discuss further in the next chapter.

Chapter 7

A Highly Scalable Concurrent Queue for Many-Core Architectures

7.1 Introduction

Multicore architectures have taken over the CPU market, and many-core accelerators and coprocessors, such as GPUs and Intel MIC, are becoming available to all segments of computing. Each new generation contains more cores, further compounding the demands on the scalability of software. That scalability, more often than not, is governed by the cost of synchronization and communication.

Concurrent data structures have become basic building blocks for the new wave of highly parallel applications, providing intuitive abstractions atop the complexities of low-level synchronization and memory coherence primitives. The result can be both increased productivity and, when designed well, performance. One of the most ubiquitous of these is the concurrent first-in, first-out (FIFO) queue.[1]

The concurrent queue has been studied extensively over the last four decades. It has gone through a variety of forms – from infinite-array queues [45, 50] to lock-free queues [69, 92] to advanced distributed lock-free [47, 54] or even wait-free [55, 56] variants. As concurrency has increased, so has the contention on concurrent queues and the cost of synchronization, and frequently serialization, in these designs.

Our goal with this chapter is to characterize the performance requirements and considerations of concurrent queues in the multi- and many-core era and to create a concurrent queue that is tailored for high throughput, even under extreme contention. Our design and evaluation span CPU, GPU and co-processor architectures using the C and OpenCL programming models.

Specifically, this chapter makes the following contributions:

1. A characterization of modern multi- and many-core architectures as targets for concurrent queue designs.

2. The design of a robust, simple, and extremely scalable *blocking* FIFO queue that is based on the above characterization.

3. A thorough evaluation of our queue in OpenCL and OpenMP, including a comparison with several classic and state-of-the-art concurrent queues and demonstrating up to a 2-fold speedup

[1]As we only discuss FIFO queues in this chapter, the term *queue* shall be used in place of *FIFO queue* henceforth.

on CPUs and as much as a 1000-fold speedup on GPUs running more than 1000 concurrent threads.

The rest of the chapter is laid out as follows. Section 7.2 presents the background and setup for our work, including the machine abstraction that we employ to discuss synchronization and threading in OpenCL and OpenMP environments interchangeably as well as the performance and scaling of the atomic operations that make the abstraction possible. Section 7.3 presents the design of our queue and its three interfaces while Section 7.3.5 discusses linearizability [50]. Section 7.4 presents our experimental setup and benchmarks while Section 7.5 presents the results of our experiments. Finally, concluding remarks and future work are presented in Section 7.6.

7.2 Background

In order to discuss the properties of our target architectures in a uniform manner, we first present our abstraction of the concurrency and memory model that we use across devices. This section discusses the abstraction that we employ in this chapter in order to discuss OpenMP on CPUs and OpenCL on CPUs, GPUs and coprocessors all interchangeably along with our microbenchmark evaluation of atomic operations that make this possible across each architecture.

7.2.1 Threading Abstraction

While the threading models of OpenMP and OpenCL are significantly different, they can be reconciled. An OpenCL kernel runs a set of work-groups, each consisting of work-items or, as they are sometimes unfortunately misnamed "threads." We exclusively use the term "work-item" to refer to these throughout this chapter. Work-items are usually a single lane of a vector computation, rather than an independent thread of control. OpenMP presents no observable equivalent to the work-item, though a single iteration of a loop parallelized by an `omp simd` directive would be closest.

OpenCL does have an equivalent to the OpenMP thread however, though it changes from device to device. In NVIDIA GPUs, one thread is a "warp," composed of 32 work-items. In AMD GPUs, a thread is a "wavefront" of 32 or 64 work-items. When run on CPUs, work-items may be either operating system threads or individual lanes of vector calculations as on GPUs. For common CPUs, each thread may be composed of one to eight work-items. The width of the thread-equivalent used in a compiled kernel in OpenCL can be reliably determined based on the OpenCL 1.1 kernel work group info property "preferred work group size multiple," which is what we use in our implementations. To establish consistent terminology, we use the term *thread* to refer to *OpenMP threads* on CPU and Xeon Phi or *independent groups of work-items* in OpenCL. Since work-items within a thread must execute in lockstep, allowing more than one work-item in a thread to interact with a concurrent data structure simultaneously is unsafe. When a thread accesses any queue in this chapter, only one work-item is active.

The additional wrinkle is that OpenCL offers no mechanism to determine the number of threads that actually run concurrently. While a user can request any number of threads, the number that run concurrently can be anywhere from one to the requested number. We add counters as depicted in Figure 7.1 to all benchmarks to count the number of threads that exist before the first thread finishes execution, which is a reliable upper bound on the number of concurrent working threads regardless of the behavior of the OpenCL runtime.

```
void test(unsigned *num_threads, unsigned *present){
    if(atomic_read(num_threads) != 0)
        return;
    atomic_fetch_and_add(present,1);
    run_benchmark();
    atomic_compare_and_swap(num_threads, 0, atomic_read(present));
}
```

Figure 7.1: Design of concurrency detection in OpenCL benchmarks

7.2.2 Memory Model

CPU models like OpenMP depend on cache-coherent shared memory for correctness. The OpenCL standard does *not* provide a sufficiently strong coherence model, or a memory flush that can be used to implement one. The standard states that "there are no guarantees of memory consistency between different work-groups executing a kernel [6]." Writes in different work-groups are only guaranteed to be synchronized at the end of a kernel, and are thus available in subsequent kernels. The standard specifically allows writes to global memory *never* to become visible to other work-groups within a single kernel.

The exception is atomic operations, available since OpenCL 1.1, which are guaranteed to be visible and coherent across work-groups within a kernel as long as all work-groups execute on the same device. Thus, *every* write and *every* read to global memory that is shared between work-groups *must be atomic* to ensure correctness in OpenCL. In practice, some OpenCL devices support a more coherent memory model, but it is not required and several architectures do not.[2]

For consistency, we express all algorithms as a set of abstract atomically coherent instructions. In OpenMP, the atomic reads and writes are standard load and store instructions. In OpenCL, they are implemented with explicit atomic intrinsics.

7.2.3 Atomic Performance

To understand the scaling behavior of current queues, an understanding of the scalability of atomic operations on modern architectures is required. Figure 7.2 shows this scalability in terms of throughput for two operations commonly found at critical points in concurrent queue designs, the Compare-And-Swap (CAS) and the Fetch-And-Add (FAA). The throughput for the CAS operation is further decomposed into the number of CAS operations attempted, and the number that succeeded in changing the target value. The scalability of the operations on each of the three CPU systems matches common knowledge, FAA is faster than successful CAS at high contention by as much as $10\times$. Neither operation scales well however, losing throughput with additional threads on both Intel systems and gaining only marginally on the AMD system. The higher cost of CAS has been considered acceptable in order to guarantee progress in concurrent algorithms for many years, the extent of it being limited by the comparatively small number of threads that could be run

[2]NVIDIA GPUs present a weak coherence model, but offer a fence/flush through the PTX instruction *membar.gl*, but this is not standard OpenCL and must be used carefully. AMD GPUs have similar instructions at the ISA level but inline assembly only accepts the intermediate CAL language, which has no equivalent.

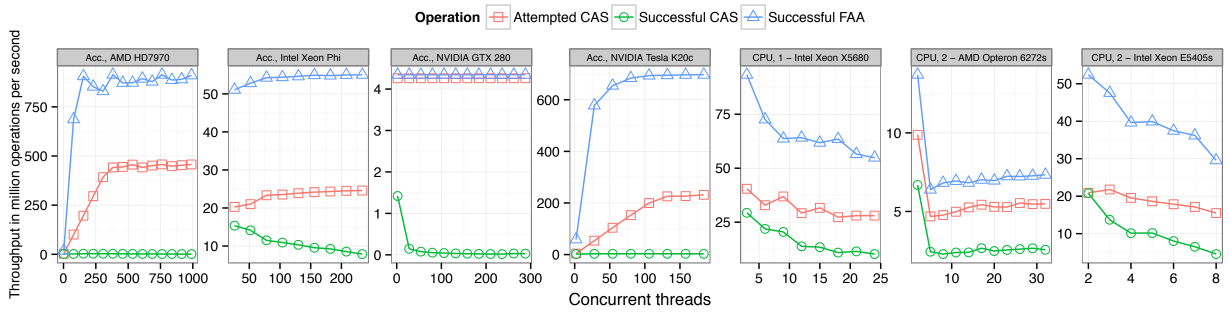


Figure 7.2: Atomic operation throughput across processors and thread counts. Each thread attempts 1,000,000 contended atomic increments either with FAA or CAS. Successful CAS represents only the CAS operations that succeed in updating the value.

concurrently on CPU systems.

On the GPUs and Intel's Xeon Phi the picture is completely different. FAA throughput scales up dramatically as more threads are added, and attempted CAS operations increase as well. Successful CAS throughput does not increase however. At worst, the difference between successful CAS and FAA expands to $600\times$ lower throughput with 1000 threads on the AMD 7970. Operations such as FAA are executed completely in the memory controller without any chance of failure or retry, extra operations being queued there for execution. On modern hardware, a single memory location can be incremented by FAA *once per cycle* in this fashion. CAS operations can also be executed there, but because failure requires program logic to retry, they must be executed, then go through a full round-trip back to the processing core before another attempt.

