REALM: PERFORMANCE PORTABILITY THROUGH COMPOSABLE ASYNCHRONY

A DISSERTATION
SUBMITTED TO THE DEPARTMENT OF COMPUTER SCIENCE
AND THE COMMITTEE ON GRADUATE STUDIES
OF STANFORD UNIVERSITY
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY

Sean Jeffrey Treichler

December 2016

I certify that I have read this dissertation and that, in my opinion, it is fully adequate in scope and quality as a dissertation for the degree of Doctor of Philosophy.

Alex Aiken, Primary Adviser

I certify that I have read this dissertation and that, in my opinion, it is fully adequate in scope and quality as a dissertation for the degree of Doctor of Philosophy.

Pat Hanrahan

I certify that I have read this dissertation and that, in my opinion, it is fully adequate in scope and quality as a dissertation for the degree of Doctor of Philosophy.

Philip Levis

Approved for the Stanford University Committee on Graduate Studies.

Patricia J. Gumport, Vice Provost for Graduate Education

Abstract

Modern supercomputers are growing increasingly complicated. The laws of physics have forced processor counts into the thousands or even millions, resulted in the creation of deep distributed memory hierarchies, and encouraged the use of multiple processor and memory types in the same system. Developing an application that is able to fully utilize such a system is very difficult. The development of an application that is able to run well on more than one such system with current programming models is so daunting that it is generally not even attempted.

The Legion project attempts to address these challenges by combining a traditional hierarchical application structure (i.e. tasks/functions calling other tasks/functions) with a hierarchical data model (logical regions, which may be partitioned into subregions), and introducing the concept of mapping, a process in which the tasks and regions of a machine-agnostic description are assigned to the processors and memories of a particular machine.

This dissertation focuses on Realm, the "low-level" runtime that manages the execution of a mapped Legion application. Realm is a fully asynchronous event-based runtime. Realm operations are deferred by the runtime, returning an event that triggers upon completion of the operation. These events may be used as preconditions for other operations, allowing arbitrary composition of asynchronous operations. The resulting operation graph naturally exposes the available parallelism in the application as well as opportunities for hiding the latency of any required communication. While asynchronous task launches and non-blocking data movement are fairly common in existing programming models, Realm makes all runtime operations asynchronous — this includes resource management, performance feedback, and even, apparently paradoxically, synchronization primitives.

Important design and implementation issues of Realm will be discussed, including the novel generational event data structure that allows Realm to efficiently and scalably handle a very large number of events in a distributed environment and the machine model that provides the information required for the mapping of a Legion application onto a system. Realm anticipates dynamic behavior of both future applications and future systems and includes mechanisms for application-directed profiling, fault reporting, and dynamic code generation that further improve performance portability by allowing an application to adapt to and optimize for the exact system configuration used for each run.

Microbenchmarks demonstrate the efficiency and scalability of the Realm and justify some of the non-obvious design decisions (e.g. unfairness in locks). Experiments with several mini-apps are used to measure the benefit of a fully asynchronous runtime compared to existing "non-blocking" approaches. Finally, performance of Legion applications at full-scale show how Realm's composable asynchrony and support for heterogeneity benefit the overall Legion system on a variety of modern supercomputers.

Acknowledgments

Life is full of team efforts. Nobody crosses the finish line without having a support team to back them up: people that help you train, people that make sure you have the right equipment, people that clear the road ahead of you, and people that give you that emotional support you need when you're too tired to take another step.

As my advisor, Alex Aiken served in all these roles. I arrived at Stanford with a solid idea of what I wanted to build, but little plan for how to do it. Alex helped me turn that long-term vision into the smaller steps needed to make actual progress. He helped me deal with pitfalls along the way, but more importantly, used his experience and intuition to anticipate and avoid many more pitfalls. I'd like to think that I taught Alex a few things along the way, but the balance of knowledge transfer is pretty clearly in the other direction.

Phil Levis and Pat Hanrahan deserve mention too, and not just for slogging through the early drafts of this dissertation. Pat constantly encouraged me to think big and look for ways that my work could be used to solve larger problems. Phil helped me refine my arguments and distill the key ideas on which my work is based.

The faculty and staff in the Computer Science Department at Stanford have constructed an amazing environment for research. I will feel forever guilty about the amount of paperwork that was done (and done cheerfully!) on my behalf, letting me focus on my work. Similarly, the willingness of the folks at NVIDIA to let me run "solo" while staying connected has made it much more likely that my research will be of practical use in future systems.

The Legion project has become a sprawling thing, but it would not exist without fellow founding member Michael Bauer. Working with him to first flesh out a vision for performance portable computing and then prove that it could be done has been a rewarding experience on many levels. Current and former members of Alex's research group have contributed materially to the Legion project (Elliott Slaughter, Wonchan Lee, Zhihao Jia, Todd Warszawski, Rahul Sharma) and/or provided general perspective and the occasional distraction that kept me from losing track of my surroundings (Eric Schkufza, Peter Hawkins, Adam Oliner, Stefan Heule, Manolis Papadakis).

Our work has given us the opportunity to collaborate with many scientists at the Department of Energy's laboratories. All have provided the healthy skepticism that drives research in the first place, but also enthusiasm and support for exploring new approaches to big problems. Outstanding in their enthusiasm and support have been Pat McCormick and Galen Shipman at Los Alamos National Laboratory and Jackie Chen at Sandia National Laboratories.

My family and friends have been with me for every step of this long road. Shoulders to cry on, ears to talk off, smiles of support, and even the occasional nagging all rekindle the fire when the pace begins to drop. In particular, Julie, Naomi, and Lisa remind me every day that the effects of caffeine pale in comparison to those of unconditional love.

At the end of the race, it is my name that will be recorded in the book, but this is not my accomplishment alone. It is a team effort, and I have the best team that I could ask for.

Contents

List of Figures

Chapter 1

Introduction

The world of high performance computing is in the midst of a programmability crisis. It takes multiple person-years of effort to port such an application to a new supercomputer[18], and that current expectations are that similar efforts will be needed for each new supercomputer architecture that comes online[17]. Today's supercomputers have simply become too hard to program, and while existing tools may allow a programmer, often with significant effort, to achieve good performance on a single target system, *performance portability* (the ability to have the same source code not just run, but run well, on multiple systems) is becoming out of reach. The systems that are planned to come online in the next few years promise an even greater challenge. In this thesis, we present Realm, a runtime system for high performance computing that investigates a basis for a lasting solution to this crisis.

The key idea that drives Realm is that of *composable asynchrony*, the idea that *all* operations performed by the runtime on behalf of the application can be deferred until their inputs are available, and that this deferral is handled by the runtime. Runtime deferral of operations allows the application code to be written in a way that is agnostic to the performance characteristics of the system on which the application is currently running. This agnosticism avoids the primary cause of portability problems: the need to change the original application code to improve performance on a specific machine and the interference between the changes needed for different machines.

We will discuss composable asynchrony and the system of *explicit dependencies* on which it is based at length, but we warn the reader in advance that this will not be sufficient. A comprehensive solution to the problem of performance portability should include the following as well:

- abstractions for the processing and storage resources in the system that allow application code to avoid using architecture- or vendor-specific interfaces for initiating computation or moving data within the system
- the ability to manage, and ideally generate, variants of performance-critical sections of code

for different processor architectures

- a model of the system at runtime, allowing an application to discover what resources are available and how they are connected
- an online profiling framework that provides information about the applications dynamic behavior back to the application itself
- a modular internal design that simplifies the addition of support for new system architecture features
- external interfaces with a clean separation of concerns, anticipating that in many cases, the "application" is actually higher-level runtime(s), library code, and/or even domain-specific language(s)

We believe Realm is currently unique in satisfying all of these requirements, and we offer Realm both as an artifact for use today but also as an argument that the combination of these features in a runtime system is both desirable and feasible.

This thesis is divided into three parts. The first part provides some background. The rest of this chapter describes what we mean when we say "high performance computing" and "supercomputers", and give a quick introduction to *bulk-synchronous programming*, the currently predominant programming style for high performance computing. The following two chapters will discuss the two root causes of the current programmability crisis, the growing latency gap and increases in system heterogeneity. That discussion includes a survey of existing tools and techniques that attempt to address these issues.

The second part forms the bulk of the thesis and describes composable asynchrony, its embodiment in the various components and concepts of Realm, how they work at a high level, and the rationale for the design. Chapter 6 dives a little more deeply into the implementation and benefits of *generational events*, a key reason why Realm can provide its composable asynchrony with low enough overhead to satisfy the needs of high performance computing. The Realm source code consists of over 60,000 lines of code, and the reader will be relieved that almost none of it appears in this thesis. Most of the code snippets are demonstrations of how an application might use Realm's features to attain performance portability.

The final part of the thesis is Chapter 11, an extended case study of the porting, tuning, and adaptation of S3D, a "real" high-performance computing application, to the Legion programming model. Legion uses Realm as its foundation, and this chapter contains the bulk of the experimental results of this thesis, both quantitative and qualitative. However, readers that wish to skip ahead to the results should stop in Chapters 4, 5, and 6, all of which contain results from micro-benchmarks or mini-apps, measuring the benefits and/or overheads of composable asynchrony.

1.1 High Performance Computing

Although it is a term in everyday use, the exact definition of *high performance computing* (HPC) is difficult to pin down. A common "definition" in use is to say that any application that requires a supercomputer is an HPC application. This suffers from, among other things, the fact that today's personal computers and even smartphones exceed the computational capabilities of the supercomputers of 10-15 years ago. Another approach is to look at various applications that are generally agreed to be "HPC applications" and identify commonalities. For example, the six applications that were used for application readiness testing of the Titan supercomputer at Oak Ridge National Laboratory were:[16]

- S3D, a simulation of the combustion of gases in a turbulent environment (e.g. the interior of an automobile engine)
- WL-LSMS, an application measuring the role of material disorder in nanoscale materials
- NRDF, a simulation of radiation transport, important in the study of astrophysics, laser fusion, medical imaging
- LAMMPS, a molecular model of cell membrane fusion used to study how molecular enter or exit living cells
- CAM-SE, an application that examines climate change adaptation and mitigation scenarios
- Denovo, an alternative approach to modeling radiation transport, especially in the field of nuclear energy

In general, these applications are constructing models of a physical system and then simulating how the system changes under the influence of known physical laws. The absolute scale of the system being modeled can vary from the subatomic to the galactic, and the time over which the system is evolved may vary from nanoseconds to trillions of years.

Other applications might work backwards instead, figuring out what physical laws would explain an observed change in a system. Examples include cosmology simulations that test dark matter hypotheses or reverse time migration, which identifies subterranean features based on seismic wave measurements. Finally, the system being modeled may not always be tangible. It might instead model social interactions for health or commercial reasons, or the complicated connections in today's financial markets.

Whether real or hypothesized, these systems are complicated enough that an exact solution is impossible to obtain. Instead, an approximate solution is computed by discretizing the system into millions or even billions of elements (e.g. atoms, photons, points in the simulation volume) and the timeline into thousands or millions of intervals, or *time steps*. Various properties are tracked for each element (e.g. energy, velocity, temperature) — there may be only a handful or hundreds.

The properties are computed for each interval, based on other properties, other elements, and/or other interval. This computation can be very simple (e.g. a single instruction to yield velocity from the difference of positions) or very complicated (e.g. thousands of instructions to determine the net reaction rate of one chemical species based on the local concentration of all the other species). One of the key features of HPC applications is that these same computation(s) get performed for many elements and for many intervals.

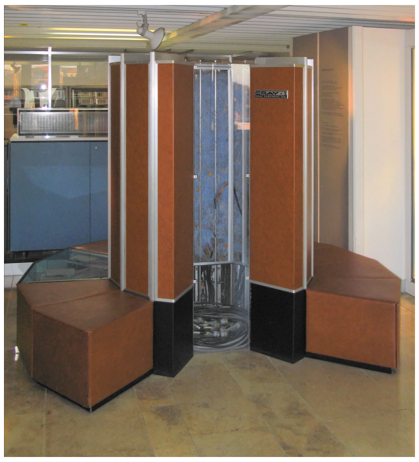
1.1.1 Comparisons to Big Data

Another commonly used but poorly defined class of applications of great interest today is *big data*. Big data applications are similar to HPC applications in that they operate on large data sets and demand “high performance”, but the performance metrics of interest differ greatly. As a result, the needs of the programmer differ in important ways.

A typical HPC application will run for hours on the same data set, and the most common metric used to compare implementations is the sustained rate at which the computation is being performed, either measured in *FLOPS* (*floating-point operations per second*) or as a fraction of the theoretical throughput of a given computer system. With the knowledge that the same computations are going to be performed on the same data set many times, an HPC application writer is willing to spend some effort up front to determine the best way to structure the computation and to distribute the data for the particular system on which the application will be run. The cost of this effort is more than paid back over the thousands or millions of iterations performed in the run, but anything the programming model or runtime can do to reduce either the programming time required or the necessary analysis time at application startup is greatly appreciated. Further, it is important that the model give the programmer as much control as possible over how the computation is performed. The optimal approach is likely to be different for every application, and a more generic algorithm that gets 90% of the optimal for a given workload is viewed as “wasting” that last 10%.

In contrast, the target runtime of a common big data application is minutes or seconds (or less!), and it is the response time that is of greatest concern (e.g. for a data analyst getting a response to an interactive query or for a web server to decide what to include in a requested web page). Efficient use of computing hardware resources is still important for big data applications, but the optimization opportunities are greatly reduced compared to the HPC case. The same big data application may be run many times, but the data sets and/or the precise analysis being performed are constantly changing. The return on investment for determining an optimal implementation for a given data set and query is often minimal. As a result, the programmer of a big data application will generally prefer a programming model that allows for rapid iteration and will happily yield control over how the computation is performed to those generic algorithms that provide “good enough” performance on whatever system is available to run the computation.

While Realm is primarily focused on the needs of HPC applications, efforts have been made to


(a) Cray-1 (Deutsches Museum)[1]

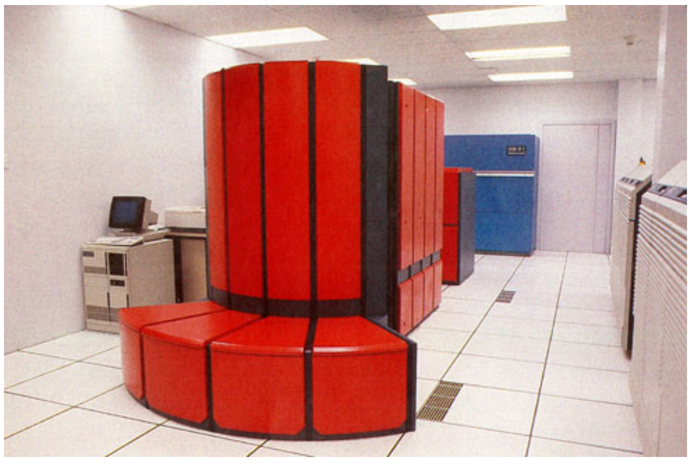

(b) Cray Y/MP (CSIRO)[2]

Figure 1.1: Early Cray Supercomputers

anticipate some of the needs of the big data space as well. Virtually everyone agrees that the line between HPC and big data has already started to blur. HPC applications are beginning to include more dynamic behavior and incorporate in-situ analysis capabilities. Big data applications are becoming more computationally intensive as well, and the ability to get the most out of a computer system is becoming more important to keep costs down.

1.2 Supercomputers

In aggregate, a single run of an contemporary HPC application can require the execution of $10^{10} - 10^{21}$ instructions on a working set that ranges from gigabytes in size up to petabytes. Even if you could somehow fit enough memory into a personal computer to run the application, the larger runs would take days or years to complete. Instead, computer systems are specifically designed for these workloads and are called *supercomputers* (completing our earlier circular definition).