7.3 Design and Implementation

The main goal of our design is to create a queue with as little overhead and as much concurrency as possible while maintaining linearizability. Given the significant throughput advantages of FAA over CAS, our goal becomes to produce a linearizable concurrent queue without the need for contended CAS operations in the common case. To accomplish this, we propose a queue with two distinct interfaces, similar to those offered by communication libraries (e.g., TCP sockets), each with different waiting characteristics: (1) a high-throughput blocking interface and (2) a low-latency non-waiting interface.

7.3.1 The Queue Structure

Our queue's structure, represented in Figure 7.3a, is relatively simple, containing head and tail counters as unsigned integers along with arrays of items and IDs. For simplicity we use unsigned integers as the values, but extensions for arbitrary data are trivial.

In order to handle integer rollover correctly, a real concern as 2^{32} queue operations can complete in a matter of seconds on GPUs, we include a `#define` for the maximum ID value based on the queue size. The maximum is selected such that when the head or tail roll over to zero, the ID value will do the same. The `MAX_ID` value must be at least double the value of `MAX_THREADS+1` and the sum of `MAX_THREADS` and `QUEUE_SIZE`, the `MAX_DISTANCE`, must be less than half of the maximum value representable by the unsigned integer type storing head and tail. All values in the data structure should be initialized to zero.

```c
1   /* Defines */
2   #define MAX_ID (UINT32_MAX/(QUEUE_SIZE*2))
3   #define MAX_DISTANCE (QUEUE_SIZE \
4                         + MAX_THREADS)
5
6   /* Macros */
7   #define GET_ID(X) ((X / QUEUE_SIZE) * 2)
8
9   /* Structures */
10  typedef union {
11    uint64_t combined;
12    struct {
13      uint32_t head, tail;
14    };
15  } pair;
16
17  typedef struct {
18    union {//anonymous pair
19      uint64_t combined;
20      struct {
21        uint32_t head, tail;
22      };
23    };
24    bool closed;
25    uint32_t items[QUEUE_SIZE];
26    uint32_t ids[QUEUE_SIZE];
27  } queue_t;
```

(a) Definitions and Structure

```c
1   int enqueue(queue_t *q, uint32_t item) {
2     if (atomic_read(&q->closed) != 0)
3       return CLOSED;
4     uint32_t ticket = atomic_add(&q->tail,1);
5     uint32_t target = ticket % q->size;
6     uint32_t id = GET_ID(ticket);
7     while(atomic_read(&q->ids[target])!=id){
8       if (atomic_read(&q->closed) != 0)
9         return CLOSED;
10      backoff();
11    }
12    atomic_write(&q->items[target], item);
13    atomic_write(&q->ids[target],
14                 (id + 1) % MAX_ID);
15    return SUCCESS;
16  }
```

```c
1   int dequeue(queue_t *q, uint32_t * p) {
2     if (atomic_read(&q->closed) != 0)
3       return CLOSED;
4     uint32_t ticket = atomic_add(&q->head,1);
5     uint32_t target = ticket % q->size;
6     uint32_t id = GET_ID(ticket) + 1;
7     while(atomic_read(&q->ids[target])!=id){
8       if (atomic_read(&q->closed) != 0)
9         return CLOSED;
10      backoff();
11    }
12    *p = atomic_read(&q->items[target]);
13    atomic_write(&q->ids[target],
14                 (id + 1) % MAX_ID);
15    return SUCCESS;
16  }
```

(b) Blocking Interface

```c
1   int enqueue_nb(queue_t *q, uint32_t item) {
2     if(atomic_read(q->closed) != 0)
3       return CLOSED;
4     uint32_t ticket = atomic_read(&q->tail);
5     uint32_t target = ticket % q->size;
6     uint32_t id = GET_ID(ticket);
7     if(atomic_read(&q->ids[target]) != id)
8       return BUSY;//next slot not ready
9     if(atomic_cas(q->tail,
10                 ticket, ticket+1) != ticket)
11      return BUSY;//CAS failed, return
12    atomic_write(q->items[target], item);
13    atomic_write(q->ids[target],
14                 (id+1) % MAX_ID);
15    return SUCCESS;//element enqueued
16  }
```

```c
1   int dequeue_nb(queue_t *q, uint32_t * p) {
2     if(atomic_read(q->closed) != 0)
3       return CLOSED;
4     uint32_t ticket = atomic_read(q->head);
5     uint32_t target = ticket % q->size;
6     uint32_t id = GET_ID(ticket);
7     if(atomic_read(&q->ids[target]) != id)
8       return BUSY;//oldest not ready
9     if(atomic_cas(&q->head,
10                 ticket, ticket+1) != ticket)
11      return BUSY;//CAS failed, return
12    *p = atomic_read(q->items[target]);
13    atomic_write(q->ids[target],
14                 (id+1) % MAX_ID);
15    return SUCCESS;//element dequeued
16  }
```

(c) Non-waiting Interface

Figure 7.3: Structure and interfaces to the queue with the volatile keyword removed for clarity; All "atomic_*" calls map to the corresponding atomic intrinsic

7.3.2 The Blocking Interface

Our blocking functions are represented in Figure 7.3b.[3] While developed independently, our blocking interface could be considered a refinement of the CB-queue or the Gottlieb queue. It differs from each of these, however, in that it has been explicitly designed to support inspection of the queue's status and access through a *separate non-waiting interface*. Our blocking functions begin by acquiring a ticket for their current transaction by atomically incrementing their respective counters on line 4. A ticket serves to select both the target location, by being modded by `QUEUE_SIZE` on line 3, and the ID for the transaction, which is the number of times the algorithm has passed

[3]Note that the OpenMP and OpenCL implementations of all of our interfaces are identical save for the addition of memory qualifiers in the OpenCL version. In fact, our evaluation uses a single source version of all queues for both OpenMP and OpenCL tests, simply compiled with different compilers.

through the entire queue's length, by use of the `GET_ID()` macro, plus one in the dequeue case, on line 4. Once the target ID is equal to the transaction ID the current thread effectively holds a lock on that element of the queue and leaves the loop. A value is then either added to or copied from the queue, as appropriate on line 12, and the ID safely incremented to preserve consistency across rollover on lines 13 and 14, which also frees the next transaction on that slot.

Subsequent, or concurrent, calls to the blocking interface receive unique target addresses and ID combinations without retries or waiting, thanks to the FAA. As long as the queue is not full, each enqueue can complete with a constant total of four atomic operations. When the target item is busy, it blocks on the loop at line 7 checking for closed status and backing off as appropriate.

Since a primary motivation of our design is producer-consumer applications, we expect threads to wait on an empty or a full queue regularly. Once all items that will exist have been completely enqueued and dequeued, there must be a way to inform the threads blocked in dequeue to exit. The `closed` value in the queue is used for this purpose. Setting `closed` to true causes any enqueue or dequeue that is blocking to exit immediately with the status `CLOSED` and all subsequent calls to immediately return with the same value. Closing our queue is equivalent to closing a communication channel like a socket or file descriptor.

7.3.3 The Non-Waiting Interface

Figure 7.3c presents our non-waiting interface. Fundamentally, the non-waiting functions are inverted versions of the blocking functions. Rather than immediately reserving a ticket, which could require them to block, the non-waiting functions simply read the value. Having read a ticket, they check whether the current target item is ready on line 7. If the ID value indicates the item is busy then acquiring an item at this time would require blocking, so `BUSY` is returned immediately. If the ID matches, the thread attempts to acquire the associated target by incrementing the counter with the CAS on line 9. If the CAS fails, the non-waiting function returns `BUSY`, otherwise it has successfully acquired a ready target so it completes its operation and returns `SUCCESS`.

While we implemented the blocking interface completely without CAS, to implement the non-waiting functions in that way is infeasible. Specifically, to atomically acquire a specific ticket a conditional atomic or transaction, such as CAS, Load-Linked Store-Conditional, or optimally a compare-and-add, is required. Using an unconditional FAA in place of the CAS would get a ticket, but with no guarantee that it would be the ticket that had been checked ahead of time.

We avoid the use of the terms "wait-free," "lock-free" and "non-blocking" in this section. While these functions do not wait, calls to either `enqueue_nb()` or `dequeue_nb()` may fail an arbitrarily large number of times due to another thread blocking in a critical region, preventing us from claiming any such progress guarantees.

7.3.4 The Status Inspection Interface

Much like the CB-queue, our blocking interface does not support returning "full" or "empty" states. While they are not required for a correct concurrent queue, these states are often used to simplify the detection of completion in a concurrent algorithm. Rather than re-designing the algorithm to address this weakness however, we design a separate interface that provides checks for these states as well as the number of waiting threads on either end of the queue. The main difficulty in implementing this functionality is that head and tail *cannot* be directly compared. At any point one, but not the other, might have rolled over causing the less-than or greater-than relationships to be reversed.

We address this case by establishing the maximum absolute distance possible between head and tail and checking to see if the current distance is greater than that maximum. If this happens, it must be because one counter has rolled over and the distance should be calculated across the rollover point. The maximum distance between head and tail in our queue is the sum of the queue length and the maximum number of threads that are allowed to interact with the queue concurrently. If the maximum distance is less than half the maximum value representable by the counter, single-counter rollover can be reliably detected in this fashion. For a 64-bit unsigned integer, the sum of the queue's length and the maximum concurrent accessors must be less than or equal to $2^{63} - 1$, which we believe is a reasonable limit. If more is required, the size of the counter should be increased.