Early supercomputers were custom-designed for the purpose, and looked nothing like a system you might have at your desk. Figure 1.1 shows the distinctive look of some of the more famous Cray systems of the 1970's and '80's. For a number of reasons (cost is perhaps the largest), today's supercomputers are constructed very differently. Instead of being a single monolithic system, a supercomputer today is a *cluster* of hundreds or thousands of smaller computers (often called *nodes*), each of which is similar to what you might find in a high-end personal computer. These individual systems are bolted into racks and the racks are lined up next to each other, generally filling a large room in a building specifically built for these systems. The Titan system takes up 4,352 square feet

[1] `https://en.wikipedia.org/wiki/Cray-1#/media/File:Cray-1-deutsches-museum.jpg`
[2] `http://www.csiro.au/news/newsletters/SIROscope/2010/March10/htm/supercomputing.htm`

of floor space, not counting the space on the floor below for delivering the 9 MW of electricity and on the roof above for dissipating the 9 MW of waste heat.

With the move from custom to commodity computer architecture, the “secret sauce” in a modern supercomputer is now in the interconnection network that allows the individual computers to communicate much more efficiently than the commodity networking (i.e. Ethernet) used in personal computers or data centers. A supercomputer’s interconnect makes use of custom hardware (e.g. Infiniband) with much higher peak bandwidth as well as custom software interfaces that allow applications to get data in and out of the network with much less overhead. The programmer of an HPC application must be very careful about when and how much data is moved from one node to another within the cluster if the application is to run well on one of today’s supercomputers.

1.3 Bulk-Synchronous Computing

The predominant programming model in use for high performance computing applications today is the *bulk-synchronous* style of computation. In bulk-synchronous computation, a program is conceptually split into a repeating cycle of three phases. (In many implementations, two of the phases may be entangled, but never all three.) Computation phases consist of a hopefully-large number of operations that can be performed without needing the result of any other operations being performed in the same phase. They are followed by communication phases, in which the results of these computations are made available for computations that occur in later phases. Each communication phase is then followed by a synchronization phase, which ensures that all the communication has completed before the next computation phase actually begins. The bulk-synchronous style is a good fit for most HPC applications, as an operation being performed on a large number of elements results in very “wide” computation phases. This ability to place a lot of work within a single computation phase is known as *data-parallelism*.

Some people dislike the use of the term *bulk-synchronous* to describe this approach, as the actual execution is rarely completely synchronous. In nearly all bulk-synchronous implementations, a node is allowed to advance to the next computation phase as soon as it has synchronized with at least the peers with which it interacted in the previous communication phase, allowing minor variations in execution speed to be absorbed. However, assuming the communication graph is connected, the progress of all nodes is ultimately still limited by the slowest node. Our focus in this discussion is on the programming model, and these optimizations do not change the programmer’s view, which remains one of alternating phases of computation, communication, and synchronization.

There are two major ways in which bulk-synchronous programming is done. We will start with the more intuitive one which is based on the idea of a *global view of control*. However, we will quickly shift to the *single-program multiple-data* (SPMD) approach, for two reasons. In addition to being (by far) the more common of the two in practice, virtually all compilers for languages with a global

```
#include <upc.h>

#define GRID_POINTS ...
#define TIME_STEPS ...

shared float [*] T[GRID_POINTS], d2Tdx2[GRID_POINTS];
float alpha = ...;  /* diffusion coefficient */
float dx = ...;  /* size of steps between grid points */
float dt = ...;  /* size of timesteps */

/* set initial conditions for T[] */

for(int t = 0; t < TIME_STEPS; t++) {
  upc_forall(int i = 1; i < GRID_POINTS-1; i++; &T[i]) {
    float left = T[i - 1];
    float right = T[i + 1];
    d2Tdx2[i] = (left + right - 2*T[i]) / (dx * dx);
  }

  upc_forall(int i = 1; i < GRID_POINTS-1; i++; &T[i])
    T[i] += alpha * dt * d2Tdx2[i];
}
```

Figure 1.2: Simulation of heat diffusion in UPC

view of control choose to generate SPMD style code for actual execution.

1.3.1 Global View of Control

Languages that provide a global view of control allow the programmer to write a bulk-synchronous program as if it were executing sequentially, but provide language constructs and/or compiler analysis that is able to mark the boundaries of computation phases, determine what communication must occur and in which phases, and insert the necessary synchronization. Examples of these languages include X10[23], UPC[19], and Chapel[22], and while they differ significantly in their syntax and what other features they provide, they are very similar with respect to the bulk-synchronous programming style.

Figure 1.2 shows a small bulk-synchronous program written in UPC. This is one of the canonical examples of an HPC application, and simulates the diffusion of heat over time along a one-dimensional metal bar. Simple UPC programs look nearly the same as C, so we will describe the overall application structure before returning to the specific features of UPC that allow it to parallelize the program across a large supercomputer.

Using the terminology above, the elements in this application are equally spaced grid points along the bar, and there are two properties for each element: the current temperature (`T`) and an estimate of the second derivative of the temperature along the length of the bar (`d2Tdx2`). In UPC, an array is used to store the value of a property for each element, so two arrays are declared on line 6. As is common for HPC applications, the outer loop on line 12 steps through discrete intervals of time. In each interval, both properties are updated for each element. First, the spatial derivative at each location along the bar is estimated based on finite differences (line 16) between the temperature at that location and the temperature at the grid locations to the left and right of the location (lines 14 and 15). This pattern in which is property is updated by accessing properties relative to a given element is often called a *stencil* operation. The spatial derivative is related to the temporal derivative in the heat equation by the diffusion coefficient `alpha`, and line 20 performs explicit integration to update the temperature property based on one timestep's worth of diffusion.

Returning to line 6, there are two pieces of syntax in the array declarations that are part of UPC's extensions to the underlying C language. The `shared` keyword at the beginning declares that these arrays will potentially be shared between different processors in the system (i.e. these are the values that may move during a communication phase). In UPC, each element of a shared array is assigned a "home" location, and the `[*]` syntax requests that the arrays be distributed coarsely. The array is divided into chunks of roughly even size, with one chunk being assigned to each processor.

The second UPC language feature used in this program is the `upc_forall` loop. It operates like a normal `for` loop, but is also used to mark the bounds of a computation phase. The optional fourth parameter in the loop header is used to determine which iterations of the loop should be performed by which processor. The expression should evaluate to the address of a `shared` variable or array element, and the computation is performed by the home processor. The UPC compiler automatically inserts synchronization phases between the computation phases, and uses compiler analysis to determine which subsets of the shared data need to be sent as part of the communication phase.

1.3.2 Single Program Multiple Data

With knowledge of which computations are to be performed on each processor, and what must be communicated, the UPC compiler (along with most other compilers for these languages) effectively generates a new program that will be executed by each processor, in which only its own computation, communication, and synchronization appears. This model in which each process has only a local view and performs explicit communication with other processes is exactly the SPMD approach. Luckily, we don't have to decipher the output of the UPC compiler to get an example — we can instead look a version of the same program written in MPI[58], the most common way that HPC applications are written today.

MPI is a runtime rather than a language, and provides bindings for that allow its use from a host language, such as C or Fortran. There are, however, languages that incorporate SPMD constructs

```
#include <mpi.h>

#define POINTS_PER_RANK ...
#define TIME_STEPS ...

float T[POINTS_PER_RANK + 2]; /* explicit space for copies of neighbor data */
float d2Tdx2[POINTS_PER_RANK];
float alpha = ...;  /* diffusion coefficient */
float dx = ...;  /* size of steps between grid points */
float dt = ...;  /* size of timesteps */

int myrank, nranks;
MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
MPI_Comm_size(MPI_COMM_WORLD, &nranks);

/* set initial conditions for T[] */

for(int t = 0; t < TIME_STEPS; t++) {
  if(myrank > 0)
    MPI_Sendrecv(&T[1], 1, MPI_FLOAT, myrank-1, MPI_ANY_TAG,
                 &T[0], 1, MPI_FLOAT, myrank-1, MPI_ANY_TAG, MPI_COMM_WORLD);
  if(myrank < (nranks - 1))
    MPI_Sendrecv(&T[POINTS_PER_RANK], 1, MPI_FLOAT, myrank+1, MPI_ANY_TAG,
                 &T[POINTS_PER_RANK + 1], 1, MPI_FLOAT, myrank+1, MPI_ANY_TAG,
                 MPI_COMM_WORLD);

  for(int i = 1; i <= POINTS_PER_RANK; i++) {
    float left = T[i - 1];
    float right = T[i + 1];
    d2Tdx2[i-1] = (left + right - 2*T[i]) / (dx * dx);
  }

  for(int i = 1; i <= POINTS_PER_RANK; i++)
    T[i] += alpha * dt * d2Tdx2[i-1];
}
```

Figure 1.3: Simulation of heat diffusion in MPI

directly, such as Coarray Fortran[49]. Figure 1.3 shows the same heat diffusion application written in C with MPI. The bodies of the inner loops remain the same, but virtually everything else is different. The first indication is on line 3, which now defines the number of points per *rank*, the MPI name for one process within the collective performing the overall computation. In fact, the number of grid points in the whole computation does not appear anywhere in the code. Lines 6 and 7 declare the arrays for the properties, but only enough for the data that will be local or, in the case of the `T` array, with explicit locations for the *ghost copies* of other ranks' data that will be made during the communication stage.

Lines 13 and 14 make calls into the MPI runtime to determine the rank index of the current process and the total number of ranks. Line 18 defines the same top-level loop over time intervals, with the first operation in the MPI version of the loop being an explicit communication phase. The code implicitly assumes that ranks indices are assigned from left to right, and lines 19-21 request a data exchange (both a send and a receive) with the rank to the left, while lines 22-24 do the same with the rank to the right (if it exists). In both cases, the value sent is the one in the edge-most local element, while the value received is stored in the array location created for ghost data.

By default, MPI combines the synchronization phase with the communication phase. Each call to `MPI_Sendrecv` suspends execution of a process until the requested data exchange is complete. MPI veterans will note that the code in this example has a major performance bug that results from this automatic synchronization, with the communication phase being split by multiple synchronization phases. It is possible to separate synchronization and communication in many cases in MPI — these will be discussed more in the next chapter.

Computation phases within MPI need no decoration. They are simply code in the host language that operates only on data within the local address space. The `for` loops on lines 26 and 32 therefore only range over the grid points assigned to the current rank. In this case, no communication is necessary between the two loops, and although a synchronization phase could be forced by an explicit call to `MPI_Barrier`, it is better to omit it, effectively merging the two computation phases into one. (The UPC compiler can also perform this optimization on the code from Figure 1.2.)

As demonstrated by this code, the SPMD approach involves writing a program that does not describe the entire computation, but only captures a slice of it. The slices executed in each rank must fit together exactly to form the desired overall result. Thus even though the code is written with only a local view, the programmer must clearly still have a global view of the computation in mind. If the global view is required in both cases and compilers exist that can translate from a global view to scalable SPMD code, one might be tempted to ask why programmers would want to perform the translation themselves. One would quickly become involved in a religious debate because there are good arguments on both sides (e.g. reduced programmer effort vs. the ability to interoperate with other code written in the host language) and no objective way to rank them. A more pragmatic answer is to allow both by providing a runtime interface that is reasonably easy for humans to use but also a good target for a source-to-source compiler. A human programmer benefits from an API that limits verbosity and automatically handles obvious cases. In contrast (but critically, not in conflict), a target for compiler output should allow the compiler to express the exact behavior it intends and should present a performance model that the compiler can reason about. MPI does a good job of this, and many source-to-source compilers have chosen to target it. The design of Realm attempts to strike the same balance.

1.4 Programming Challenges

For applications with sufficient data-parallelism, the bulk-synchronous model has been incredibly successful for writing applications that make efficient use of hardware resources and scale well. However, two challenges are threatening to make matters much more complicated. Neither is new, but the severity of both has increased dramatically in recent years and is likely to increase further in the future. We will examine the effect of communication latency in Chapter 2 and of heterogeneity in Chapter 3, and the discussion will include ways in which the bulk-synchronous model has or could adapt to these challenges. With sufficient effort, any given application can likely be tuned for any current or proposed future supercomputer within the bulk-synchronous model. However, what is lost is performance portability. Each combination of application and target machine will need its own version of source code, dramatically increasing software development and maintenance costs, creating a bottleneck on the one resource that is not growing exponentially — human programmers.

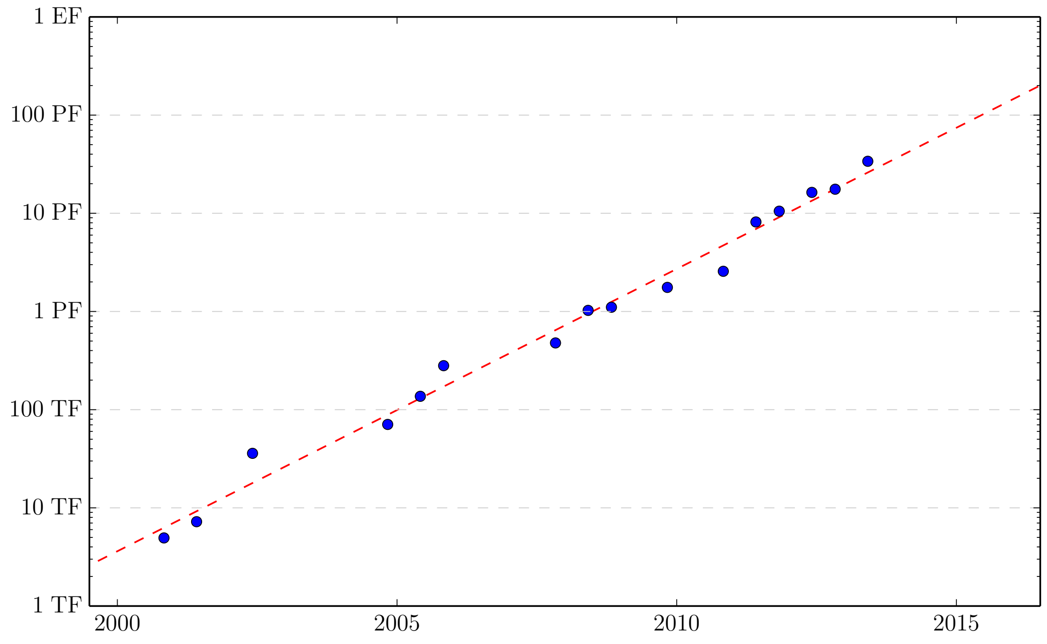
No programming system can eliminate programmer effort entirely, but Realm's composable asynchrony and abstractions for functional portability directly address these two challenges, restoring performance portability and allowing a single collection of source code to run well on current as well as future supercomputers.

Chapter 2

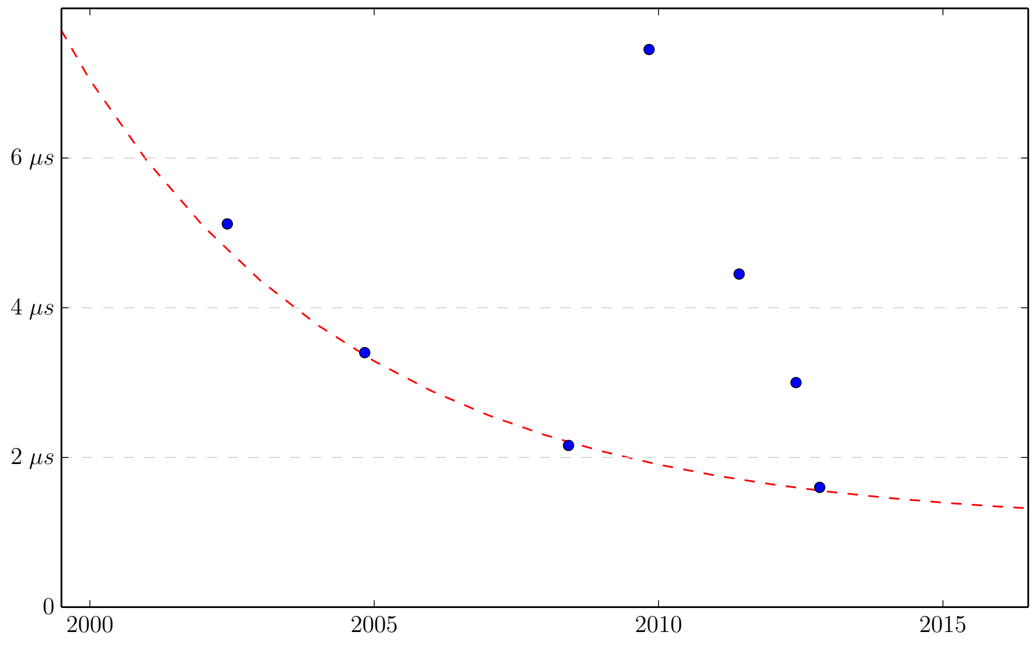
The Latency Gap

When it comes to high performance computing, the laws of physics are both friend and foe. On the one hand, they tell us that if you can make transistors smaller, they will switch faster and use less power. This has spurred ongoing investment in silicon process technology, and a steady exponential growth in transistor performance famously described by Intel's Gordon Moore in 1965. Although initially just an observation, this expectation of exponential growth has become ingrained for both computer architects and computer users. As silicon process improvements have slowed down in the past few years, researchers have looked for (and found) other ways to "preserve Moore's Law". Indeed, much of the heterogeneity challenge discussed in the next chapter is the direct result of attempts to get "moore" out of each transistor. The end result can be seen in Figure 2.1a, which shows a continued exponential trend in the performance of the world's top supercomputer, as tracked in the twice-annual Top500 list[63].

Unfortunately, transistors are not the only component of a supercomputer. To be useful, these transistors must communicate with each other. The same physics that gives us (indirectly) Moore's Law also gives us the laws of thermodynamics and electromagnetics. Transistors generate heat but also have to be kept reasonably cool (ideally around room temperature) to guarantee proper functionality. Thermodynamics places a limit on how many transistors can be placed within a given volume, and if you want more transistors than that, you must spread them apart to keep them cool. This isn't a problem on its own — Moore's Law applies to a server room just as well as to a single chip — but electromagnetics places a hard lower bound on the time it takes for transistors on opposite sides of the room to communicate. Communication also requires time spent in software and hardware, and that portion of the communication latency has significantly improved over time, as shown in Figure 2.1b. However, no improvements to either software or hardware will ever reduce the communication latency below the 500 nanoseconds it takes a photon to travel the 60 meters from one end of a supercomputer cluster to the other.



(a) Performance, as Measured by the HPL Benchmark



(b) One-Way Communication Latency

Figure 2.1: Performance and Latency of World's Fastest Supercomputers

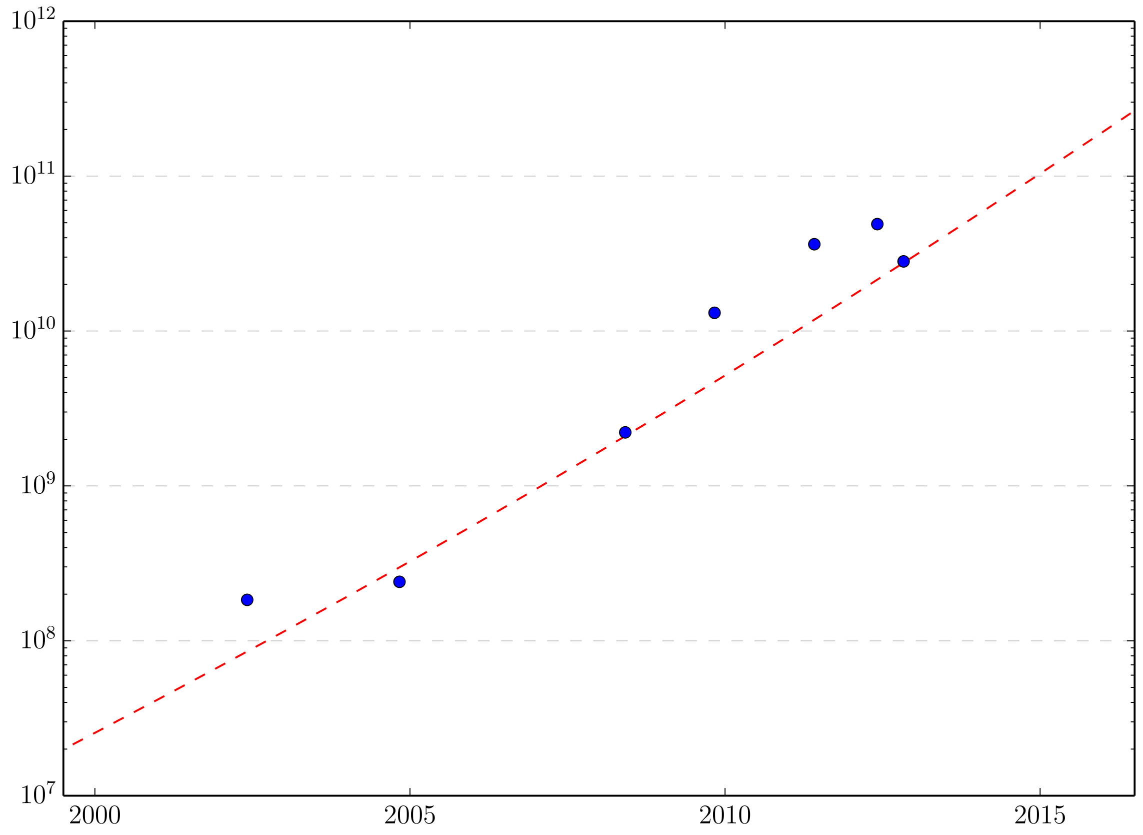


Figure 2.2: Operations Wasted by Bulk-Synchronous Communication on Fastest Supercomputers

The combination of exponential transistor performance and only asymptotically decreasing communication latency has created a *latency gap* that is fatal to the bulk-synchronous style of programming at supercomputing scales. Computation and communication are separated into distinct phases, and those transistors that are so good at computing sit mostly idle during a communication phase. By taking the product of the performance and communication latency for top supercomputers in Figures 2.1a and 2.1b, we obtain the number of potential computation operations wasted by *each* communication phase of a bulk-synchronous program (Figure 2.2). These numbers are huge and perhaps hard to put in context. One way to interpret these numbers is as the average number of instructions that must be executed in the computation phase if a program is to achieve 50% utilization of the system. If the goal is to achieve 90% (a target still considered low by many HPC users), the number is 10 times larger than that, passing through one trillion a few years ago and likely to exceed one quadrillion on the next set of supercomputers. Only the most contrived applications have sufficient data-parallelism to manage that.

```
/* initialization is the same as before */

for(int t = 0; t < TIME_STEPS; t++) {
  int nreqs = 0;
  MPI_Request mpi_reqs[4];
  /* initiate MPI sends and receives */
  if(myrank > 0) {
    MPI_Isend(&T[1], 1, MPI_FLOAT, myrank-1, ..., &mpi_reqs[nreqs++]);
    MPI_Irecv(&T[0], 1, MPI_FLOAT, myrank-1, ..., &mpi_reqs[nreqs++]);
  }
  if(myrank < (nranks - 1)) {
    MPI_Isend(&T[POINTS_PER_RANK], 1, MPI_FLOAT, myrank+1, ..., &mpi_reqs[nreqs++]);
    MPI_Irecv(&T[POINTS_PER_RANK + 1], 1, MPI_FLOAT, myrank+1, ..., &mpi_reqs[nreqs++]);
  }

  /* overlap portion of computation phase that does not need ghost data */
  for(int i = 2; i <= POINTS_PER_RANK - 1; i++) {
    float left = T[i - 1];
    float right = T[i + 1];
    d2Tdx2[i-1] = (left + right - 2*T[i]) / (dx * dx);
  }

  /* now perform synchronization phase */
  MPI_Waitall(nreqs, mpi_reqs, MPI_STATUS_IGNORE);

  /* edges can be performed only after synchronization phase is complete */
  d2Tdx2[1] = (T[0] + T[2] - 2*T[1]) / (dx * dx);
  d2Tdx2[POINTS_PER_RANK] = (T[POINTS_PER_RANK - 1] + T[POINTS_PER_RANK + 1] -
                             2 * T[POINTS_PER_RANK]) / (dx * dx);

  for(int i = 1; i <= POINTS_PER_RANK; i++) {
    T[i] += alpha * dt * d2Tdx2[i-1];
}
```

Figure 2.3: Simulation of heat diffusion with latency hiding

2.1 Hiding Latency

Although the current “hyperinflationary” period is a recent occurrence, the risk that communication latency presents to bulk-synchronous programming has been apparent to programmers and researchers for a long time. Every HPC programmer is taught that they need to *overlap* their communication and computation to hide the latencies in modern systems. Virtually every language or runtime for bulk-synchronous programming has been extended to allow this overlap in some form. We will start with a concrete example of how overlap is achieved, and then generalize to discuss the merits and drawbacks of the approach.

Recall from the previous chapter that communication primitives in MPI include synchronization phases by default. MPI allows the synchronization to be separated from the communication through the use of *non-blocking* versions of most primitives[1]. Figure 2.3 shows the timestepping loop of our running heat diffusion example, implemented with these non-blocking primitives. The first indication that non-blocking MPI communication is being performed is the declaration of an array of `MPI_Request` objects on line 5. These objects are used to track the state of overlapped communication. Lines 7-14 use `MPI_Isend` and `MPI_Irecv` to initiate the exact same sends and receives that were performed by the `MPI_Sendrecv` calls before. (For some reason, there is no `MPI_Isendrecv` in MPI.) Each call fills in its own `MPI_Request` object, and the `nreqs` variable is used to keep track of how many there are. These calls return immediately (i.e. they do not *block* the caller), allowing the data transfers to proceed concurrently with additional application code that appears before the synchronization phase.

Because it is overlapped with the communication, that portion of the application code must not read from the memory locations that are receiving data nor write to the locations from which sends are occurring. As it is only a runtime library, MPI cannot check that these rules are followed. When the rules are not followed, the resulting bugs cause non-deterministic failures that can be very challenging to fix and often remain latent in applications for years, just waiting for a system with the right performance characteristics to appear. In this case, only the first and last iterations of the stencil-calculating loop need the data from the communication phase, and by restricting the loop bounds slightly, the bulk of the loop (lines 17-21) can safely be computed concurrently with the communication phase.

Line 24 performs the synchronization phase, asking the MPI runtime to wait until the non-blocking communication has completed. Once the call to `MPI_Waitall` has returned, it is safe for lines 27 and 28 to read from `T[0]` and `T[POINTS_PER_RANK + 1]` and for line 32 to overwrite the values sent. In a perfect world, enough time was spent doing useful work in lines 17-21 that the call to `MPI_Waitall` can return immediately, keeping the processors fully utilized.

The key observation that makes latency hiding work is that data-parallelism is not the only kind of parallelism available in most applications. Different operations performed by the application may be performed concurrently if they do not *depend* on each other. The most obvious form of dependency is a data dependency (e.g. reading the value written by the other operation), but many other forms of dependencies can exist, such as a conflict on a common software or hardware resource. Using the generic term *task* to describe some collection of operations, the availability of independent tasks in an application is called *task-parallelism*.

Our initial implementation of the heat diffusion simulation could be considered to have two tasks (stencil computation and temperature update), but lacked any task-parallelism. However,

[1]These primitives are sometimes described as being "asynchronous". We follow MPI's lead here and stick with "non-blocking", in part because we argue these operations are still carefully synchronized with the calling code, and in part because we wish to reserve the "asynchronous" term for a qualitatively different approach to come.

by splitting the single stencil computation into two tasks, one operating on the interior and one on the boundary, we were able to expose the independence between the interior stencil and the preceding temperature update tasks on other ranks. The boundary stencil still depends on the remote temperature updates, but the interior computation can proceed while the boundary one waits for communication and synchronization with the other ranks. This pattern of exposing task-parallelism through an interior/boundary split is a favorite in HPC applications, in part because it is easy for compilers and/or libraries to assist with the transformation.

While data-parallelism in an HPC application tends to scale with the number of elements, task-parallelism tends to scale with the number of properties that exist per element. For today's and future supercomputers, neither task-parallelism nor data-parallelism is sufficient. A programming model must take full advantage of both to bridge the latency gap.

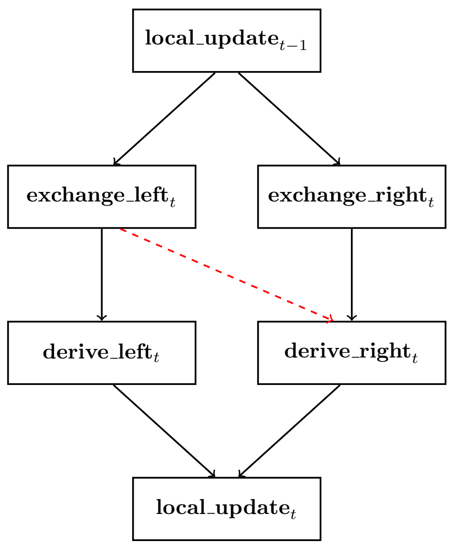
Automatic identification of dependencies between tasks is intractable in general languages such as C, C++, and contemporary Fortran[2], and assistance from the programmer is needed. There are many different ways in which models ask the programmer to describe the dependencies that exist between operations within the application. The property we focus on here is whether the dependencies themselves are *implicit* or *explicit*. As the name suggests, a model using *explicit dependencies* provides language and/or runtime constructs to explicitly name pairs of dependent operations. In contrast, a model based on *implicit dependencies* is one in which the dependencies are not spelled out, but are instead implied by other language or runtime constructs, often related to scheduling. We discuss both in more detail below, as the choice between implicit and explicit dependencies is not just a matter of aesthetics for the programming model — it impacts the performance portability of applications written in that model.

2.2 Implicit Dependencies

Models based on implicit dependencies are easily identified through their use of non-blocking operations. Constructs are provided that allow the caller to initiate work and then to wait on the completion, ideally after performing enough other work to hide the latency of the non-blocking operation. This approach has two major benefits. First, it is a relatively simple model for programmers to understand. Delegating work to another (and often needing to wait for the work's completion) is something humans do on a daily basis. Second, the introduction of non-blocking operations is an evolutionary path that allows a language or runtime that lacked latency hiding capabilities to add it in a minimally disruptive way.

Unfortunately, having an application wait on the completion of one operation so that it can safely execute another is a fundamental flaw that prevents code written in these languages or runtimes from being performance portable. To understand why, it helps to look at how the dependencies between

[2]Earlier versions of Fortran were much more amenable to compiler analysis, and there are some that advocate for a return to such simpler days.



(a) Dependency graph

```
1   for(int t = 0; t < TIME_STEPS; t++) {
2     MPI_Request req_left, req_right;
3
4     // exchange with left and right neighbors
5     exchange_left(..., &req_left);
6     exchange_right(..., &req_right);
7
8     // compute derived fields for left ghost
9     MPI_Wait(&req_left, ...);
10    derive_left(...);
11
12    // compute derived fields for right ghost
13    MPI_Wait(&req_right, ...);
14    derive_right(...);
15
16    local_update(...);
17  }
```

(b) Application code

Figure 2.4: Undesired scheduling constraints resulting from implicit dependencies

tasks are described in these systems. As we saw in the MPI example above, if a dependency exists between two tasks, a correct application must wait on the completion of the first task sometime before executing the second. However, the converse is not universally true. The fact that a task is executed after the application has waited on an earlier task does not guarantee a dependence exists. As an example, consider a minor variation of our MPI example in which the local computation uses fields derived from the ghost values rather than the ghost values themselves. The derivations for different ghost cells are independent, but there is no way to precisely express that independence with application waits. Figure 2.4a shows the desired dependency graph with solid black arrows, but the ordering of the code (Figure 2.4b) implies the undesired red arrow as well, as the call to `derive_right` on line 14 follows the wait on `req_left` on line 9. If the exchange of the right ghost cells is likely to complete before the exchange of the left, the red arrow can be switched by moving lines 13-14 above lines 9-10, but no ordering of the code will be free of undesired dependencies.

As a result, a system in which application waits are used to implicitly encode dependencies between tasks has very little freedom to make scheduling decisions, because the system has no way of knowing which tasks before and after a wait are dependent. Virtually all decisions must be made by the programmer. Worse, the decisions are implemented by a manual reordering of the application code and wait operations. For applications that were ported to Titan as part of the acceptance testing, this manual code restructuring was estimated to account for 70-80% of the person-years of developer time per application[18]. Quite often, the pieces of code being reordered come from different parts of the source code, requiring a large amount of effort and destroying the modularity of the code. Finally, if a different scheduling of tasks is needed for two different systems (one might have faster processors, while the other has a faster interconnect), two different versions of the source code are needed to capture those two schedulings.