7.3.5 Linearizability

In order to provide a proof of linearizability, we must first define the semantics of our target data structure. Based on instruction ordering our algorithm models a concurrent FIFO queue. When return states, such as empty, full, closed and busy, are included in the requirements for linearizability however, our states do not match. We give our queue the semantics of a *channel queue*: a queue that models a double-sided communication channel, such as is presented by file descriptors and sockets, that can return success, closed, busy, empty or full. If a channel queue is in the closed state then all functions will return closed. If a non-waiting function cannot complete without blocking, busy is returned. All other cases model a concurrent FIFO queue, allowed only to return success, full or empty. This semantic is more common of concurrent queues in production than the traditional model's restriction to empty, full and success, and is modeled by the interface of the standard BlockingQueue class in Java as well as the interface to the concurrent WaitingQueue class proposed for inclusion in the C++1y standard.

Using the techniques and definitions presented by Herlihy et al. [50], we model access to our concurrent queue as a history h. That history is a potentially infinite series of invocation and response events, representing the beginning and end of calls to functions defining our interface. Any response in h is necessarily preceded by a matching invocation in h but it is valid for an invocation in h to remain pending, lacking a response, at the end of h. Events are said to be ordered in h only if the response of an event e_1 precedes the invocation of an event e_2 and this relation is denoted by $e_1 <_h e_2$. Any pair of events that cannot be compared in this way is said to overlap, and thus may be ordered arbitrarily with respect to one another. The history, h, is linearizable if the strict partial order can be fixed into a total order $\rightarrow_h$ such that the specifications of the object are preserved.

Any history that can be produced by our implementation can be associated with a history mapped onto an auxiliary array of infinite size. Using this auxiliary array, our algorithm guarantees that every enqueue, blocking or non-waiting, will monotonically increase the values of the tail counter and thus insert elements consecutively from beginning to end in the array. In the same way, our dequeues monotonically increase the value of head and consume elements consecutively from beginning to end. Thus all items are dequeued in the same order they were enqueued, or are overlapped. Any element that is added is accounted for, and cannot be removed until it is acquired. Acquisition can only happen in order, preventing any items from being skipped or dequeued before being enqueued. All interleaving between the blocking and non-waiting interface are also in-order, as they acquire and interact with the queue using the same ticket and turn mechanism. Given these, the only source of non-linearizable behavior possible is from multiple operations waiting on the same target item with the same ID. Given our invariant that the `MAX_ID` is greater than double `MAX_THREADS`, this case cannot occur, as ids are not recycled until the queue has been passed through at least `MAX_ID` times. Given that invariant, even if a queue of length one were to have the maximum number of threads waiting on it, both ordering and fairness are preserved between those accesses.

The above sketches a proof of the key invariant for concurrent queues, that if enqueue(x) $<$ enqueue(y), where x and y are the values enqueued, then dequeue(x) $<$ dequeue(y) or dequeue(x) and dequeue(y) overlap. To simplify reasoning about the ordering, our functions "take effect" at specific points between their invocation and response, but as with any algorithm employing critical sections, no single instruction serves as the universal linearization point. Operations are considered to take effect on the status of the queue, observable only through the `get_distance()` function and its siblings, after committing to the addition or removal of an element by acquiring a ticket with either FAA or incrementing CAS. All enqueue and dequeue operations are ordered in the sequential history by their increment of the ID associated with their target item. Any temporary discrepancy in queue structure between invocation and response is protected by the critical region formed between ticket acquisition and ID increment.

7.4 Experimental Setup

In order to perform our evaluation across a wide range of modern hardware, we have created a version of each queue using both OpenMP and OpenCL. This section will discuss the evaluated queues, our benchmark designs and the hardware evaluated.

7.4.1 The Queues

In addition to our own, we include implementations of two traditional lock-free queues, the TZ and MS queues, the Flat Combining (FC) queue, and the LCRQ. All queues are implemented to store 32-bit unsigned integers and where memory allocation would normally be necessary use a non-blocking concurrent free-list of appropriately-sized objects that is pre-allocated before each test. The same free-list mechanism is used on both CPUs and GPUs for consistency.

Our MS-queue implementation is directly derived from the source code used in the original MS-queue publication [69]. It has been modified minimally to support thread-based rather than process-based parallelism and the memory model presented by OpenCL. The TZ-queue has been faithfully re-implemented using the algorithm and optimizations described in the paper that proposes it [92]. The flat combining queue is based on the authors source but reimplemented in C/OpenCL from the original C++. Lastly, the LCRQ is based on the pseudocode in the publication that proposes it [71][4]. Our LCRQ implementation deviates in two key ways from the original pseudocode, it includes the spin waiting optimization proposed in the paper, and uses 32 rather than 64 bit values. The value size is changed to allow the algorithm to function on devices that support 64-bit but not 128-bit CAS operations. We evaluated the 32-bit version against a 64-bit version of the algorithm and found that the throughput remains within the range of measurement error for all cases.

The OpenCL and OpenMP implementations of each queue share the same source, with only memory location qualifiers, atomics and memory synchronization primitives differentiated through C macros. For all fixed-length queues, the queue length was set at 65,536 elements for the purpose of our evaluation, separate tests with varied sizes did not reveal significant correlation with performance except when using very small sizes, so these results are elided.

7.4.2 Benchmarks and Methodology

These queue implementations are evaluated across a pair of microbenchmarks. The first is a traditional matching enqueue and dequeue benchmark. All threads execute a loop containing an enqueue, a call to some work, a dequeue, and another call to work. The work between each queue operation is comprised of 100 iterations of addition and multiplication on a value read from

[4]We did correct one error in the pseudocode, line 45 should compare (safe,idx,val) rather than (safe,h,val) as the original states, the text description in the original paper agrees with this modification.

and stored back to positions in global memory determined by the value last received from the queue. This work is sufficient to avoid a single thread running through multiple operations without interference, and decreases the performance of the highest throughput implementations by approximately 10% compared to a version without work[5]. Our second benchmark is based on an imbalanced producer/consumer pattern. One in every four threads only enqueues, and the other three only dequeue, these operations are also separated by the same work as in the first benchmark. Both benchmarks are configured to perform as many operations as possible in five seconds and report the number of successful operations. We selected five seconds after running a round of tests ranging from two seconds to a minute and a half per data point and finding that anything over three seconds is sufficient to overcome variance effects across our target platforms.

The OpenMP implementation ends the test by creating an extra thread that sleeps and sets a *done* value, stopping the test after the specified time. OpenCL offers no such mechanism, neither the extra thread nor the sleep. To get around this, we assign one thread to execute a loop performing mathematical operations on its registers for approximately five seconds. Since the number of operations required changes based on the device, the test and sometimes the queue under test, as a result of register usage changes, our run-scripts automatically tune the number of iterations such that each test runs for between 4.95 and 5.5 seconds on all OpenCL platforms. The downside to this approach is that we lose one potential thread, but with throughputs that range up to three orders of magnitude, evaluation using a fixed time rather than a fixed number of operations is essential.

[5]Some implementations, including LCRQ, perform *better* with the work than without it.

Device	Cores/ device	Threads/ core	Max. threads	Max. achieved
GPUs/Co-processors				
AMD HD5870	20	24	496	140
AMD HD7970	32	40	1280	386
AMD HD7990(one die)	32	40	1280	1020
Intel Xeon Phi P1750	61	4	244	244
NVIDIA GTX 280	30	32	960	960
NVIDIA Tesla C2070	14	32	448	448
NVIDIA Tesla K20c	13	64	832	832
CPUs				
2xAMD Opteron 6272s	16	1	32	32
4xAMD Opteron 6134s	8	1	32	32
2xIntel Xeon E5405s	4	1	8	8
Intel Xeon X5680	12	2	24	24
Intel Core i5-3400	4	1	4	4

Table 7.1: Target hardware platforms

7.4.3 Devices

Table 7.1 lists the devices used to conduct our experiments, along with their core counts, the number of thread contexts that can be loaded concurrently on each core, and the maximum hardware threads on the device. The maximum threads listed in the table is the theoretical maximum, and in the case of GPUs is not always achievable due to limitations on available register space. The maximum achieved column lists the largest number of concurrent threads available to our tests, not all queues make it to those values but none make it above. All test systems run Debian Wheezy Linux on a 64-bit 3.2.0 stock kernel. NVIDIA devices use driver version 313.30 and the CUDA 5.0 SDK for OpenCL. AMD GPUs use the AMD APP SDK version 2.8 for OpenCL and the FGLRX version 9.1.11 driver. The Intel Xeon Phi card uses the MPSS gold update 3 driver and firmware. OpenMP tests were compiled with the Intel ICC compiler version 13.0.1 with optimization level 3 and inter-procedural-optimization turned on.