2.3 Explicit Dependencies

If one is willing to step outside of the standard bulk-synchronous paradigm, a significant portion of the problem of scheduling tasks to hide latency can be moved to the compile and/or runtime in a system that uses *explicit dependencies*. When every task is explicitly annotated with the set of tasks on which it depends, the order in which tasks are launched by the application does not limit the possible schedules. A single version of the source code describes all valid schedules, and the compiler or runtime has the maximum possible freedom in selecting a schedule that will run most efficiently on a given system.

A natural way to consider an application's collection of tasks and their dependencies is as a graph, with nodes in the graph representing tasks and edges between nodes representing explicit dependencies between two tasks. When designing a model based on explicit dependencies, the first decision that must be made is whether than graph is constructed and analyzed at compile time, at runtime, or some combination of the two. Most systems that use explicit dependencies fall at the more static (i.e. compile-time) end of the spectrum. Fully static approaches use the operation graph within the compiler, generating code for machine (or sometimes lower-level runtimes) that are unaware of the boundaries between tasks and their dependencies. By doing all the analysis and optimization at compilation time, any runtime overhead due to the use of explicit dependencies can be minimized. However, that analysis (and the optimizations that result from it) suffers from the standard limits of static analysis: the behavior of application code is often impossible to capture precisely, especially when it might depend on the exact inputs to a given run.

Some of those limitations can be addressed by augmenting the compile-time static analysis with some dynamic analysis at runtime. For example, the compiler's analysis might be limited to smaller subgraphs which may be instantiated at runtime based on the data set being operated on.

The dynamic end of the spectrum, in which task graphs are constructed and analyzed entirely at runtime, is much more sparsely populated. A heavy bias towards compile-time analysis is natural in the high performance computing space, where applications tend to be regular and run for a long time. Compile-time overhead that can be amortized over thousands or millions of iterations is strongly preferred over run-time overhead that is incurred on every iteration. However, as applications and machines both become larger and more dynamic, the limits of static analysis create a gap between theoretical and achieved performance. As that gap grows, so does the willingness of an HPC user to accept some run-time overhead in exchange for improvements in scheduling efficiency, programmer productivity, and of course portability.

A fully dynamic approach based on explicit dependencies must deal with several challenges. First, and most obviously, the runtime overhead must be as low as possible. The costs involved in creating, storing, analyzing, and ultimately destroying the operation graph must be small, and must scale well to very wide and/or very deep task graphs. The common strategy used by all of the runtimes discussed in the next section is to execute the task graph as it is being constructed, keeping only a frontier of the graph that ideally includes only the tasks that have not yet been executed. Without the whole graph available, any scheduling (what to run next) or mapping (where to run it) decisions made by the runtime must necessarily be local decisions.

This leads to another major challenge for dynamic approaches. The frontier of the task graph must include enough parallelism that good local scheduling or mapping decisions are even possible. The key to this is to make sure the application is able to "run ahead" of the actual execution, enumerating tasks in a programmer-friendly way (e.g. depth-first traversal of one components tasks before moving on to the next component). Explicit dependencies allow the enumeration and execution of tasks to be asynchronous, but the maximal benefit is obtained when all dependencies are explicit. Any application dependency that cannot be expressed explicitly requires some other task to wait before launching it, and this wait brings back all the drawbacks of the implicit dependency model. The programmer must again reason about what parts of the task graph might be hidden behind the wait and whether they need to be manually reordered to expose independent work in the task graph's frontier to fully utilize the system.

2.4 Microprocessor Design Analogy

The concepts of implicit and explicit dependencies can be seen in many other fields. For those who study microprocessor design, they are familiar in the difference between *in-order* and *out-of-order* architectures. In nearly all microprocessor designs, a pipelined architecture is used to improve the processor's clock speed while maintaining high instruction throughput. As a result, multiple instructions will be in different stages of execution at the same time. A later instruction may depend on the result of a previous instruction that is still in the pipeline, and must *stall* until the data is ready.

Designs that use in-order execution suspended the entire instruction stream in such a case, stalling all following instructions as well. Based on knowledge of the pipeline depth, a compiler that generates code for an in-order architecture can usually select an order of instructions that minimizes the stalls for results of arithmetic instructions, but inputs that come from memory loads depend on cache behavior and are hard to predict at compile time.

As the cost of a cache miss grew (another latency gap!), microprocessor design moved to out-of-order execution. In these designs, an instruction whose input data was not ready could be set aside, allowing instructions later in the instruction stream to be decoded and hopefully executed.

An instruction that is set aside is annotated (a variety of techniques exist, including many based on Tomasulo's algorithm[62]) with the exact list of earlier instructions that must complete before the later instruction can be dispatched. This choice between an in-order and out-of-order design requires making the same sorts of trade-offs between performance, cost, and portability that we have discussed here.

2.5 Related Work

Virtually every language or runtime system used for high-performance computing includes mechanisms that attempt to address the problems caused by the latency gap. Most are based on implicit dependencies. Our examples of non-blocking communication in this chapter used the MPI[58] runtime, but other communication libraries such as GASNet[66] handle it similarly.[3] Languages that use implicit dependencies include the previously-discussed UPC[19], Chapel[22], and X10[23] as well as others such as Cilk[11], Titanium[67], and the Habanero family of languages[21]. Runtimes that make use of GPUs as accelerators, either for graphics (e.g. OpenGL[37] or Direct3D[47]) or for general-purpose computation (e.g. CUDA[50] or OpenCL[38]), all use implicit dependencies to hide the latencies associated with initiation work on and communicating with the GPU. Syntax and, more importantly, the choice of which operations are available in non-blocking form, can vary quite a bit between these languages and runtimes, but the basic approach of initiating a non-blocking operation and then some time later waiting on its completion is common to all.

For explicit dependencies, the spectrum between static and dynamic handling of the dependency graph allow for much more diversity. The static end of the spectrum includes Sequoia[33] and Deterministic Parallel Java (DPJ)[12]. DPJ uses the graph to identify sections that can be run on separate threads managed by the operating system. In contrast, Sequoia uses a performance model of the target machine to compute an exact execution schedule for all tasks and data movement during compilation.

Many systems perform dataflow analysis (generally coarse-grained) to extract what amount to templates for subgraphs of the task graph, allowing efficient instantiations of those templates at runtime to construct the overall graph[36, 59, 2, 26]. Concurrent Collections[15] makes these dataflow templates an explicit part of the language. Most of these systems do include some runtime component, but that component is not directly aware of the task graph.

At the fully dynamic end of the spectrum, there are three major research efforts in addition to Realm. OmpSs[31] uses a custom compilation flow that adds directives to existing host languages (C, C++, Fortran) that allow pieces of code in a function to be *outlined* into separate tasks, which may execute concurrently with the caller or with each other. Tasks are annotated with their inputs and outputs. Based on the directives, the compiler generates code that performs calls into the OmpSs

[3]Realm uses only a subset of the GASNet API calls to avoid the pitfalls associated with implicit dependencies.

runtime to launch tasks and provide the locations of the task's inputs and outputs. The runtime can then compare the locations with that of other tasks to determine the inter-task dependencies.[4] The OmpSs runtime includes a scheduler that tracks which tasks are ready to run and selects from that set when processor(s) become available. OmpSs supports execution on multiple systems in a cluster, but does so using a master/worker model in which all task graph analysis is performed on the master[60].

StarPU[3] is a purely library based runtime. The programmer of a StarPU application writes in C or C++ and manually inserts calls into the StarPU runtime to create tasks, to attach buffers for input and output, and to submit the task for execution. As with OmpSs, the StarPU runtime deduces the explicit dependencies between task based on accesses to common buffers, and then performs both scheduling and mapping of the tasks once those dependencies have been satisfied. Originally designed to address challenges related to heterogeneity (which we will discuss in the next chapter), a StarPU runtime only manages a single node. For an application to run on a cluster, a so-called *hybrid* programming model is used, combining StarPU with MPI. In addition to the other challenges with a hybrid model (discussed in the next chapter), this approach limits the benefits of using explicit dependencies to just tasks that run on the same node.

A recent extension to StarPU, StarPU-MPI[1] encapsulates the hybridization by allowing StarPU tasks to attach input buffers that may reside on other ranks. The application may choose an *owner* for each buffer (this is similar to Chapel's *domain maps* and more expressive than X10 or UPC's distribution mechanisms) and uses the standard *owner-computes* model in which tasks are executed on the rank that owns the output buffer(s). The StarPU-MPI runtime on each rank automatically inserts the necessary MPI communication and synchronization primitives to connect with producer or consumer tasks on other ranks, but to do so, it must know which tasks those are. This forces a deviation from the single-program multiple-data model, as each rank must include those remote producers and consumers as part of its program. (The runtime knows not to execute them due to the owner-computes rule.) The safest thing to do is to give each rank a global view of the program's task graph, but scalability results show that *pruning* of the graph on each rank is absolutely necessary for running on systems larger than 1000 nodes or so. Unfortunately, this pruning must be done by the programmer and is vulnerable to the same sorts of missed synchronization bugs that occur in implicit dependency models.

The third active effort in the space of runtime systems based on dynamic explicit dependencies is the Open Community Runtime (OCR)[52]. OCR is also library based and does not directly include a compiler component. However, it is not designed for direct use by human programmers either. Instead, it is expected that OCR programs are the output of source-to-source compilation

[4]One might argue that the task dependencies are only implicitly described through these data interactions. However, the key distinction we are interested in is whether a "producer" and a "consumer" are directly connected through an explicit linkage or if their connection is only implied by connections of both producer and consumer back to their parent task.

from other programming languages. For example, OCR completely prohibits the use of application waits (except for debugging purposes), demanding the use of a *continuation-passing-style* of code that is easy for compilers to mechanically produce but very laborious for humans. In addition to explicit dependencies based on use of common *data blocks*, OCR includes several kinds of explicit event objects that can be used to define other dependencies between tasks. OCR is intended to work for clusters but assumes a homogeneous system architecture. It presents a logically centralized scheduler that has a global view of the system and manages the mapping of tasks and any necessary data movement.

All three of these systems focus on explicit dependencies between tasks, but lack the ability to describe explicit dependencies between tasks and other operations such as memory allocation or explicit data movement. In some cases, this shortcoming can be addressed by encapsulating the other operations inside a task (at some cost). In others, this lack of composability results in application waits to guarantee correctness.

Chapter 3

Heterogeneity

The second major challenge faced by the bulk-synchronous programming paradigm is the dramatic increase in the complexity and critically, heterogeneity, of both supercomputer system architectures and the workloads that are being run on them.

When supercomputers first made the transition from large custom machines to clusters, the individual nodes were generally very simple. They contained a single processor and a single pool of memory, and every node was identical. The interconnection networks were also designed in a way that made them appear as uniform as possible. Infiniband-based networks would connect nodes and routers in a *fat tree* such that any pair of nodes would have the same peak bandwidth and message latency as any other pair. The fat tree topology added significant cost relative to other reduced (but still fully-connected, of course) topologies, but the benefit was a simple performance model for the system in which data was either local or remote and in which the nodes themselves were interchangeable. The only system architecture parameter that mattered was the number of nodes on which the application was running. Even better, an application designed on one system would generally run well on any other systems. Although the ratios of processor to interconnect performance might vary between systems, the performance difference between accessing local data vs. remote data was so large that algorithmic changes to the application were not needed. This simplicity was appreciated by both HPC programmers and the providers of compilers and runtimes for these systems.

Over time, the nodes and the interconnect became more complicated. To provide additional performance, processor manufacturers started putting multiple processor *cores* into a single package, and system designers started putting multiple processor packages into a single node. In some cases, these multiple processors would share the same memory pool, forming *Symmetric Multi-Processor* (SMP) architectures. In others, multiple pools of memory were included, with a small communication network between the processor cores and the pools of memory. For cost and performance reasons, these intra-node networks were not made like fat trees, and the question of how to best deal with

Non-Uniform Memory Architectures (NUMA for short) raged on for much of the 1990's.

A programmer (or system software) that wanted to reason about the performance of a NUMA-based cluster now had to worry about the number of nodes, the number of processors per node, the number of memory pools per node, and the different *affinity* each processor had to each memory pool in the node. Worse, the differences in affinities in different architectures resulted in cases where algorithmic changes were required when moving an application from one supercomputer to another.

Although many other strategies were considered, the most effective strategy found for dealing with NUMA in HPC applications was to ignore it. Instead of dealing with complicated nodes with multiple processor cores and multiple memories, each complicated node was divided into multiple logical nodes. Each logical node was given exactly one processor core and a fraction of the pool of memory with the best affinity to that core. The programmer was able to go back to just thinking about local and remote memory and applications written for older architectures could now run unchanged.

3.1 Hybrid Approaches

This "rank per core" strategy remains in widespread use on some systems even today, but there is a growing class of systems for which it for which it cannot be used. These are systems that contain heterogeneous processors (i.e. processors of two or more completely different designs). The use of heterogeneous processor types in a single system has a long history, and is based on the idea of specialization: a processor that is specialized for a particular kind of workload (e.g. floating-point arithmetic) can often provide significantly better performance on that workload than a general-purpose processor. The downside is that a specialized processor may be much less efficient at (or in some cases, incapable of) performing other workloads. As a result, systems that include specialized processors almost always include a general-purpose CPU, and employ a *coprocessor* architecture in which the activities of the specialized processor are managed by the the general-purpose one.

In recent years, "heterogeneity" has mostly meant "GPU computing", in which the specialized processors are *graphics processing units*. Originally designed to accelerate 3-D rendering using APIs such as OpenGL[37] or Direct3D[47], the computational horsepower of GPUs began to exceed that of the CPUs in the system, and efforts began to make it accessible to applications other than 3-D rendering. Initially, these efforts had to map these applications back onto the graphics APIs (e.g. Brook[14]), but APIs now exist to allow more general access: CUDA[50], which is specific to GPUs from NVIDIA, and OpenCL[38], which attempts to be a vendor-neutral specification. However, even these newer APIs maintain the coprocessor model — the resources of the GPU and execution on it are managed by the application and/or driver code that runs on the CPU.

In early GPU-enabled systems, a large performance disparity existed between the GPU and the CPU cores in one direction or the other. Either a given task was well suited to the specialized

GPU architecture and performed significantly better than on the CPU cores, or it was not and performed significantly worse. This encouraged a model in which an application used either the CPU or GPU for any given task, switching back and forth as the workloads changed. However, with many fewer GPUs (often only one) per node than CPU cores, the "rank per core" strategy cannot be used. To ensure that each process had exclusive access to a GPU for the compatible workloads, a "rank per node" (or occasionally "rank per GPU") strategy was necessary. In these strategies, multiple CPU cores were assigned to the same rank, and a second mechanism was needed to fully utilize the CPU cores for the portions of an application that were unsuited to the GPU. The combination of one system for inter-process parallelism across nodes (or GPUs) and another for intra-process parallelism across CPU cores in a single process is known as the "hybrid" strategy. An application written using the hybrid strategy must carefully manage the interactions between the two systems, and although each system may provide mechanisms for hiding latency, they are unlikely to be composable. Additionally, the problems with NUMA described above could no longer be ignored.

Luckily, the handling of heterogeneous processors remained fairly manageable for these systems under a hybrid strategy. With a hybrid application effectively only working on one task at a time, adding support for for a new coprocessor type was a matter of deciding which of the tasks should be offloaded, and adding implementations of those tasks to the executable. While possible to do this completely manually, two main techniques exist for reducing the programmers effort. Programming models with compiler support can attempt to compile a common piece of source code into machine code for each possible processor type. Another approach is to allow (or force, depending on your perspective) the programmer to write variants of a task for each processor type in a native programming model for that processor that can achieve maximum performance. Once these variants are written, runtimes can provide mechanisms to link the variants together behind a single entry point for the caller.

3.2 Titan

The "discrete" heterogeneity of early systems did not last. The architecture of heterogeneous supercomputers has evolved from the early GPU-enabled systems, and these changes present significant challenges for the programmer of an HPC application. To illustrate these changes, we look at Titan, the world's fastest supercomputer when it went online in 2012. (It remains the third fastest as of late 2016.) Figure 3.1 shows the high-level system architecture.

Each Titan node contains two kinds of processors. There are 16 general purpose CPU cores, depicted by the array of "fat" blue boxes in the top left. In part due to the challenge from GPU accelerators, CPUs have become significantly better at floating-point computation, and Titan with just its CPUs would have been the 8th fastest supercomputer in the world in 2012. However, Titan

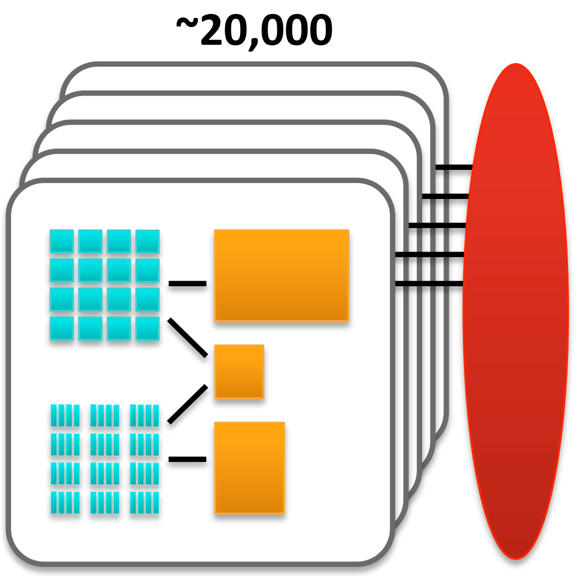


Figure 3.1: System Architecture of Titan (Oak Ridge National Laboratory)

also included the most powerful GPU at the time, a Kepler GPU from NVIDIA. The Kepler GPU is made up of a large array of "thin" cores, shown in the bottom left. GPUs have also seen many architectural improvements since their first use as accelerators, making them suitable for many workloads beyond 3-D rendering and the "stream processing" of Brook. This trend towards more number-crunching in CPUs and more general programmability in GPUs results in a situation where many workloads can run well on both processor types, and an application that runs a task on only one processor type is leaving significant performance on the table. However, they remain different enough in both capability and how they are used that they cannot be treated as one. As a result, applications must now decide if they have enough task-parallelism to run different tasks concurrently on different processor types or whether they need to take advantage of data-parallelism and *load-balance* the workload of a single task across all processors. In both cases, the optimal mapping and scheduling depends on the exact ratio of performance provided by each type of processor for each particular workload. These values can be very challenging to predict and worse, can vary wildly between two systems with the same architecture at the "block diagram" level. Chapter 8 will demonstrate one way Realm can help an application solve the load-balancing problem, while Chapter 11 will include an example of how a quantitative difference in relative performance of processor types can require a qualitatively different mapping of an application.

The second major programmability challenge posed in recent heterogeneous systems is related to the connections between the processors and memory within the system. In Titan (and virtually all other high-performance heterogeneous systems), the CPU and GPU each have their own dedicated memory. The *host memory* used by the CPU is reasonably fast and large. Titan has 32 GB of host

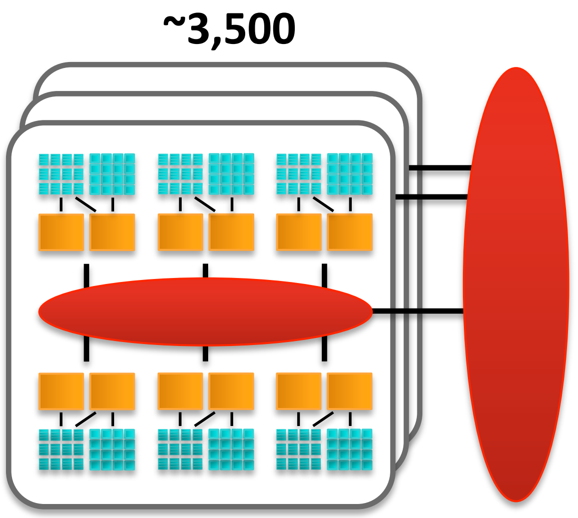


Figure 3.2: System Architecture of Summit (Oak Ridge National Laboratory)

memory per node, for a total of 584 TB across all nodes on the system, and is sufficient for virtually all HPC applications. The GPU's *device memory* (often still called the frame buffer, showing its graphics heritage) is much faster, but also much smaller. Titan only has 6GB of GPU device memory per node, and many applications are forced to move data back and forth between device memory and host memory. This is complicated by the inability of either processor to efficiently access the other's memory. Instead a small amount of the host memory must be specifically registered for *zero copy* access from the GPU. To avoid unnecessary copies between the zero copy memory and the rest of the host memory, tasks running on the CPU must be aware of whether they are reading data from, or producing data for, a task that will run on the GPU and if so, access the zero copy memory instead. An application can no longer make mapping decisions for each task in isolation. Additionally, the GPU's access to the zero copy memory must travel across the PCI-Express bus that connects the GPU to the rest of the system, and the bandwidth and latency of this connection are not significantly better than that of the interconnection network between the nodes. In effect, a latency gap has opened up between the CPU and GPU even in the same system, requiring an application (ideally with assistance from the runtime) to overlap computation on both the CPU and GPU with any data movement between their memories.

3.3 Heterogeneous Heterogeneity

The challenges in programming heterogeneous systems like Titan came as no surprise to the system's architects. Oak Ridge established its Center for Advanced Application Readiness three years before Titan's arrival, and is continuing the effort for their new system, offering important applications a

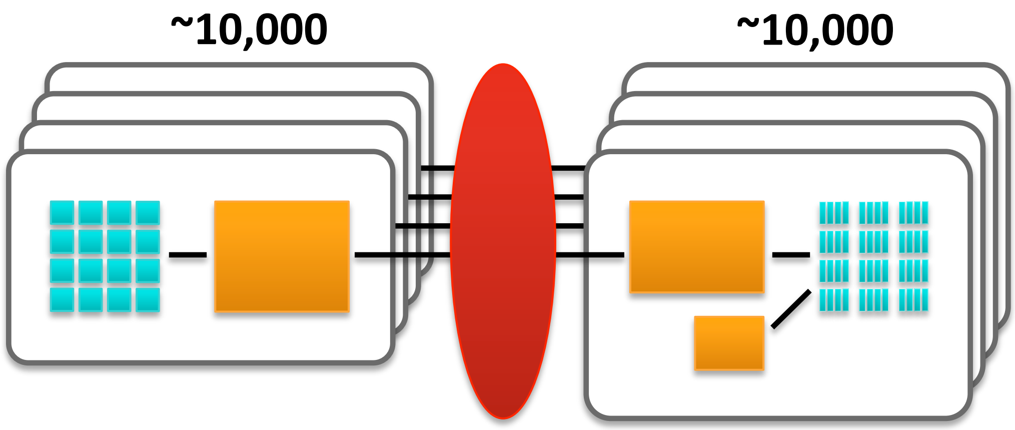


Figure 3.3: System Architecture of Trinity (Los Alamos National Laboratory)

full-time programmer for two years (on top of the application's own development staff)[17]. The complexity of the system was an unavoidable consequence of the performance target Titan was trying to meet (i.e. the #1 spot on the Top500 list, among other metrics).

This mandate to keep up with Moore's Law demands ongoing architectural innovation, and HPC application programmers will have to deal with at least three radically different new architectures in the next couple years. None of these systems exist yet, so while there may be some fear, uncertainty, and/or doubt around how hard they will be to program, each contains new challenges relative to a system like Titan. An application that wishes to be performance portable must address all of them.

The first system we discuss is Summit, which will be the replacement for Titan at Oak Ridge National Laboratory[29]. Shown in Figure 3.2, Summit continues to combine CPUs with CUDA-capable GPUs. However, not only will it use significantly more powerful versions of both, there will be multiple multi-core CPUs and multiple GPUs within a node. This dramatic increase in processors necessitates an intra-node interconnect that is distinct from the inter-node interconnect. It's possible that an application will be able to treat the two interconnects as one (i.e. a return to the "rank per core" strategy, but with much larger "cores"), but doing so will lose the ability to take advantage of the fact that the connections within a node will be 4-8x faster than the ones between nodes.

A very different system is coming to Los Alamos National Laboratory in the form of Trinity[28]. This system (shown in Figure 3.3) continues to have general purpose CPUs, but instead of GPUs, uses the Xeon Phi processor from Intel. This is a many-core processor that can run any code that a CPU can (importantly for this architecture, it can run the operating system), but provides additional instructions that use a very wide *vector datapath* that operates on 16 or 32 values in parallel. The major change is that instead of having processors of each type in each node, Trinity will have heterogeneous nodes. Some nodes will be CPU nodes and others will be Xeon Phi nodes.

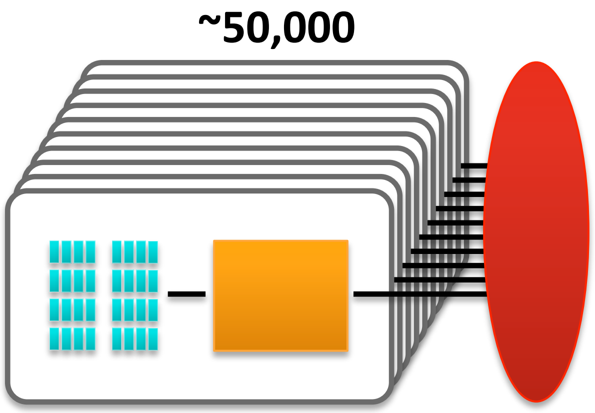


Figure 3.4: System Architecture of Aurora (Argonne National Laboratory)

With Trinity, the heterogeneity of the system can no longer be contained in the inner half of a hybrid strategy. A second challenge with Trinity is the Xeon Phi's use of two different types of memory: a small pool of fast memory and a larger pool of slower memory. Although similar in some respects to the problem of managing device and host memory on a system like Titan, Trinity's memory hierarchy may present challenges for programming models that reason only about processors, assuming each processor has a single preferred memory.

The third system is one that, on paper, looks to be much easier to program. Shown in Figure 3.4, the Aurora supercomputer at Argonne National Laboratory will use only a single kind of node, with only a single kind of processor, the successor to the Xeon Phi processor being used for Trinity[30]. For Aurora, a single piece of compiled code will be able to run anywhere, but the challenge will come instead from finding enough parallelism to keep the system busy. Aurora will consist of at least 50,000 nodes, and require "multi-million-core computing." Many applications will lack sufficient data-parallelism to fill the machine with a bulk-synchronous model. Task-parallelism will be needed, and it will likely need to happen at a node level (i.e. some nodes working on one task while others work on another) to keep data movement between nodes from becoming the bottleneck.

3.4 Generalized Heterogeneity

While daunting, the heterogeneity coming to high performance computing in the form of Summit, Trinity, and Aurora is arguably just the tip of the iceberg. Many other qualitatively different approaches to building a supercomputer have been proposed, considered, or even prototyped. These approaches can offer significant performance improvements and, in many cases, the barrier to adoption is less related to cost or engineering concerns and more due to a fear that it will be too hard for applications to make use of the new architecture. We offer a few examples here, but the architectural design space for supercomputers is incredibly large and it is trivial to find many more examples.

A first example has to do with the host CPUs in modern supercomputers, which can crunch application data at an impressive rate, but must also run all the non-performance-critical parts of an application, such as I/O or bookkeeping operations. They are massively overqualified for such jobs, but nothing else is available to run such code. The introduction of lightweight cores to cheaply perform these operations and free up the main CPU cores for computational tasks would yield noticeable improvements for negligible additional cost, but it introduces a third type of processor (and possibly another kind of memory) that applications must deal with. Some have even suggested putting these lightweight processors closer to storage and or network devices, further increasing both potential benefits and programming challenges. There is constant research into other kinds of specialized processors (e.g. for image analysis, graph processing, or deep learning) as well. All of these would likely be additions to the processor mix on systems rather than replacements for one of the current two processor types.

Another area in which architectures are trying to move is to turn coprocessors such as GPUs into peers of the CPU. This allows more efficient management of resources and operations on the GPU, but represents a usage model fundamentally at odds with the accelerator approach used by some models. The Xeon Phi roadmap is a good example of this. The Knights Corner coprocessor used in conjunction with standard Xeon CPU cores was replaced with the “self-hosted” Knights Landing processor, but Trinity’s architecture splits those processors into separate nodes rather than having them share the same node as Xeon CPUs.

A third active area of exploration is in interconnection networks that blur the lines between nodes in the cluster. Especially interesting are the ones that include multiple network connections per node and spread them around the node’s non-uniform memory hierarchy. The result can be systems in which a processor in one node might have better affinity to some processor in another node than to some of the other processors in the same node. This may already be a reality in the form of NVIDIA’s SATURNV supercomputer[51].

3.5 Related Work

Existing efforts to handle the heterogeneity can be classified in three ways: how they handle code generation, how (or if) they expose the memory hierarchy in the system, and whether they handle heterogeneity within a node or across a cluster. As described above, the question of code generation usually comes down to whether a model is language-based or runtime-based. Models such as OpenACC[53] and OmpSs[31] make use of compiler support to extend standard languages (e.g. C or Fortran) with constructs or directives that request code be generated for a different processor type. Others define their own language (often a more GPU-friendly one) for tasks, but then generate code for both CPU and GPU for those tasks. OpenCL[38] falls into this category. The ideal for both types of code generation is that a single piece of source code will run well on all types of processor being targeted, but the differences between CPU and GPU architecture (or between different GPUs) are so large that performance-focused programmers usually settle for the partial victory of at least being able to write the variants in the same language. Runtime systems that do not have compiler support (e.g. StarPU[3] or the underlying Nanos++[60] runtime used by OmpSs) focus instead on the linkage approach, accepting a different variant (or perhaps multiple) of the task for each processor and making sure the correct one is invoked at runtime.

The second question, how the memory hierarchy is exposed, is effectively binary for current approaches to heterogeneity. Systems such as StarPU and OmpSs use the same information about task's use of data that defines inter-task dependencies to automatically move data between memories as needed, hiding the complexity of the memory hierarchy from the programmer. StarPU-MPI is able to extend this across nodes in the cluster as well. The alternative approach used today presents a model in which each processor and memories are tightly coupled. This is apparent in the host and device memories exposed by OpenACC, CUDA, and OpenCL. It can also be seen in the tree-shaped memory models used by Sequoia and the hierarchical place trees used by the Habanero languages[65]. Both create virtual processors to match up with the higher nodes in the tree, maintaining an isomorphism between processing elements and storage locations.

The final question, whether the heterogeneity is handled within or across nodes, also sees a roughly even split of answers. Systems like OpenACC and StarPU follow the hybrid approach, letting HPC programmers take advantage of the familiarity and performance of MPI for inter-node data movement and synchronization, accepting that application waits will be required where the two models meet. Others, including OmpSs and StarPU-MPI, handle inter-node data movement within their model, allowing dependencies between such data movement and asynchronous task execution to be captured and used for dynamic scheduling decisions.

Chapter 4

Composable Asynchrony

In the previous two chapters, we have explored the two major obstacles to achieving performance portability for high performance computing. The latency gap (Chapter 2) demands the exposure of task-parallelism in applications and a way of scheduling those tasks that satisfies dependencies between the tasks while making the best use of the computational and communication resources of a target system. The increasing heterogeneity of supercomputers (Chapter 3) results not just in the need for multiple versions of computational tasks, but the combinatorial explosion in the ways the APIs for each type of processor or memory interact.

Although they have been presented as two different issues, they interact with each other in several ways. The latency gap is one of the driving reasons for today's distributed memory hierarchies, as each processor needs at least some memory it can access with low latency. In the other direction, the plethora of different APIs and different execution models complicates the scheduling of tasks by preventing dependencies that cross heterogeneity boundaries from being precisely captured. A solution for performance portability must therefore address both the issues of latency and heterogeneity to be effective.