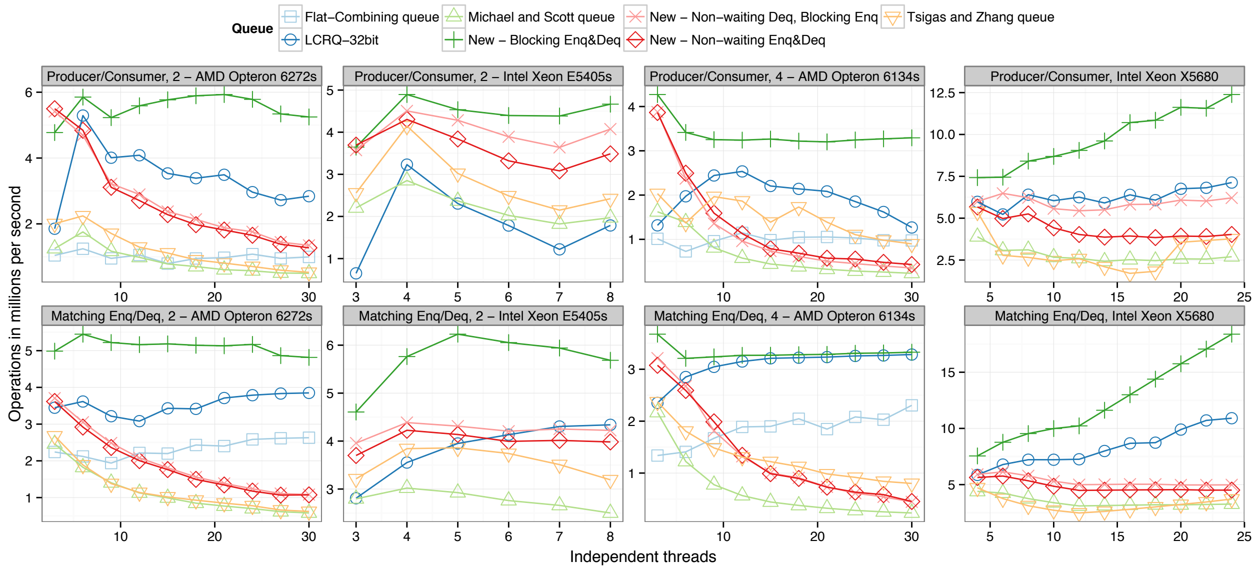


Figure 7.4: Throughput on each CPU across thread counts and benchmarks

7.5 Evaluation

We evaluate all queues across all hardware discussed above, with the exception of LCRQ on AMD GPUs which do not support bitfields or 64-bit atomic CAS. Since our queue presents two interfaces, we present three different configurations for it. Each is labeled in the figures as “New -” followed by which enqueue and dequeue functions it uses for all enqueues and dequeues in the test. The three configurations are the two homogeneous configurations, paired sets of blocking or non-waiting interface calls, plus a version that uses the non-waiting dequeue with the blocking enqueue. We expect that the most common use-case would be to use the blocking interface for all but one, or perhaps a small number, of threads that use the non-waiting interface to detect algorithm completion, which is best represented by the blocking results.

7.5.1 CPU Performance

The CPU results based on these tests can be found in Figure 7.4. Each CPU is tested from two threads up to the maximum number of hardware threads supported by the system. In multi-socket systems threads are spread in round-robin fashion across dies using the Intel OpenMP "scatter" affinity policy. While the multi-socket systems tend to maintain or lose throughput as threads are added, the single socket Intel Xeon X5680 gains throughput with each additional thread. This is due to the fact that the single CPU has only one memory controller, which allows atomics to be completed without out-of-die coherence overhead.

In terms of the individual queues, in almost all cases the highest throughput comes from our blocking interface, followed by the LCRQ. The TZ and MS queues fare poorly in general across each of the CPUs, their performance degrading with each additional thread due to the increasing CAS retry overhead. On the AMD devices and the Xeon X5680, the FC-queue performs materially better than the classic lock-free variants for the matching enqueue/dequeue benchmark. The FC-queue even gains performance with additional threads on the AMD devices thanks to its comparatively low coherence overhead.

LCRQ's performance on the AMD systems reveals an important characteristic of its design. In the matching enqueue/dequeue test it scales well, performing nearly as well as our blocking interface up to 32-cores. The producer/consumer benchmark, on the other hand, shows LCRQ's performance degrading sharply as more threads are added. This is due to retries and memory initialization overhead caused, not by CAS, but by LCRQ operations skipping slots by marking them unsafe. Whenever an operation times out, as is common in our imbalanced producer/consumer benchmark, the item reserved by that operation is marked unsafe, and it retries. Each retry forces the matching operation on that item to retry as well. Eventually, the retries cascade until the CRQ is closed, forcing initialization of a new CRQ by all threads attempting to enqueue at that time. We employ the optimizations proposed to minimize this behavior, specifically spin waiting before marking a slot unsafe and employing a high starvation cutoff for enqueues, but still observe the problem. The Intel X5680 does not observe this behavior because of those optimizations, but they are insufficient for the multi-socket systems. This condition could be avoided in LCRQ if it were allowed to wait indefinitely for a matching enqueue, but that would make it blocking, and can actually produce deadlocks in the algorithm, since dequeuers might not be aware of the need to move to a new CRQ.

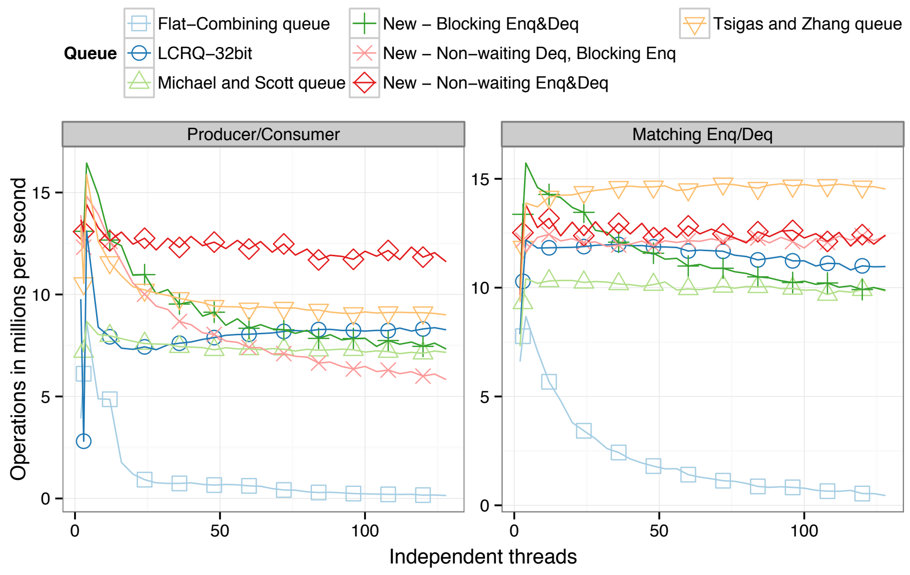


Figure 7.5: CPU performance when heavily oversubscribed, results of tests running from two to 128 threads on a four core Intel CPU

7.5.2 Effects of Oversubscription

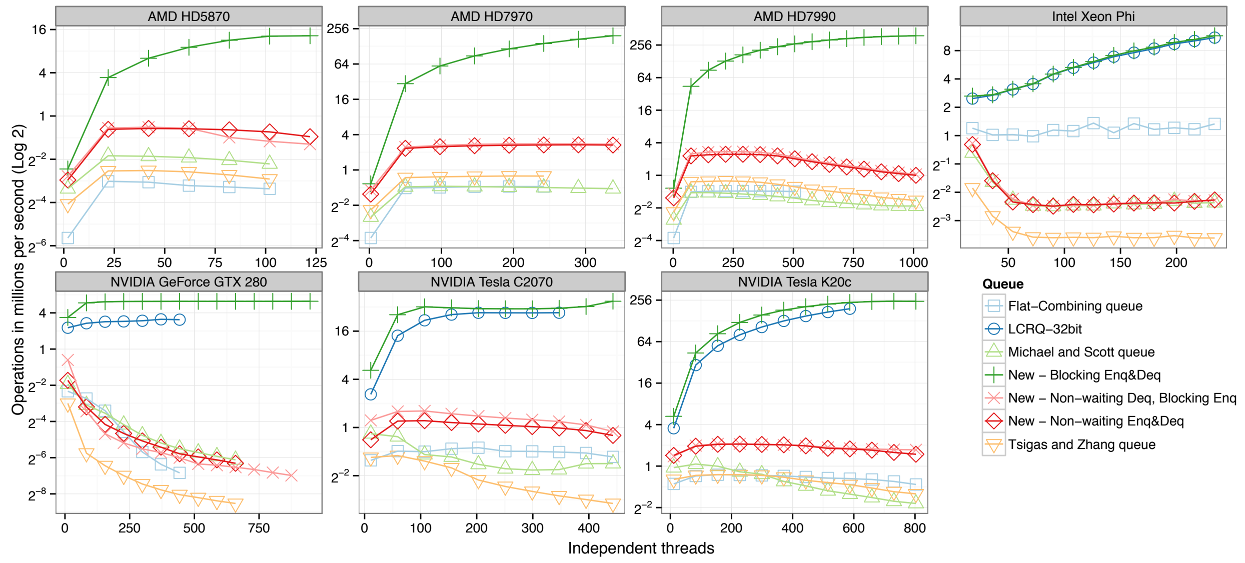
In order to evaluate the effect of oversubscription on throughput, a traditional concern with blocking designs, we tested all queues with thread counts from two to 128 on a four-core CPU in Figure 7.5. All queues include a thread yield as part of their back-off routine, immediately allowing another thread to be scheduled in its place.

As has been shown in other recent work [71], the FC-queue suffers greatly from oversubscription as a result of the combiner being scheduled out frequently. The lock-free queues, MS, TZ and LCRQ on the other hand perform quite well in this test, as expected since this is the environment for which they are designed. Both the MS and LCRQ designs maintain their performance across the full range. On the other hand the TZ-queue and our non-waiting interface tend to perform better than either by between 10 and 75%.

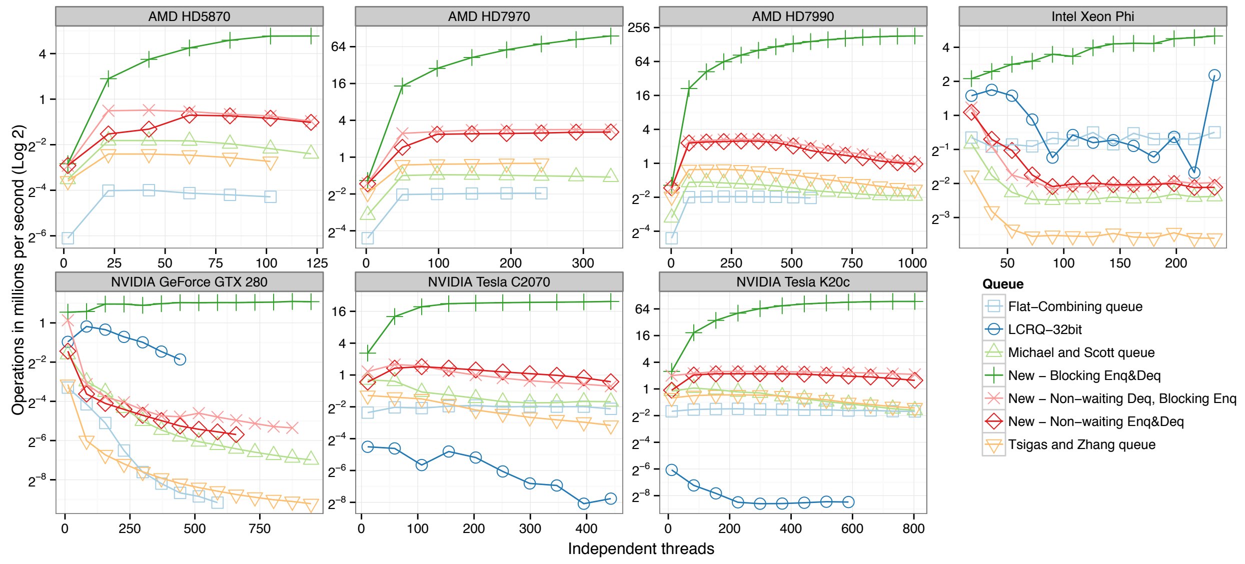
Finally the blocking interface does lose performance as more threads are added, but not so much as might be expected from a blocking design. Since the blocking is extremely fine-grained, and the potential concurrency extremely high, the blocking interface actually outperforms the MS-queue and maintains 50% of its maximum throughput with $32\times$ more threads than hardware thread contexts.

7.5.3 Accelerator Performance

This section presents throughput results with the same benchmarks across seven many-core accelerator architectures in Figures 7.6a and 7.6b. Unlike the CPU results, the range in performance on



(a) Throughput on each accelerator for the weak-scaling matched enqueue/dequeue benchmark



(b) Throughput on each accelerator for the producer/consumer benchmark, of every four threads, one is producer the other three are consumers

Figure 7.6: Accelerator benchmark results

the accelerators requires us to use a log-scale for the bandwidth axis on our plots.

The first important difference between the accelerators and CPUs is the sheer number of thread contexts the accelerators support. Even the smallest, the AMD 5870, hosts 140 concurrent thread contexts for most benchmarks, more than four times as many as the CPUs. Recall that these numbers are measured in threads, not OpenCL work-items, for the number of work-items multiply the threads on AMD GPUs by 64, and NVIDIA GPUs by 32 to get the full number. The two largest go far higher, with the 7990 reaching 1020 concurrently loaded threads, and the K20c hosting 832 for a total of 65,280 and 26,624 work-items respectively. The Phi device runs the OpenMP benchmark source from the CPU tests, so its 244 threads are standard OpenMP threads.

Matching Enqueue/Dequeue Results

The enqueue/dequeue results on accelerators (Figure 7.6a) scale more like a single-socket CPU than a multi-socket system. Since the accelerator cores share a single memory controller, this is expected. The material difference from the CPUs is that each additional thread increases performance noticeably for our blocking interface. On the 7990 the performance scales from 0.585 million operations per second on two threads to 380 million operations per second on 1019 threads. Overall, the 7900 displays a $650\times$ increase in throughput for a roughly $509\times$ increase in the number of threads[6]. Similarly, the K20c attains $256\times$ higher throughput with $415\times$ more threads. The cache-coherent Intel Xeon Phi coprocessor scales somewhat less than the GPUs, going from 0.963 Mops/s to 11.462, for a more modest but still significant increase in throughput of $12\times$ for roughly

[6]The super-linear increase in throughput is not due to any super-linear property of the algorithm, but rather to the fact that the GPU tends to run in a lower performance state when under-utilized.

$120\times$ more threads. The exceptions to the rule in terms of scalability are the GTX280 and C2070 NVIDIA GPUs, whose atomic implementations are less mature, and as a result only scale to a fraction of the throughput of the others.

By far the best performing lock-free design across the accelerators is the LCRQ. On the Xeon Phi its performance is nearly indistinguishable from that of our blocking interface. The NVIDIA implementations do not scale to the full number of threads due to LCRQ's high register usage, but for the thread-counts supported the throughput is quite high. LCRQ's highest performance on the K20c, at 623 threads, is 201.848 Mops/s, only 16% below the throughput of our blocking interface with the same number of threads.

The contentious-CAS-based queues, FC-queue and our non-waiting interface, tend to lose performance as the number of threads increases and the rate of successful CAS operations drops. The fastest of these, the TZ and our non-waiting design, only achieve 0.342 and 0.994 Mops/s respectively on 1019 threads on the AMD 7990, or $1,112\times$ and $383\times$ lower respectively than the blocking interface in the same test. The K20c results are similar, with TZ performing 0.386 Mops/s and our non-waiting performing 1.357 for differences of $635\times$ and $181\times$ respectively. FC performs similarly to these traditional designs on the GPUs, but achieves $5\times$ better throughput on the cache-coherent Phi, where its cache friendly design offers material benefits. While it is not lock-free, the non-waiting interface of our queue tends to outperform its counterparts in this space on these architectures, probably due to the lower number of instructions per operation.

Producer/Consumer Benchmark Results

Results for the producer/consumer test are presented in Figure 7.6b. As expected, the producer consumer test shows roughly 50% lower throughput across the board due to using 50% less enqueuers than dequeuers. All queues are affected by the imbalance roughly equally, except LCRQ.

The change in LCRQ results is most visible on the Xeon Phi, where rather than being nearly a match for the blocking interface, it drops to the performance of FC-queue after only 75 threads. Though LCRQ's throughput is variable, it never reaches half of the throughput of the blocking interface in this test on Xeon Phi. On the NVIDIA GPUs, LCRQ is now below the traditional lock-free designs and our non-waiting interface by a factor of 8. LCRQ's performance degrades with each added thread on the K20c, reaching a low of 0.003 Mops/s with 623 threads, where the next lowest, the FC-queue, is 0.322 Mops/s, and the blocking interface performs 91.803 Mops/s, *four orders of magnitude* higher throughput than LCRQ. Applications of this nature, where consumers and producers are imbalanced, are pathologically bad for LCRQ. None of the other queues are materially affected by the imbalance.

7.6 Conclusions

In this chapter, we have presented the design of a high-throughput FIFO queue for many-core architectures. Our design includes both high-throughput waiting and low-latency non-waiting interfaces to customize interactions with the queue on a per-thread or per-interaction basis, both of which are linearizable to the semantics of a “channel queue.” While queues with hard progress guarantees and unbounded size have their benefits, we have shown that focusing on throughput and avoiding retry-based algorithms can produce exceptionally high throughput across a wide range of real-world multi- and many-core hardware. Counter-intuitively, designing an algorithm that allows blocking to occur but increases the maximum concurrency of the structure results in greater throughput. In fact, our evaluation finds that performance can be improved by as much as *1000-fold* for some problems in an environment with more than 1000 concurrent threads.

Chapter 8

Summary and Future Work

8.1 Summary

Heterogeneity, and the complexity that comes with it, is becoming increasingly common. Exploiting all resources in a system is also becoming increasingly critical as performance and energy efficiency have become important drivers of cost in HPC. This dissertation has explored approaches and mechanisms to automate the management and exploitation of heterogeneous resources in a system, improving programmability, performance portability and even overall performance in some cases. We first presented an approach to detecting the symptoms of heterogeneity in homogeneous MPI applications. By collecting hardware counters, as well as communication and computation time information, SyMMer can detect inappropriate mappings of MPI ranks to cores, and how to re-map to correct them. Overall, SyMMer proved effective at exploiting the dual imbalance of applications and hardware to consistently provide high application performance. The downside to this approach was also one of its greatest strengths, as an extension of MPI, it had no hooks to alter the application's behavior, only its placement. As Remote Direct Memory Access (RDMA) networks, Message Signaled Interrupts (Extended) (MSI-X) devices capable of targeting interrupts on a per-process level, and irqbalance have became more common, the inherent imbalance in the operating system has decreased, limiting the opportunities for SyMMer. Given all of these factors, we began to investigate alternative substrates for our autonomic systems.