The core concept behind Realm's novel approach to performance portability is *composable asynchrony.* Although simple to define, there are far-reaching consequences that must be considered. In a nutshell, composable asynchrony requires the following:

1. All operations requested by the application are *deferred.* Any request to perform an action returns immediately, with the action itself occurring asynchronously to the requestor's execution. For any operation that has an observable side effect, a handle or event must be returned that represents the eventual completion of that operation.

2. All operations are *deferrable.* As part of a request for any type of operation, the application may supply one or more handles as the *precondition.* The new operation does not begin until all operations whose handles appear in the precondition have completed. As with the operation

itself, this initial deferral is performed asynchronously to the requestor's execution.

The goal of composable asynchrony is that no application code should ever need to wait for something to occur, nor have to fret about what it could or should be doing while it is waiting. Instead, it can describe what it would do once it was done waiting and then either describe additional work to be performed or free up the execution resources so that other code can do the same. Computations involving complicated dependency graphs can be traversed in a modular, programmer-friendly order (e.g. a depth-first traversal), and opportunities for parallelism and latency-hiding are naturally exposed to the compiler or runtime.

4.1 From Tasks to Operations

Composable asynchrony clearly falls into the category of systems based on explicit dependencies, and the first obvious consequence of adopting composable asynchrony is that the task graph used by other systems must be augmented to include all other operations. This covers operations that perform data movement, but also those that are used to synchronize the actions of different processors in the system. Tasks may need to share a common resource, and small messages that coordinate mutual exclusion are impacted by the latency gap just as much as large data transfers are.

Perhaps surprisingly, composable asynchrony also demands the inclusion of other operations that one normally thinks of as being insensitive to latency concerns. For example, dynamic memory allocation (at least for local memories) requires no data movement and no coordination, and is generally thought of as executing instantly. Indeed, all existing systems perform such operations synchronously. However, once such operations become *deferrable*, that guarantee is lost. The instantaneous execution is preceded by an unknown delay for preconditions to be satisfied (e.g. all tasks using a dynamically allocated buffer must complete before the buffer is freed), and from the caller's perspective, the operation itself now has an unknown latency.

The move from a graph of just tasks to one that includes all operations often results in a significantly larger graph that must be handled by the system. Composable asynchrony can benefit from compile-time techniques that reduce the size of the graph. For example, dependencies can be safely eliminated if they are entailed by the transitive closure two or more others. Even with such optimizations, a system such as Realm that wishes to adapt to dynamic execution behavior must be designed to keep both storage and analysis overhead for these large graphs under control. Realm's novel *generational events* capture very large operation graphs in a compact and scalable form, and will be the subject of Chapter 6.

4.2 Heterogeneity and Hybrid Approaches

In a heterogeneous system, some operations will execute on one processor type, while others will execute on another processor type. More importantly, these operations are often initiated using different APIs. Systems that use a hybrid approach to handle intra-node operations differently from inter-node communication (e.g. OpenACC or StarPU with MPI) have a similar mix of APIs. The points where an application switches from issuing requests with one API to issuing requests with another are a common sources of stalls or bottlenecks. Composable asynchrony demands a way to capture dependencies that cross these API boundaries, effectively converting them to some form of portable dependency.

However, it also forbids the use of any mechanisms based on implicit dependencies provided by an API, even if it does not cross an API boundary. These mechanisms must be also be augmented to make dependencies explicit. Taken to an extreme, this might appear to call for explicit dependencies between every line of code or every subexpression, so some moderation is required. In particular, an implementation of composable asynchrony should only promote dependencies from implicit to explicit that it is capable of handling with minimal overhead and for which reordering of the operations in question is likely to yield performance benefits.

The result is that an implementation of composable asynchrony works from the top down, augmenting or replacing the coarse-grain connections between component-specific APIs. Although it is not required, this provides the opportunity to unify operations that occur in multiple APIs (e.g. task launches) behind a portable interface. Realm takes advantage of these to further reduce programmer effort significantly, as we will see in Chapter 5. However, more minimalist approaches are certainly possible, while a system with compiler support might attempt to provide a language for high performance computing in which asynchronous operations are composed at even finer granularity.

4.3 Deterministic Behavior

The introduction of asynchrony to a system opens the door to non-deterministic application behavior, but one must be careful to not open the door any wider than necessary. Chapter 2 argues that non-deterministic scheduling of operations is a highly desirable feature, compared to the alternative of manual enumeration of a deterministic scheduling for each target machine. However, nearly every HPC programmer expects the results produced by their application to be deterministic, and that is often not even sufficient.

HPC applications tend to consume significant fractions of the memory resources of a supercomputer, and a programmer will want the application's resource usage to be deterministic as well. Because resource management operations are incorporated into the operation graph, an approach based on composable asynchrony is able to limit the non-determinism in operation scheduling just enough to maintain resource usage guarantees.

4.4 Application Structure

A natural consequence of capturing all dependencies in the operation graph is the ability to use that graph as another description of the application structure. The graph captures both control and data dependencies, and can be used for much more than composable asynchrony. Realm explores two such opportunities.

First, a profiling framework that operates at the granularity of operations can allow targeted measurement of the costs of both execution and data movement. This can reduce the overhead of such profiling to the point where it is feasible to perform it during normal execution of an application, allowing dynamic optimization.

Second, an understanding of which operations depend on others allows a runtime such as Realm to assist with fault recovery in applications. Faults caused by hardware or software problems can be isolated to the operation in which they occur. Dependencies between the operations allow the runtime to limit the spread of the fault, often allowing other parts of the application to continue running while recovery is performed in affected area.

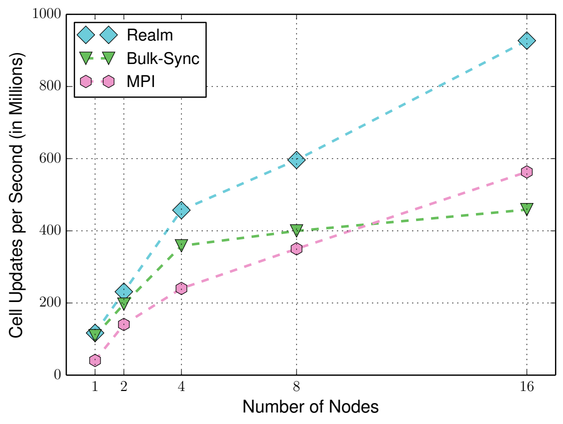
These two capabilities are described in Chapter 8, but the application structure exposed by the operation graph could be used for many other purposes. Either a compiler or a runtime could perform optimizations by transforming the graph or even use it to enable *elastic computing* by dynamically requesting and releasing execution resources to match the application's instantaneous needs.

4.5 Legion

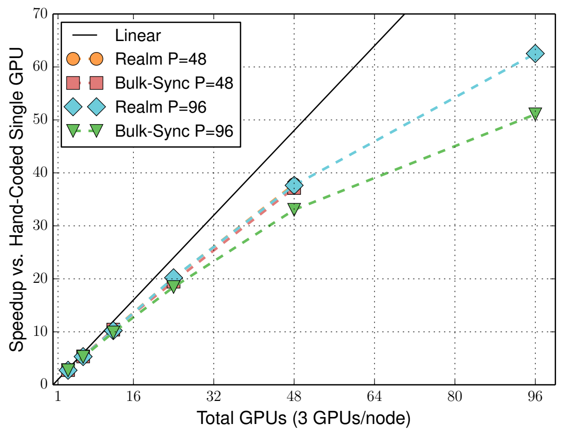
The insistence on fully composable asynchrony provides great benefits to programmability and portability. We will examine these benefits in a variety of ways in the chapters to come, but we offer the impatient reader a quick demonstration of the effect that composable asynchrony can have here using a few simple applications.

The applications we examine here are written in the Legion programming model[5], of which Realm is a principal component. As a result, the Legion runtime[7] is currently the most common "user" of Realm and its composable asynchrony. Applications in the Legion programming model are written either as C++ programs that directly use the Legion runtime API or as Regent programs, which are compiled into optimized machine code that links against the Legion and Realm libraries. The Regent compiler is able to eliminate much of the (necessary) verbosity of runtime APIs, allowing code to be compact and easily maintainable. It also implements a number of important optimizations that direct users of the C++ API are expected to perform manually[57].

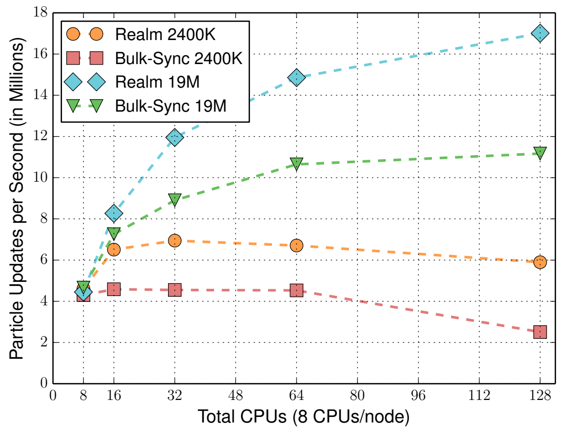
The Legion programming model presents a fairly standard hierarchical task model, but complements it with a novel hierarchical data model in which data resides in *logical regions*, which may be partitioned into subregions in multiple and arbitrary ways. This model allows an application to describe how both code and data can be decomposed into smaller pieces for efficient parallel



(a) AMR



(b) Circuit



(c) Fluid

Figure 4.1: Performance Comparison of Explicit vs. Implicit Dependencies

execution. The data model is expressive enough to handle both structured and unstructured data, and describe the data sharing patterns (e.g. the ghost cells from our heat diffusion example) that arise in HPC applications.

Performance portability is a key goal of the Legion programming model as well. Legion application code is machine-agnostic — there are no primitives to control mapping available to the application code itself. Instead, Legion defines a separate mapping process in which tasks are assigned to processors and instances of logical regions are assigned to memories. This is similar in spirit to the mapping performed by the Sequoia compiler, but the process in Legion is performed dynamically by an object that implements Legion's *mapping interface*, allowing the use of domain-specific knowledge and/or reaction to dynamic behavior of the input or the machine.

Although separated by a clean API boundary, Legion relies on Realm's feature set in a way that would be difficult to replace. Some of these dependencies are obvious — the Legion mapping interface exposes Realm's *machine model* directly and the runtime uses Realm's abstractions for processors and memories to remain agnostic to the decisions made by an application's mapper.

Others are less obvious. For example, parallelism in the Legion programming model is implicit. Legion applications have sequential execution semantics, and the Legion runtime uses information about the usage of logical regions by tasks to extract explicit dependencies between tasks.[1] The performance of the Legion programming model is therefore critically dependent on the ability of the Legion runtime to "run ahead" and expose enough parallelism to fill a large machine. Legion counts on Realm to make all operations asynchronous and avoid ever stalling the initiating task. To quantify that benefit, we re-examine the tests used to evaluate Legion in [5]. We first note that the scalability and performance relative to non-Legion reference versions for two of the applications is a good initial indication that the overhead introduced by Realm's dynamic analysis is not onerous. (Chapter 11 will show similar results for a full application and at much larger node counts.)

We then re-ran the Legion versions of the applications using a modified version of the Legion runtime in which the runtime waited on any dependencies of an operation rather than submitting them to Realm as explicit dependencies for the new operation. This models a runtime which is like Realm in all respects but one, replacing Realm's explicit dependency model with an implicit one similar to those used in bulk-synchronous programming models. The comparative results can be seen in Figure 4.1. In all cases, the original version using Realm's explicit dependencies outperforms the "bulk-sync" version with implicit dependencies, with the gap growing as node counts and therefore communication latencies increase. The adaptive mesh refinement mini-app (Figure 4.1a) scales fairly well with explicit dependencies, but the version with implicit dependencies begins to show scalability problems after 4 nodes, running at approximately half the speed of the full Realm version at 16 nodes. The circuit simulation (Figure 4.1b) didn't suffer as badly with implicit dependencies, but explicit

[1]This is similar to, but significantly more powerful, than the analyses performed by StarPU and OmpSs. In particular, Legion's support for *structure slicing* allows the (implicit) exposure of task-parallelism within an application while still allowing convenient access and cache-friendly memory layouts[6].

dependencies still account for approximately 20% of the performance at 16 nodes. Finally, the fluid (smooth particle hydrodynamics) simulation in Figure 4.1c was already showing signs of being limited by latency due to the relatively small problem size it operates on. Without the benefit of explicit dependencies, the larger problem loses a third of its performance and the smaller problem loses over half, resulting in negative scaling at 16 nodes.

Chapter 5

Realm

In this chapter, we introduce the basic Realm runtime API and provide some examples of its usage. The API (and runtime implementation) is written in C++, but some care has been taken to allow bindings to be created for other programming languages as well (e.g. C, Lua). Realm has been designed to be usable by not only human programmers, but by higher-level compilers and runtimes as well. Realm takes what we feel is an appropriately pragmatic approach for high performance computing applications, strongly discouraging "bad" behavior such as waiting directly on events or using pointers to global memory within a process, but neither prohibiting such actions nor hurting their performance unnecessarily.

Realm is designed for operation on large heterogeneous clusters, and provides a programming model which cleanly composes intra- and inter-node operations. Similarly, Realm uses an abstraction layer to simplify (and again render composable) the handling of heterogeneous resources. Realm provides a global and uniform functional model of the machine, allowing references to any execution resource or runtime object from anywhere, simplifying distribution and/or migration of work within the system. (No programming model can provide a uniform performance model.) The runtime itself is implemented in a distributed manner, with the operation graph itself distributed across the nodes in the cluster. Each node makes scheduling decisions for its subgraph and communicates with other nodes only as necessary. This eliminates any scaling bottlenecks that would result from centralizing the operation graph and/or its scheduling.

Realm handles scheduling of operations (i.e. the handling of preconditions and the selection of a ready operation to run next on an available resource), it takes the stand that the problem of efficiently computing good mappings (i.e. on which processors tasks should run and in which memories data should be placed) for arbitrary applications is simply not possible (at least not to the standards expected by the HPC community) with only local information. Rather than supplying a mediocre mapping algorithm that users must "trick" into making the desired choice, Realm provides no algorithm at all, giving the application total control over the process. The application is free to make its own choices, or delegate them to a library that provides good algorithms for applications within a particular domain, and Realm offers tools that can assist the application or library in making those decisions. These tools include a dynamic model of the machine on which the application runs (Chapter 7) and a profiling framework that allows real-time feedback on the quality of a given mapping decision (Chapter 8).

5.1 Realm Objects

The bulk of the Realm functionality is exposed through methods on objects of the following main classes:

- `Runtime`, a singleton object that represents the Realm runtime.
- `Machine`, a singleton object that exposes a *machine model* for the system on which the application is running. This is covered in detail in Chapter 7.
- `Processor`, a handle for an execution resource on which *tasks* can be run.
- `Memory`, a handle for a pool of memory in which application data can be stored.
- `RegionInstance`, a handle for a specific allocation within a `Memory`.
- `Event`, a handle that refers to the completion (or eventual completion) of some operation on which other operations may be dependent. Data movement is explicit in Realm, so all `Events` represent *control dependencies* between operations.
- `UserEvent`, `Barrier`, and `Reservation`, which permit more complicated synchronization patterns between operations.

Operations requested by the application (e.g. tasks, copies) are anonymous in Realm, but can be indirectly named by the `Event` that refers to their completion.

The Realm runtime is distributed across one or more separate processes, but presents a global view to the application. The handle to a Realm object may be used in any process, stored in arbitrary data structures, and passed as part of the arguments to a task. Realm uses the GASNet communication library to communicate between processes, utilizing both the *active message* (i.e. remote procedure call) and direct *memory-to-memory transfer* features. GASNet provides *conduits* for all of the high-performance interconnects used in today's supercomputers.

GASNet provides a `gasnetrun` script that assists with starting the necessary processes on each of the nodes in a system. Several GASNet conduits provide inter-operability with MPI, and the Realm runtime preserves this capability, allowing Realm applications to be started with more familiar job management tools such as `mpirun` and `aprun` (on Cray systems).