Shifting from heterogeneity in physically homogeneous systems to physically heterogeneous ones presented new opportunities and challenges. Using Accelerated OpenMP as a target, we gained greater control over the behavior of the application. With that new control, we presented a new model for efficiently and accurately predicting an appropriate share of work for each device in heterogeneous systems. The adaptive static scheduling approach, and the linear model developed to feed it in Splitter and refined further in CoreTSAR, generates work distributions that consistently achieve within 5% of the static optimal for regular applications. We also show that this model of work distribution can both save overhead and increase locality when compared to more traditional dynamic or "chunk" style scheduling schemes. Our benchmarks show near-linear scaling to four GPUs, and good portability across a wide range of machine architectures as well. Even so, our experience with certain benchmarks in CoreTSAR led us to design an adaptively sized chunk scheduler as well, to handle applications with irregular work distribution across iterations.

In addition to the scheduling component CoreTSAR also presented a new memory association model, allowing worksharing to be finer grained without requiring user control over final placement of data. This work differentiates itself from previous efforts by reversing the usual paradigm. Rather than specifying a distribution of data across devices and then running computation on whichever device the data happens to reside on, we associate the data with the computation. Given that information, our model can assign the computation appropriately, and move or replicate the data as necessary to fit that distribution. This reversal allows for both straightforward specification, the user only must specify the association and an arbitrary number of devices may be targeted, and greater freedom for the worksharing scheduler. An initial downside to this approach however was the high data-movement cost, and the difficulty of maintaining transparency when replicating subsets of data in other address spaces.

Building on the memory association work done in CoreTSAR, we have also presented a more complete and precise memory association system as part of AffinityTSAR. Rather than treating memory association as only a problem, or opportunity, for heterogeneous devices and regions, we expanded the scope to cover the partitioning of work and data across threads on the CPU as well. The result is a more flexible and holistic model for memory affinity, thread affinity, and managing partitioning of work across parallel OpenMP regions. This approach also served to address the memory movement overheads that originally plagued the model by allowing the partitioning to occur in an outer scope, and only move data when required or requested rather than at every entry into a region. This model provided a greater than three-fold speedup for some GPU tests over even our previous model. The partitioning approach that we present also serves to re-balance affinity for CPU cores by re-mapping of memory and work on CPUs to provide consistently high performance, as SyMMer would accomplish with process re-mapping.

In support of our future goals to support finer-grained and pull-based scheduling across heterogeneous platforms, we also present our work toward creating a more scalable concurrent FIFO queue for many-core architectures and platforms. By trading off the progress guarantee, our queue is *not* lock-free, we create a queue that can scale to over a thousand contending threads while remaining reasonably user-friendly. In fact, our design exceeds the throughput of others on some GPU platforms by hundreds of times, even over a thousand fold in some cases.

As a whole, the approaches presented in this dissertation provide well explored solutions to achieving programmability in the face of heterogeneity and adaptation. Further, we have implemented each approach into a runtime system, and combined their concepts into a single over-arching system that exploits our techniques to provide performance improvements and a testbed for further research into autonomic systems for heterogeneous systems.

8.2 Future Work

While this dissertation covers our work toward creating a new autonomic system for heterogeneous computers, we cover a finite subset of the dimensions of heterogeneity, and significant future work presents itself along this path. Each of the sections below covers a major branch of future work, all of which could be combined with our existing and proposed work to create an ever more encompassing system for dealing with heterogeneity.

8.3 Alternative Performance Metrics

Throughout this work, we have assumed that the metric of performance is speed, or work completed per second. In many environments, other metrics are appropriate, and deserve equal attention. For example, one of the primary limiting factors in the growth of supercomputers today is energy, the amount of electricity actually consumed and dissipated by the machine. An investigation of energy as an alternative metric for performance could allow our existing designs to target it with little modification. By optimizing for energy, the system could assist in energy conservation efforts for battery powered or power constrained computations. In a supercomputing setting, a multi-dimensional optimization that targets the best of speed and energy efficiency could produce better results. Regardless of the platform, automated scheduling based on energy requirements is a worthwhile problem. Beyond energy and speed, power limiting and focus on latency or bandwidth specifically could each be treated separately as well.

8.4 Fine-Grained Scheduling and Synchronization in Multi-Accelerator Systems

Our work has focused on a certain class of programs, mainly those written in the OpenMP/BSP style, employing large parallel loops for their parallelism. While many scientific applications map well into this model, many others either would prefer or even require a more task-centric or finer-grained approach to perform well. Up to this point, scheduling at the granularity of a single core, or SMX to use the CUDA Kepler generation terminology, on a GPU has been somewhere between impractical and impossible. As new architectures are released, notably AMD's HSA and NVIDIA's planned support for unified memory in Maxwell generation GPUs, communication with individual compute units on the accelerators is becoming practical. We are interested in designing a scheduling system that can treat every individual sub-processor of every device in a system on an even footing. Allow for BSP style scheduling as we do now, but also allow for extremely fine-grained worksharing across devices and memory spaces. The first step toward this goal is to develop a set of techniques and data structures that make efficient communications in this particularly difficult environment possible, including work-queues, stacks, barriers and other concurrency constructs. The queue that we presented in Chapter 7 is the first step on this path, and we are in the process of developing a stack with similar properties.

8.5 Inter-Node Parallelism

In our current work, the focus is on heterogeneity within a single computer, but that is not due to clusters, grids and other aggregate computers presenting homogeneous systems. Quite the contrary in fact, just as nodes now have non-trivial topologies, aggregate systems take those to another level entirely. Developing a system to manage heterogeneity across nodes, rebalancing work at the granularity of nodes, then devices in them could produce significant benefits for such architectures. The major issue in this direction is the programming model to apply. Classically, MPI or other message passing interfaces dominate this space though systems based on partitioned global address spaces are also popular. Neither of these retains the clarity of serial code, and generally force users to flatten the patterns of their computation into discrete message passing or shared memory communication semantics. Rather we envision a system that would express large scale parallelism in terms of patterns of communication and computation, giving the runtime foreknowledge of the intended structure. In such an environment, load balancing, fault recovery, and greater re-usability of communication infrastructure, in the form of patterns, all become attainable.

Bibliography

[1] PAPI. http://icl.cs.utk.edu/papi/.

[2] TOP500 Supercomputer Sites. http://top500.org.

[3] CUDA Programming Guide. `http://docs.nvidia.com/cuda/cuda-c-programming-guide/`, 2007.

[4] Intel Threading Building Blocks. https://www.threadingbuildingblocks.org/, 2007.

[5] Thrust. http://code.google.com/p/thrust/, 2009.

[6] The OpenCL Specification. `https://www.khronos.org/registry/cl/specs/opencl-1.2.pdf`, Nov. 2012.

[7] OpenACC 2.0 Application Programming Interface Specification. `http://www.openacc.org/sites/default/files/OpenACC%202%200.pdf`, June 2013.

[8] R. Anandakrishnan, T. R. W. Scogland, A. T. Fenley, J. C. Gordon, W. Feng, and A. V. Onufriev. Accelerating Electrostatic Surface Potential Calculation with Multi-scale Approximation on Graphics Processing Units. *Journal of Molecular Graphics and Modelling*, 28(8):904–910, June 2010.

[9] J. H. Anderson, J. M. Calandrino, and U. C. Devi. Real-Time Scheduling on Multicore Platforms. *Proceedings of the 12th IEEE Real-Time and Embedded Technology and Applications Symposium*, pages 179–190, 2006.

[10] J. Archuleta, Y. Cao, T. R. W. Scogland, and W. Feng. Multi-dimensional Characterization of Temporal Data Mining on Graphics Processors. In *International Parallel and Distributed Processing Symposium*, pages 1–12, 2009.

[11] Argonne National Laboratory. MPICH2: High Performance and Portable Message Passing. http://www.mcs.anl.gov/research/projects/mpich2.

[12] C. Augonnet, S. Thibault, and R. Namyst. StarPU: a Runtime System for Scheduling Tasks over Accelerator-Based Multicore Machines. Technical Report RR-7240, Laboratoire Bordelais de Recherche en Informatique - LaBRI, RUNTIME - INRIA Bordeaux - Sud-Ouest, Mar. 2010.

[13] C. Augonnet, S. Thibault, R. Namyst, and P.-A. Wacrenier. StarPU: A Unified Platform for Task Scheduling on Heterogeneous Multicore Architectures. In *International Euro-Par Conference on Parallel Processing*. Springer-Verlag, Aug. 2009.

[14] E. Ayguadé, B. Blainey, A. Duran, J. Labarta, F. Martínez, X. Martorell, and R. Silvera. Is the Schedule Clause Really Necessary in OpenMP? In *Workshop on OpenMP Applications and Tools*. Springer-Verlag, June 2003.

[15] D. H. Bailey, E. Barszcz, J. Barton, D. Browning, R. Carter, L. Dagum, R. Fatoohi, P. Frederickson, T. Lasinski, R. Schreiber, H. Simon, V. Venkatakrishnan, and S. Weeratunga. The NAS Parallel Benchmarks Summary and Preliminary Results. In *International Conference for High Performance Computing, Networking, Storage and Analysis (SuperComputing)*, pages 158–165, 1991.

[16] A. Basumallik, S.-J. Min, and R. Eigenmann. Programming Distributed Memory Sytems Using OpenMP. In *International Parallel and Distributed Processing Symposium*, pages 1–8, 2007.