5.2 Basic Realm Application

Figure 5.1 shows the code for a very simple Realm application in its entirety. Although a typical Realm application is much more complicated, the initialization and shutdown code for most applications looks the same. Each of the main Realm classes is defined in its own C++ header file (e.g. `realm/processor.h`) but a convenient aggregation of them is available in `realm.h` and that is included on line 2. To avoid naming conflicts, all Realm definitions are placed within the `Realm` namespace. Most Realm applications will import them with a `using namespace Realm;` statement, but this example skips that shortcut for clarity.

Lines 4-9 define the implementation of the main (and only) task in this application. All Realm tasks use the same prototype, which is intentionally made more "C-like" to allow tasks to be implemented in languages other than C++. Each Realm task receives three arguments:

- `args` provides the arguments given by the caller at the time this task instance was *spawned*. The size of the arguments is provided in `arglen`. Due to both the needs of deferred execution and the distributed memory environment in which Realm operates, this is almost always a bitwise copy of the original arguments passed by the caller, so the arguments must not contain any pointers or any non-trivially-copyable C++ objects. (As mentioned above, Realm objects other than the `Machine` and `Runtime` objects are trivially-copyable, and this is one of the primary reasons for that. Realm includes some basic capabilities for packing and unpacking common C++ classes (e.g. strings, STL containers), but an application is free to use more powerful serialization libraries (e.g. Boost, cereal, or Protocol Buffers) as well.

- `userdata` is any arguments provided at the time the task was *registered* with the runtime. This is not used in our early examples, but will be covered in Chapter 9.

- The `Processor` on which the task is executing. This can be used to spawn additional tasks on the same execution resource or as part of queries of the machine model to find other processors or memories in the system.

All tasks in Realm have a `void` return type, as the spawner of a task has moved on. Most inter-task communication in Realm is done through `RegionInstance`s, but it is possible to create constructs similar to *futures* using `Barrier` objects (see Section 5.10).

The `main` function of a Realm application is the part that runs "outside" of the Realm runtime, and manages initialization and cleanup. In nearly all applications, this consists of the following steps:

1. Line 13 creates the Realm runtime. Although the `Runtime` is a singleton object, allowing the application to choose precisely when to create it allows Realm-based application code to be embedded with a larger host application. (This technique was used for the Legion port of S3D, the subject of Chapter 11.)

2. Line 16 initializes the Realm runtime, based on environment variables (often set by launching scripts such as `gasnetrun` or `mpirun`) and/or command-line arguments. This call must be performed by every process across which the Realm runtime is being distributed. Once it returns, the Realm runtime is running, but mostly idle. A small amount of background communication is performed to monitor system status, but all of the execution resources are dormant until tasks are spawned on them.

3. Line 20 *registers* our `hello_task` with Realm, allowing us to refer to it from anywhere in the application using the ID provided. This example uses a very simplified form of task registration in which the supplied function pointer is registered on every `Processor` in the current process, regardless of processor kind. (Due to address space layout randomization, different processes may not have identical pointers for this function, so every process does its own registration.)

4. Before the application can spawn our task, it must decide on which `Processor` it should be executed. Line 23 requests a reference to the `Machine` singleton and line 26 uses the simplest possible query to just get the first processor in the machine.

5. Line 31 *spawns* an instance of our `hello_task` on the chosen processor, providing the first command line argument (if present) as an argument to the task. As we are still “outside” the Realm runtime, `collective_spawn` method on the `Runtime` object is used to ensure that exactly one instance of the task is launched, regardless of the number of processes over which the runtime is distributed. (Had the normal `Processor::spawn` method, been used, a separate task instance would be spawned by each of the processes.) The `collective_spawn` method returns immediately, and the same `Event` handle is returned to every process.

6. This application is only running a single task, so line 37 tells the Realm runtime to initiate the shutdown of the runtime as soon as the event tracking the task completion has triggered. This call may be performed either by a Realm task or from the “outside”. It need not be called on all processes, but there is no harm in doing so.

7. Line 40 asks the original process to wait until the Realm runtime has completed its shutdown. Regardless of who initiated it, this call does need to occur in every process to ensure a clean shutdown.

The reader has probably noted by now that Realm’s “Hello World” is quite a bit longer than similar examples in most other languages. While such verbosity can be a hassle if a programmer is coding applications directly in Realm, recall that the expectation is that Realm is providing a lower-level performance portability layer for higher level runtimes and/or domain-specific libraries. For such customers, the simplicity of having a single way of doing things and the clarity of how all the constructs interact with each other outweighs the costs of a little more code generation.

```
#include <stdio.h>
#include <realm.h>

void hello_task(const void *args, size_t arglen,
                const void *userdata, size_t userlen,
                Realm::Processor p)
{
  printf("Hello from Realm! (%.*s)\n", arglen, (const char *)args);
}

int main(int argc, const char *argv[])
{
  Realm::Runtime runtime;

  // initialize Realm runtime
  runtime.init(&argc, &argv);

  // register hello_task, associate it with an ID we select
  const int HELLO_TASK_ID = 1;
  runtime.register_task(HELLO_TASK_ID, hello_task);

  // get the machine model singleton
  Realm::Machine machine = Realm::Machine::get_machine();

  // choose a processor to run our task
  Realm::Processor target_proc = Realm::Machine::ProcessorQuery(machine).first();

  // pass our first command line argument to the task, event 'e' will trigger
  //  when task is finished running
  const char *msg = (argc > 1) ? argv[1] : "no msg";
  Realm::Event e = runtime.collective_spawn(target_proc,
                                            HELLO_TASK_ID,
                                            msg,
                                            strlen(msg));

  // tell the runtime it can shut down as soon as that task finishes
  runtime.shutdown(e);

  // wait for the runtime to shutdown before terminating the process
  runtime.wait_for_shutdown();

  // succesful termination
  return 0;
}
```

Figure 5.1: Hello world in Realm

5.3 Data Model

Persistent application data is stored in `RegionInstances`. A `RegionInstance` is created in a particular `Memory` and cannot be moved. Instead, an application that wishes to migrate or replicate data will create more than one `RegionInstance` and issue *copy* operations between them. A copy may overwrite the destination or it may be a *reduction copy* that accumulates the source value into the destination value according to the chosen *reduction operator*. Realm also provides support for performing a *fill* operation that sets the contents of `RegionInstance` to a constant value.

A common problem that arises when multiple tasks share data is that they do not share all of it. One of the tasks might only be interested in some of the properties of elements (e.g. position but not velocity) or only in a subset of the elements (e.g. just cells on the boundary of a volume). Forcing every task to work with a single layout of the data can result in excess data movement and reduce the effectiveness of caches. Realm provides capabilities for accessing and copying subsets of an instance, but requires some understanding of the structure of the data to do so.

The model Realm uses treats each `RegionInstance` as a logical array of structures. The dimensions of the array are expressed by a `Domain`, which can describe either a multi-dimensional rectangle or possibly-sparse bitmask for unstructured data. The fields of the logical structure are not named, but are identified by their sizes and offsets within the structure. For example, the following code allocates a `RegionInstance` that holds 1000 elements, each with an `id` and `mass` field:

```
struct LogicalLayout {  // shown for clarity - not used by Realm
  int id;        // offset of 'id' is 0
  double mass;  // offset of 'mass' is 4 (packed layout)
};

Rect<1> bounds(0, 999);  // rectangle bounds are _inclusive_
Domain dom = Domain::from_rect<1>(bounds);
RegionInstance inst;
Event e1 = mem.create_instance(dom, { 4, 8 } /* field sizes */,
                               1, /* block size */
                               inst);
```

and the code to initialize just the `mass` field of all elements looks like:

```
double init_mass = 1.0;
CopySrcDstField mass_field(inst, 4 /* offset */, 8 /* size */);
Event e2 = dom.fill({ mass_field },
                    &init_mass, sizeof(init_mass),
                    e1 /* precondition */);
```

Note that the call to `create_instance` returns an `Event` that is used as a precondition to the `fill` method. The reason for this is discussed in Section 5.5.

The *block size* argument to the `create_instance` method controls the actual layout of the data in the newly-created instance. A value of 1 requests a standard *array of structures* (AOS) layout, in which all the fields for one element appear in memory before fields of the next element. A value of 1000 in this case would request a *structure of arrays* (SOA) layout, in which the `id` fields for all 1000 elements appear consecutively, followed by the `mass` fields for all 1000 elements. Intermediate values for the block size correspond to hybrid layouts in which smaller groups of elements are interleaved in memory. Such hybrid layouts are often beneficial for processors with wide vector datapaths.

5.3.1 Accessors

In addition to being able to copy and fill `RegionInstance`s, tasks must be able directly access individual elements and their fields. For direct access, there is a tension between the desire for portability and the need for performance. Ideally a task is written in a way that is concise and functionally correct regardless of what kind of memory an instance is in or how it is laid out. However, many tasks will want to iterate over some or all of the elements and it is desirable that accesses to the instances within these loops be as efficient as possible.

Realm addresses this with a `RegionAccessor` type that is templated both on the type of the field being accessed but also on an *accessor type* that defines what operations are possible and how expensive they are. A call to the `get_field_accessor` method of an instance initially returns a `Generic` accessor that provides the ability to `read`, `write`, or `reduce` any element for any memory type and layout combination, but does so at the cost of a method call per access. In contrast, the `Affine` accessor type (itself templated on the dimensionality of the array index) implements these methods as inlined array-address calculations and provides array-style reference notation, but supports only memories that are directly addressable (i.e. local to the node on which the task is executing) and instances that are laid out regularly (e.g. AOS or SOA, but not hybrid layouts).

The `Generic` accessor type supports a `convert` method that will produce an accessor of a different type if possible, as well as a `can_convert` method that should be used by code that intends to be portable. A portable version of the BLAS routine `SAXPY` might look like this:

```
void saxpy_task(const void *args, size_t arglen,
                const void *userdata, size_t userlen, Processor p)
{
  // unpack from arguments
  RegionInstance inst = ...;
  Rect<1> bounds = ...;
  float alpha = ...;
  RegionAccessor<Generic, float> ra_x = inst.get_field_accessor<float>(0 /* x offset */);
  RegionAccessor<Generic, float> ra_y = inst.get_field_accessor<float>(4 /* y offset */);

  if(ra_x.can_convert<Affine<1> >() && ra_y.can_convert<Affine<1> >()) {
    // optimized case
    RegionAccessor<Affine<1>, float> aa_x = ra_x.convert<Affine<1> >();
    RegionAccessor<Affine<1>, float> aa_y = ra_y.convert<Affine<1> >();
    for(int i = bounds.lo; i <= bounds.hi; i++)
```

```
      aa_y[i] += alpha * aa_x[i];
  } else {
    // portable case
    for(int i = bounds.lo; i <= bounds.hi; i++)
      ra_y.write(i, ra_y.read(i) + alpha * ra_x.read(i));
  }
}
```

Again, one can easily imagine some syntactic sugar that would only require the inner loop to be written once (ideally with the array-style references) and be automatically instantiated with the desired accessor type(s). The Realm philosophy is to let such magic be added by the application so that it can be done in precisely the way the application prefers.

5.3.2 Relaxed Data Models

A data model based on logical arrays of structures works well in most cases, but it can appear to be too restrictive for applications that work on data of varying types. A common approach for such instances is to describe them as a large array of single-byte elements, performing pointer arithmetic and casting on the application side to access the actual data. Although the base pointer will differ depending on where an instance is placed in memory, relative address from the base pointer are stable and Realm is still able to assist with copying or filling of subsets of the data.

5.4 Heterogeneous Processors

The examples so far have only shown tasks that run on a single core of the host CPU in the system. Realm refers to these as *host processors* and exposes them as one of the kinds of `Processor` in the machine. A `Processor` of this kind has a physical CPU core reserved for it, and runs only a single task at a time, guaranteeing the full resources (e.g. datapaths, caches) of that core are available to that task.

As we learned in Chapter 3, the bulk of the computational capability in many of today's supercomputers is provided by CUDA-capable GPUs. The CUDA programming model adds significant complexity to an application. The *kernels* that run on the GPU itself are written in CUDA, which is similar to C++ but uses many extensions that are not understood by standard C++ compilers. However, the code that launches these kernels must be run on the host CPU, and for systems with multiple GPUs, that code must maintain a separate *context* for each GPU and associate kernel launches with the correct one.

Numerous efforts have been made to unify the programming model for CPU and GPU code, and while some of them are promising, many application (and library) authors are either forced to write directly in CUDA or are more comfortable doing so. In keeping with the notion of separation of mechanism and policy, Realm exposes throughput-optimized cores as a separate processor kind for which the application must generate code. However, once the code has been generated, Realm's data movement and task spawning mechanisms allow the code that interacts with these tasks to be agnostic to the kind of processor that actually runs the tasks.

As an example, we will extend the SAXPY code from the previous section to be able to run on either the host CPU or a CUDA-capable GPU. We first need to write an CUDA implementation of the main computational task:

```
1  __global__ void saxpy_kernel(RegionAccessor<Affine<1>, float> ra_x,
2                               RegionAccessor<Affine<1>, float> ra_y,
3                               Rect<1> bounds, float alpha)
4  {
5    // standard CUDA idiom for determining which element this thread will compute
6    int idx = bounds.lo + (blockIdx.x * blockDim.x) + threadIdx.x;
7    if(idx > bounds.hi) return;
8
9    // now the simple computation
10   ra_y[i] += alpha * ra_x[i];
11 }
12
13 void cuda_saxpy_task(const void *args, size_t arglen,
14                      const void *userdata, size_t userlen, Processor p)
15 {
16   // unpack from arguments
17   RegionInstance inst = ...;
18   Rect<1> bounds = ...;
19   float alpha = ...;
20   RegionAccessor<Generic, float> ra_x = inst.get_field_accessor<float>(0 /* x offset */);
21   RegionAccessor<Generic, float> ra_y = inst.get_field_accessor<float>(4 /* y offset */);
22
23   assert(ra_x.can_convert<Affine<1> >() && ra_y.can_convert<Affine<1> >());
24
25   // determine kernel launch parameters
26   int threads = bounds.volume();
27   int threads_per_block = 256;
28   int blocks = (threads + (threads_per_block - 1)) / threads_per_block;
29
30   // launch kernel on GPU
31   saxpy_kernel<<< threads_per_block, blocks >>>(ra_x.convert<Affine<1> >(),
32                                                 ra_y.convert<Affine<1> >(),
33                                                 bounds, alpha);
34 }
```

This code is compiled with the CUDA compiler `nvcc`, which generates an object file containing both GPU executable code for the kernel (lines 1-11) and the host CPU code for launching it (lines 13-34). The host CPU code uses the same task signature and initialization code as our early CPU implementation, but there are a few differences to note.

This CUDA version of SAXPY does not include "fallback" code for dealing with memory kinds or instance layouts that are not compatible with the `Affine` accessor type. This is due in part to limitations of the CUDA programming model, but is primarily the result of programmer pragmatism. Why spend time writing a code path that will be so slow you never want to run it? This highlights an important point regarding Realm's performance portability goals. Realm is designed to assist a programmer in writing code that is as portable as they want, and discourages patterns that may inadvertently limit portability (e.g. implicit dependencies), but it does not *require* a programmer to write completely portable code.

Another key difference between our two implementations is that the CPU version was single-threaded, whereas this CUDA task is (massively) multi-threaded. There are two ways to look at this discrepancy. The pessimistic view is to see this as a limitation of the CUDA programming model. While individual CPU threads can be launched and associated with particular physical cores using the standard POSIX threads library, the CUDA driver APIs require that threads be started in large batches (or *grids*) and currently provide no ability to control which threads run on which physical processors within the GPU. Realm cannot expose mechanisms to control task launches with any finer granularity.

However, the optimistic view to see this as a reminder that both *task-parallelism* and *data-parallelism* are powerful tools, and Realm's ability to compose its task-parallelism with the data-parallelism of other libraries permits more efficient implementations that would be possible in a purely task-parallel approach. Indeed, libraries exist for fine-grained parallelism on traditional CPUs (OpenMP[27] is the most well-known but there are many others[32, 35]), and there are efforts in progress that will add new processor kinds that group together multiple CPU cores on the same node (and probably the same NUMA domain) and spread the fine-grained data-parallelism within an individual Realm task over these cores.