[17] H. J. C. Berendsen, D. van der Spoel, and R. van Drunen. GROMACS: A Message-Passing Parallel Molecular Dynamics Implementation. *Computer Physics Communications*, 91(1-3):43–56, Sept. 1995.

[18] M. Berkelaar, K. Eikland, P. Notebaert, and Others. lp_solve: Open-Source (Mixed-Integer) Linear Programming System. *Eindhoven U. of Technology*, 2004.

[19] J. C. Beyer, E. J. Stotzer, A. Hart, and B. R. de Supinski. OpenMP for Accelerators. In *Lecture Notes in Computer Science: OpenMP in the Petascale Era*, pages 108–121. Springer Berlin Heidelberg.

[20] J. Bircsak, P. Craig, R. Crowell, Z. Cvetanovic, J. Harris, C. A. Nelson, and C. D. Offner. Extending OpenMP For NUMA Machines. In *International Conference for High Performance Computing, Networking, Storage and Analysis (SuperComputing)*, page 48, 2000.

[21] G. E. Blelloch, P. Cheng, and P. B. Gibbons. Scalable Room Synchronizations. *Theory of Computing Systems*, 36(5):397–430, Aug. 2003.

[22] R. D. Blumofe, C. F. Joerg, B. C. Kuszmaul, C. E. Leiserson, K. H. Randall, and Y. Zhou. Cilk: An Efficient Multithreaded Runtime System. *Journal of Parallel and Distributed Computing*, 37(1):55–69, 1996.

[23] R. R. Bordawekar and U. Bondhugula. Can CPUs Match GPUs on Performance with Productivity?: Experiences with Optimizing a FLOP-Intensive Application on CPUs and GPU. *IBM Reseach Report RC25033*.

[24] F. Broquedis, J. Clet-Ortega, S. Moreaud, N. Furmento, B. Goglin, G. Mercier, S. Thibault, and R. Namyst. hwloc: A Generic Framework for Managing Hardware Affinities in HPC Applications. In *International Conference on Parallel, Distributed and Network-Based Processing (PDP)*, pages 180–186. IEEE Computer Society, 2010.

[25] F. Broquedis, N. Furmento, B. Goglin, R. Namyst, and P.-A. Wacrenier. Dynamic Task and Data Placement Over NUMA Architectures: An OpenMP Runtime Perspective. In *International Workshop on OpenMP: Evolving OpenMP in an Age of Extreme Parallelism*. Springer-Verlag, May 2009.

[26] F. Broquedis, N. Furmento, B. Goglin, P.-A. Wacrenier, and R. Namyst. ForestGOMP: An Efficient OpenMP Environment for NUMA Architectures. *International Journal of Parallel Programming*, 38(5-6):418–439, May 2010.

[27] J. Bueno, X. Martorell, R. M. Badia, E. Ayguadé, and J. Labarta. Implementing OmpSs Support for Regions of Data in Architectures with Multiple Address Spaces. In *ACM International Conference on Supercomputing*. ACM, June 2013.

[28] J. Bueno, J. Planas, A. Duran, R. M. Badia, X. Martorell, E. Ayguadé, and J. Labarta. Productive Programming of GPU Clusters with OmpSs. *International Parallel and Distributed Processing Symposium*, pages 557–568, 2012.

[29] D. Cederman, B. Chatterjee, and P. Tsigas. Understanding the Performance of Concurrent Data Structures on Graphics Processors. *Euro-Par 2012 Parallel Processing*, 2012.

[30] L. Chai, Q. Gao, and D. K. Panda. Understanding the Impact of Multi-Core Architecture in Cluster Computing: A Case Study with Intel Dual-Core System. In *IEEE/ACM International Symposium on Cluster Computing and the Grid*, 2007.

[31] B. L. Chamberlain, D. Callahan, and H. P. Zima. Parallel Programmability and the Chapel Language. *International Journal of High Performance Computing Applications*, 21(3):291–312, 2007.

[32] M. Daga, T. R. W. Scogland, and W. Feng. Architecture-Aware Mapping and Optimization on a 1600-Core GPU. In *International Conference on Parallel and Distributed Systems*, pages 316–323, Tainan, Taiwan, 2011. IEEE Computer Society.

[33] L. Dagum and R. Menon. OpenMP: An Industry-Standard API for Shared-Memory Programming. *IEEE Computational Science and Engineering*, 05(1):46–55, 1998.

[34] J. Dean. MapReduce: Simplified Data Processing on Large Clusters. *Communications of the ACM*, 2008.

[35] R. Dolbeau, S. Bihan, and F. Bodin. HMPP: A Hybrid Multi-Core Parallel Programming Environment. In *GPGPU 2007: Workshop on General Purpose Processing on Graphics Processing Units*, 2007.

[36] P. Du, R. Weber, P. Luszczek, S. Tomov, G. Peterson, and J. Dongarra. From CUDA to OpenCL: Towards a Performance-Portable Solution for Multi-Platform GPU Programming. *Parallel Computing*, 38(8), Aug. 2012.

[37] A. Duran, E. Ayguadé, R. M. Badia, J. Labarta, L. Martinell, X. Martorell, and J. Planas. OmpSs: A Proposal for Programming Heterogeneous Multi-Core Architectures. *Parallel Processing Letters*, 21(2):173–193, 2011.

[38] M. Elteir, H. Lin, W. Feng, and T. R. W. Scogland. StreamMR: An Optimized MapReduce Framework for AMD GPUs. In *International Conference on Parallel and Distributed Systems*, pages 364–371, 2011.

[39] A. Fedorova, M. Seltzer, and M. D. Smith. Cache-Fair Thread Scheduling for Multicore Processors. Technical Report TR-17-06, Harvard University, Oct. 2006.

[40] W. Feng and T. R. W. Scogland. The Green500 List: Year One. In *Workshop on High-Performance, Power-Aware Computing (IPDPSW: HPPAC)*, Rome, Italy, May 2009. IEEE Computer Society.

[41] A. T. Fenley, J. C. Gordon, and A. Onufriev. An Analytical Approach to Computing Biomolecular Electrostatic Potential. I. Derivation and Analysis. *The Journal of Chemical Physics*, 129(7):075101, 2008.

[42] M. Frigo and S. G. Johnson. The Design and Implementation of FFTW3. *Proceedings of the IEEE*, 93(2):216–231, 2005.

[43] A. Gidenstam, H. Sundell, and P. Tsigas. Efficient Lock-Free Queues that Mind the Cache. In *3rd Swedish Workshop on Multi-core Computing (MCC 2010)*, July 2010.

[44] J. Gordon and A. Fenley. An Analytical Approach To Computing Biomolecular Electrostatic Potential. II. Validation and Applications. *The Journal of Chemical Physics*, 2008.

[45] A. Gottlieb, B. D. Lubachevsky, and L. Rudolph. Basic Techniques for the Efficient Coordination of Very Large Numbers of Cooperating Sequential Processors. *Transactions on Programming Languages and Systems (TOPLAS*, 5(2), Apr. 1983.

[46] S. Grauer-Gray, L. Xu, R. Searles, S. Ayalasomayajula, and J. Cavazos. Auto-Tuning a High-Level Language Targeted to GPU Codes. *Innovative Parallel Computing*, pages 1–10, 2012.

[47] A. Haas, M. Lippautz, T. A. Henzinger, H. Payer, A. Sokolova, C. M. Kirsch, and A. Sezgin. Distributed Queues in Shared Memory: Multicore Performance and Scalability Through Quantitative Relaxation. In *ACM International Conference on Computing Frontiers*, New York, New York, USA, 2013. ACM Press.

[48] B. He, W. Fang, Q. Luo, N. K. Govindaraju, and T. Wang. Mars: A MapReduce Framework on Graphics Processors. In *International Conference on Parallel Architectures and Compilation Techniques*, 2008.

[49] D. Hendler, I. Incze, N. Shavit, and M. Tzafrir. Flat Combining and the Synchronization-Parallelism Tradeoff. In *ACM Symposium on Parallelism in Algorithms and Architectures*. ACM, June 2010.

[50] M. P. Herlihy and J. M. Wing. Linearizability: A Correctness Condition for Concurrent Objects. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 12(3):463–492, 1990.

[51] C. Hong, D. Chen, W. Chen, W. Zheng, and H. Lin. MapCG: Writing Parallel Program Portable Between CPU and GPU. In *International Conference on Parallel Architectures and Compilation Techniques*. ACM, Sept. 2010.

[52] V. J. Jiménez, L. Vilanova, I. Gelado, M. Gil, G. Fursin, and N. Navarro. Predictive Runtime Code Scheduling for Heterogeneous Architectures. In *International Conference on High Performance Embedded Architectures and Compilers*. Springer-Verlag, Dec. 2008.

[53] L. Kale and S. Krishnan. CHARM++: A Portable Concurrent Object Oriented System Based on C++. In A. Paepcke, editor, *ACM SIGPLAN Conference on Object Oriented Programming, Systems, Languages, and Applications*, pages 91–108. ACM Press, Sept. 1993.

[54] C. M. Kirsch, M. Lippautz, and H. Payer. Fast and Scalable, Lock-Free k-FIFO Queues. In *International Conference on Parallel Architectures and Compilation Techniques*, 2012.

[55] A. Kogan and E. Petrank. Wait-free queues with multiple enqueuers and dequeuers. In *Symposium on Principles and Practice of Parallel Programming*, pages 223–234. ACM, 2011.

[56] A. Kogan and E. Petrank. A Methodology for Creating Fast Wait-Free Data Structures. In *Symposium on Principles and Practice of Parallel Programming*. ACM, Feb. 2012.

[57] J. Lee and M. Sato. Implementation and Performance Evaluation of XcalableMP: A Parallel Programming Language for Distributed Memory Systems. In *International Conference on Parallel Processing, Workshops (ICPPW)*, pages 413–420, 2010.

[58] S. Lee and S. J. Min. OpenMP to GPGPU: a Compiler Framework for Automatic Translation and Optimization. In *Symposium on Principles and Practice of Parallel Programming*, pages 101–110. ACM New York, NY, USA, ACM, 2009.

[59] S. Lee and J. S. Vetter. OpenARC: Open Accelerator Research Compiler for Directive-Based, Efficient Heterogeneous Computing. In *HPDC '14: Proceedings of the 23rd International Symposium on High-Performance Parallel and Distributed Computing*. ACM, June 2014.

[60] V. W. Lee, C. Kim, J. Chhugani, M. Deisher, D. Kim, A. D. Nguyen, N. Satish, M. Smelyanskiy, S. Chennupaty, P. Hammarlund, R. Singhal, and P. Dubey. Debunking the 100X GPU vs. CPU Myth: An Evaluation of Throughput Computing on CPU and GPU. In *International Symposium on Computer Architecture (ISCA '10)*. ACM, June 2010.

[61] C. E. Leiserson. The Cilk++ Concurrency Platform. *The Journal of Supercomputing*, 51(3):244–257, 2010.

[62] T. Li, D. Baumberger, D. A. Koufaty, and S. Hahn. Efficient Operating System Scheduling for Performance-Asymmetric Multi-Core Architectures. In *International Conference for High Performance Computing, Networking, Storage and Analysis (SuperComputing)*, pages 1–11, 2007.

[63] H. Löf and S. Holmgren. Affinity-on-next-touch: Increasing the Performance of an Industrial PDE Solver on a cc-NUMA System. In *ACM International Conference on Supercomputing*, pages 387–392. ACM, 2005.

[64] D. B. Loveman. High Performance Fortran. *Parallel & Distributed Technology: Systems & Applications, IEEE*, 1(1):25–42, 1993.

[65] C.-K. Luk, S. Hong, and H. Kim. Qilin: Exploiting Parallelism on Heterogeneous Multiprocessors with Adaptive Mapping. In *MICRO 42: Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture*. ACM, Dec. 2009.

[66] A. Mariano, R. Alves, J. Barbosa, L. P. Santos, and A. Proenca. A (Ir) Regularity-Aware Task Scheduler for Heterogeneous Platforms. In *Conference on High Performance Computing (HPC-UA)*, 2012.

[67] J. D. McCalpin. Memory Bandwidth and Machine Balance in Current High Performance Computers. *IEEE Computer Society Technical Committee on Computer Architecture (TCCA) Newsletter*, pages 19–25, 1995.

[68] Message Passing Interface Forum. MPI: A Message-Passing Interface Standard. http://www.mpi-forum.org/docs/mpi-3.0/mpi30-report.pdf, Sept. 2012.

[69] M. M. Michael and M. L. Scott. Simple, Fast, and Practical Non-Blocking and Blocking Concurrent Queue Algorithms. In *ACM Symposium on Principles of Distributed Computing*. ACM, May 1996.

[70] C. Min and Y. I. Eom. Integrating Lock-free and Combining Techniques for a Practical and Scalable FIFO Queue. *IEEE Transactions on Parallel and Distributed Systems*, PP(99):1, jun 2014.

[71] A. Morrison and Y. Afek. Fast Concurrent Queues for x86 Processors. In *Symposium on Principles and Practice of Parallel Programming*. ACM, Feb. 2013.

[72] G. Narayanaswamy, P. Balaji, and W. Feng. An Analysis of 10-Gigabit Ethernet Protocol Stacks in Multicore Environments. In *15th International Symposium on High-Performance Interconnects (HotI 2007)*, pages 109–116, Palo Alto, California, Aug. 2007. IEEE Computer Society Washington, DC, USA.

[73] D. Nikolopoulos, T. Papatheodorou, C. Polychronopoulos, J. Labarta, and E. Ayguadé. Is Data Distribution Necessary in OpenMP? In *International Conference for High Performance Computing, Networking, Storage and Analysis (SuperComputing)*, page 47, 2000.

[74] D. S. Nikolopoulos, T. S. Papatheodorou, C. D. Polychronopoulos, J. Labarta, and E. Ayguadé. A Case for User-Level Dynamic Page Migration. In *ACM International Conference on Supercomputing*. ACM, May 2000.

[75] T. Nomizu, D. Takahashi, J. Lee, T. Boku, and M. Sato. Implementation of XcalableMP Device Acceleration Extention with OpenCL. *International Parallel and Distributed Processing Symposium, Workshops and Phd Forum*, pages 2394–2403, 2012.

[76] R. W. Numrich and J. Reid. Co-Array Fortran for Parallel Programming. *SIGPLAN Fortran Forum*, 17(2):1–31, 1998.

[77] T. Odajima, T. Boku, T. Hanawa, J. Lee, and M. Sato. GPU/CPU Work Sharing with Parallel Language XcalableMP-dev for Parallelized Accelerated Computing. In *2012 41st International Conference on Parallel Processing Workshops (ICPPW)*, pages 97–106. IEEE.

[78] OpenMP ARB. OpenMP 4.0 Specification. `http://www.openmp.org/mp-documents/OpenMP4.0.0.pdf`, June 2013.

[79] D. Orozco, E. Garcia, R. Khan, K. Livingston, and G. R. Gao. Toward High-Throughput Algorithms on Many-Core Architectures. *ACM Transactions on Architecture and Code Optimization*, 8(4):1–21, Jan. 2012.

[80] S. Plimpton. Fast Parallel Algorithms for Short-Range Molecular Dynamics. *J. Comput. Phys.*, 117(1):1–19, 1995.

[81] M. M. Rafique, B. Rose, A. R. Butt, and D. S. Nikolopoulos. CellMR: A Framework for Supporting MapReduce on Asymmetric Cell-Based Clusters. In *International Parallel and Distributed Processing Symposium*, pages 1–12. IEEE Computer Society, May 2009.

[82] V. T. Ravi and G. Agrawal. A Dynamic Scheduling Framework for Emerging Heterogeneous Systems. In *International Conference on High Performance Computing (HiPC)*, pages 1–10, 2011.

[83] B. Rountree, D. K. Lownenthal, B. R. de Supinski, M. Schulz, V. W. Freeh, and T. Bletsch. Adagio: Making DVS Practical for Complex HPC Applications. In *ACM International Conference on Supercomputing*. ACM, June 2009.

[84] M. Sato, H. Harada, and Y. Ishikawa. OpenMP Compiler for a Software Distributed Shared Memory System SCASH. *International Workshop on OpenMP Applications and Tools (WOMPAT)*, 2000.

[85] T. R. W. Scogland, P. Balaji, W. Feng, and G. Narayanaswamy. Asymmetric Interactions in Symmetric Multi-Core Systems: Analysis, Enhancements and Evaluation. In *International Conference for High Performance Computing, Networking, Storage and Analysis (Super-Computing)*, 2008.

[86] T. R. W. Scogland, B. Rountree, W. Feng, and B. R. de Supinski. Heterogeneous Task Scheduling for Accelerated OpenMP. In *International Parallel and Distributed Processing Symposium*, pages 144–155. IEEE Computer Society, May 2012.

[87] T. R. W. Scogland, B. Rountree, W. Feng, and B. R. de Supinski. CoreTSAR: Adaptive Work-sharing for Heterogeneous Systems. In *International Supercomputing Conference*, Leipzig, June 2014.

[88] T. R. W. Scogland, B. Subramaniam, and W. Feng. The Green500 List: Escapades to Exascale. *Computer Science-Research and Development*, pages 1–9, 2012.

[89] S. Seo, G. Jo, and J. Lee. Performance Characterization of the NAS Parallel Benchmarks in OpenCL. In *IEEE International Symposium on Workload Characterization (IISWC)*, pages 137–148. IEEE, 2011.

[90] A. Sidelnik, S. Maleki, B. L. Chamberlain, M. J. Garzarán, and D. Padua. Performance Portability with the Chapel Language. In *International Parallel and Distributed Processing Symposium*. IEEE Computer Society, May 2012.

[91] K. Spafford, J. Meredith, and J. Vetter. Maestro: Data Orchestration and Tuning for OpenCL Devices. In *International Euro-Par Conference on Parallel Processing*. Springer-Verlag, Aug. 2010.

[92] P. Tsigas and Y. Zhang. A Simple, Fast and Scalable Non-Blocking Concurrent FIFO Queue for Shared Memory Multiprocessor Systems. In *ACM Symposium on Parallelism in Algorithms and Architectures*. ACM, July 2001.

[93] L. Valiant. A Bridging Model for Parallel Computation. *Communications of the ACM*, 1990.

[94] R. Virding, C. Wikström, and M. Williams. *Concurrent Programming in ERLANG (2Nd Ed.)*. Prentice Hall International (UK) Ltd., Hertfordshire, UK, 1996.

[95] M. Wolfe. Implementing the PGI Accelerator Model. In *Annual Workshop on General Purpose Processing with Graphics Processing Units*. ACM, Mar. 2010.