Another concern relates to the way in which the CUDA kernel is written. Although it is perfectly functional CUDA code, and will easily outperform code written for the host CPU (due to the huge disparity in available memory bandwidth and number of datapaths), it is far from optimal CUDA code. Significant improvements can be achieved by having CUDA threads handle multiple elements of the array and by matching the number of blocks in the kernel to the capacity of the exact GPU on which the kernel is executing. Choosing the right parameters and variations within the kernel requires expert knowledge and is the sort of thing that is best encapsulated in library code when possible. In fact, for dense linear algebra operations like SAXPY, the library (CUBLAS) is part of the standard CUDA toolkit. By using the standard CUDA programming model instead of trying to implement its own policies for code portability, Realm applications are able to use standard CUDA libraries and we can write our task in a way that is more concise and performs near-optimally on any CUDA-capable GPU:

```
#include <cublas.h>

void cuda_saxpy_task(const void *args, size_t arglen,
                     const void *userdata, size_t userlen, Processor p)
{
  // unpack from arguments
  RegionInstance inst = ...;
  Rect<1> bounds = ...;
  float alpha = ...;
  RegionAccessor<Generic, float> ra_x = inst.get_field_accessor<float>(0 /* x offset */);
  RegionAccessor<Generic, float> ra_y = inst.get_field_accessor<float>(4 /* y offset */);

  assert(ra_x.can_convert<Affine<1> >() && ra_y.can_convert<Affine<1> >());
  RegionAccessor<Affine<1>, float> aa_x = ra_x.convert<Affine<1> >();
  RegionAccessor<Affine<1>, float> aa_y = ra_y.convert<Affine<1> >();

  // cublasSaxpy wants base pointers and strides for x and y
  cublasSaxpy(bounds.volume(), alpha,
              &aa_x[bounds.lo], &aa_x[bounds.lo + 1] - &aa_x[bounds.lo],
              &aa_y[bounds.lo], &aa_y[bounds.lo + 1] - &aa_y[bounds.lo]);
}
```

A less obvious concern with the way Realm exposes support for CUDA has to do with CUDA's own deferred execution and support for task-parallelism and how they interact with Realm's. Like the spawning of a Realm task, a CUDA kernel launch call request returns to the caller immediately, with the launch command being placed in a queue of commands that are processed by the GPU. Commands are processed from a queue in the order they were inserted, allowing the application to launch additional kernels with the assurance that they will run after the previous kernel has finished. To allow independent kernels and/or data transfers to run concurrently, CUDA permits the creation of multiple command queues (called *streams*). Each stream still processes commands in FIFO (first-in first-out) order, but different streams can progress independently. A stream can include commands that wait on a certain amount of progress having been made in another channel, but this adds overhead and suffers the drawbacks of implicit dependencies. (Dependencies can be made explicit by putting every command in its own stream, but this increases overhead considerably.)

Realm's CUDA support makes use of streams to enable concurrent operations on the GPU, but handles dependencies between GPU operations using same `Event` handles as are used for all other dependencies between Realm operations. A relatively small number of CUDA channels are created (one for each "direction" of data movement and a small pool across which kernels are spread), and an operation is only inserted into a channel when it is *ready* (i.e. all dependencies have been satisfied). This preserves all the portability benefits of explicit dependencies, and simplifies Realm's internal handling of CUDA operations considerably.

The deferred nature of CUDA's execution model demands some caution with respect to the completion time of Realm tasks and copies that involve the GPU. When a task is spawned on a host CPU, that operation is considered to be complete as soon as control returns from the task function. However, for a task spawned on a throughput optimized core, the return from the task function is not sufficient. Any CUDA operations enqueued into channels must also finish before the task's completion event can be triggered. Realm handles this by inserting *fences* into the channel(s) used by a task, and performing periodic polling of these fences. Once all the necessary fences have been reached, the completion event is triggered and it is now safe for any dependent operations to commence.

This approach can add some scheduling latency when an operation is dependent on another operation in the same GPU, but this is not a major concern. One of the important side benefits of a runtime system designed for latency tolerance is that you can afford to increase latencies in some cases when there's a performance or efficiency benefit to be had in exchange.

Now that we have both host CPU and CUDA-capable GPU versions of our SAXPY task, we examine the features of Realm that allow calling code to use either interchangeably, yielding code that is naturally functionally portable across heterogeneous systems and provides the necessary mechanisms to implement policies designed to achieve performance portability. In our first Realm example, we saw the use of the `Runtime::register_task` method that associated a task ID with a single function pointer on every processor. Now that we have different implementations of the task for different processor kinds, we require more control and use the `Processor::register_task_by_kind` method instead:

```
const int SAXPY_TASK_ID = 2;
Processor::register_task_by_kind(Processor::HOST_PROC, false /* local node */,
                                 SAXPY_TASK_ID, CodeDescriptor(saxpy_task));
Processor::register_task_by_kind(Processor::CUDA_PROC, false /* local node */,
                                 SAXPY_TASK_ID, CodeDescriptor(cuda_saxpy_task));
```

The meaning of the second (boolean) parameter and the purpose of the `CodeDescriptor` wrapper around the function pointers are discussed in Chapter 9.

When application code calls `spawn(SAXPY_TASK_ID, ...)` on any Realm `Processor` object, Realm will automatically take care of running the correct version of the task, with the correct execution resources available. However, this only handles part of the portability problem. Different kinds of processors use different memories in the system and the data used by a task must reside in memories that are acceptable. Thus, a truly portable implementation of SAXPY will need to be able to move data around as well. A full example of this can be seen in Figure 5.2 and we cover the key points here:

- Lines 1-2 define the interface for our portable SAXPY launcher. In addition to the arguments needed for the SAXPY operation itself (the data, the vector length, and the scaling factor `alpha`), the launcher function accepts a precondition `Event` as input and produces a (usually different) `Event` handle as output. This pattern is the key to Realm's *composable asynchrony.*

- Line 5 uses the `RegionInstance::get_location` method to determine which `Memory` contains the input/output data. An application that wishes to be portable across different systems should avoid making assumptions about where data resides or where other operations have or will be executed, and instead dynamically inquire about locations and make mapping decisions accordingly. It is important that the overhead of these queries is minimized, and Realm uses an encoding for its handles that allow the handle of the containing `Memory` to be derived directly from the handle of the `RegionInstance`.[1]

- Armed with the knowledge of the current location of the data, line 10 calls a helper function that will return the `target_proc` on which the SAXPY task should run and the `target_mem` in which that processor should access the data. This decision can be made in any number of ways, and we will examine some of them in Chapters 7 and 8. In this code however, we will treat `select_saxpy_target` as a black box. This is in the spirit of clearly separating policy and mechanism, and maximizes the portability of this code. If the interface permits no assumptions about the choices made in `select_saxpy_target` (beyond the requirement that the chosen processor and memory are compatible), it is likely that the code will work for any such choice.

- Lines 17-25 show a second example of reacting to dynamic behavior. If the policy decision in line 10 prefers to operate on data in place, we can give the initial `RegionInstance` directly to the SAXPY task. If not, the code creates a temporary instance in the desired memory and requests a copy from the original instance. Note the "chaining" of the `Events` through these operations.

- Line 32 spawns the SAXPY task on the target `Processor`. The task becomes the next link in the event chain, depending on the completion of the copy that populated the temporary instance if it exists, and on the original precondition if not. The ability of a single `Event` to transitively describe an arbitrary collection of prior operations further improves composability. The spawn of the SAXPY task need not care how its input data was produced.

- If the SAXPY task operated on a temporary instance, lines 35-43 take care of copying the result back and destroying the instance. This maintains our original semantics of appearing to operate on the `RegionInstance` that was passed in to this function. However, one can imagine optimizations that delay the instance destruction to allow the possibility of reuse, or defer the copy back until the preferred location of the consumer of the SAXPY result is known. The Legion runtime system does both of these, as well as several others.

[1]An alternative that eliminates this overhead is to do the location analysis and mapping statically, as was done in Sequoia[33]. However, compile-time approaches are ill-suited to handling data-dependent behavior or dynamic execution environments and load-balancing.

- Finally, line 46 returns the end of our `Event` chain to the caller. Due to Realm's deferred execution, it will often be the case that some of the operation(s) requested in this function will not have finished (or perhaps even started), but our return `Event` names that point in the future when they have finished, allowing the caller to compose the operations requested here with its own.

In total, this example provides a good overview of Realm's approach to performance portability. The `Processor`, `Memory` and `RegionInstance` abstractions allow system-agnostic application code to easily describe task execution and data movement throughout a distributed memory system, ignoring the specific APIs used for each kind of processor memory under the covers. The `Event` abstraction allows the application code to concisely capture explicit dependencies between operations, allowing Realm to use deferred execution to hide the unavoidable and often unpredictable latencies of computation and communication. However, the abstractions stop there. Realm does not automatically produce code for different processor kinds, leaving that task to the programmer or perhaps compilers and libraries tailored for a given domain. Similarly, Realm does not attempt to make mapping policy decisions such as `select_saxpy_target` above. It provides some useful tools for making such a decision, but again demands that the decisions be made by the programmer (or some delegate).

5.5 Deferred Resource Management

One area in which all existing asynchronous runtimes fall short of fully composable asynchrony is in dynamic memory management. For example, even though all CUDA kernel launches and memory transfers are deferred, the `cudaMalloc` and `cudaFree` calls are fully synchronous. A call to either will automatically wait for all previously requested operations to complete before performing the allocation or deallocation. A request to deallocate memory clearly must not be executed until all operations using that memory have finished, but a synchronous `cudaFree` is even worse than an implicit dependency — putting independent work between the last kernel and the memory deallocation just forces the deallocation to stall until that independent work is done too!

The case for deferred and deferrable memory allocation is a little less obvious. It is always safe to execute an allocation request immediately, as there can be no earlier operations that use the memory resource that is about to be allocated. However, when the task that is issuing the requests has gotten far ahead of actual execution, the immediate execution of allocation requests can greatly increase the lifetime of the allocation, wasting memory resources. By deferring an allocation request until the first user of that allocation would otherwise be able to run, the lifetime of the allocation is minimized.

Once an allocation request becomes deferrable via a precondition, it clearly must return an `Event` for its completion. However, the benefit of making allocation deferred exists even when preconditions

```
Event portable_saxpy(Domain bounds, RegionInstance vector_data,
                     float alpha, Event precondition)
{
  // determine what memory the data is currently in
  Memory initial_mem = vector_data.get_location();

  // decide where to run saxpy and where data should be placed
  Processor target_proc;
  Memory target_memory;
  select_saxpy_target(initial_mem, target_proc, target_memory);

  // a common Realm pattern keeps an Event variable that tracks the
  //  last operation we've requested - start with the input precondition
  Event e = precondition;

  // if target memory does not match the initial_memory, create a temp instance and issue a copy
  RegionInstance target_inst = vector_data;
  if(target_mem != initial_mem) {
    e = target_mem.create_instance(bounds, { 4, 4 }, target_inst, e);

    // copy both x and y fields
    e = dom.copy({ CopySrcDstField(vector_data, 0, 4), CopySrcDstField(vector_data, 4, 4) },
                 { CopySrcDstField(target_inst, 0, 4), CopySrcDstField(target_inst, 4, 4) },
                 e);
  }

  // arguments to the saxpy task will include bounds, target_inst, and alpha
  void *args = ...;
  size_t arglen = ...;

  // spawn the task, making sure to use the right precondition
  e = target_proc.spawn(SAXPY_TASK_ID, args, arglen, e);

  // if we used a temporary instance, copy data back and destroy the instance
  if(target_mem != initial_mem) {
    // only need to copy back the y field (x is unchanged)
    e = dom.copy({ CopySrcDstField(target_inst, 4, 4) },
                 { CopySrcDstField(vector_data, 4, 4) },
                 e);

    // instance can be destroyed as soon as copy is complete
    target_mem.destroy_instance(target_inst, e);
  }

  // all done - return our finish event to the caller
  return e;
}
```

Figure 5.2: Portable saxpy in Realm

are not used. As Realm operates in a distributed memory environment, interactions with shared resources (such as the internal data structures tracking allocations within a `Memory`) will often involve inter-node communication and at least thousands of cycles of latency.

Although the actual allocation of memory is deferred in Realm, one part of `create_instance` that must still be performed immediately is the assignment of a `RegionInstance` handle for the new allocation. The caller requires this handle so that it may pass it to subsequent fills, copies, and/or tasks. The encoding used by Realm for these handles divides the space of handles between nodes so that any node can create a unique `RegionInstance` handle for any `Memory` without requiring any inter-node communication.

5.5.1 Resource Exhaustion

The choice to defer memory allocation raises two issues having to do with resource exhaustion. The first is the question of how to report an allocation attempt that has failed due to insufficient remaining resources. This will be covered as part of the discussion of fault tolerance in Chapter 8, but the key thing to keep in mind is that, for most HPC applications, memory exhaustion is treated as a fatal error. The standard "recovery method" is to allow the job to fail and then restart it with a few more nodes so that the working set on each node is reduced. Users generally accept this, but probably do so only due to an implicit assumption that the application's memory usage is deterministic. That is, if a job of a given size has completed within available memory limits on one run, it will do so on every future run as well.

However, a simple implementation of deferred memory allocation can easily result in non-deterministic behavior in which allocations may succeed or fail based on the unpredictable order in which their preconditions are satisfied with respect to deallocation requests. Effectively, a side-effect of Realm's automatic extraction of parallelism is that it can increase the application's working set dramatically compared to the programmer's expectations.

A simple example of code that is vulnerable to this issue can be seen in Figure 5.3. Each iteration of the loop creates and then destroys a temporary instance, and while the programmer might intend that the parallelism should be limited by the available memory resources, there's nothing to enforce that.

Realm makes this case work, and restores determinism to the memory allocation problem in general, by keeping a list of pending memory allocations in the order they were requested (i.e. ignoring their preconditions) as well as the set of pending memory deallocations.[2] To decide whether a new allocation request can be satisfied, Realm computes a *projected allocation state* of the memory by taking the current allocation state, performing all the pending deallocations and then all the pending (and already accepted) allocations. If the new request can be satisfied in this projected state, the allocation is accepted and added to the list. (Actual completion of the allocation is still

[2]This set may include some of the instances whose allocations are still pending!

```
Event parallel_alloc(int num_child_tasks, Event precondition)
{
  std::set<Event> child_events;

  // each iteration depends on the precondition but not on each other
  for(int i = 0; i < num_child_tasks; i++) {
    // temporary instance created for each loop iteration
    RegionInstance inst;
    Event e1 = mem.create_instance(..., inst, precondition);

    Event e2 = proc.spawn(..., e1);

    mem.destroy_instance(inst, e2);

    child_events.add(e2);
  }

  // our work is not done until all children have finished
  return Event::merge_events(child_events);
}
```

Figure 5.3: Massively-parallel memory allocation in Realm

deferred until its precondition has been satisfied and any necessary deallocations have completed.)

If the new request cannot be satisfied in the projected state, the allocation is considered to have failed, even if it turns out that additional deallocation requests are made before its precondition is satisfied. Such a fortuitous circumstance cannot be guaranteed to happen in all cases, and the desire for determinism requires conservative behavior. Importantly, this algorithm guarantees that as long as allocations and deallocations performed on a given `Memory` are race-free, a Realm application can reliably run using space no larger than would be required for a single-threaded execution, with larger memory capacity simply increasing the number of legal parallel execution orders.

Some recent work solves a similar problem of bounded memory scheduling on execution graphs extracted by an inspector/executor framework[54]. It seems likely that Realm's operation graph provides sufficient information to apply the same techniques, further improving the parallelism available when an application is operating near the memory capacity of the system.

5.6 Asynchronous Synchronization

Normal Realm `Events` are excellent for capturing dependencies between producers and consumers, but there are other forms of synchronization or coordination that are commonly used in parallel and distributed applications. We will examine what it means to perform synchronization in an asynchronous programming model, and hopefully the reader is convinced by now that whatever operations are provided for this purpose must be composable with all other runtime operations.

The most common synchronization pattern is the notion of a *critical section*, a hopefully-short sequence of code that may be executed by at most one thread at a time. Critical sections are often used to perform updates to shared data structures in a way that guarantees no reader observes the data structure in the middle of an update. In other programming models, critical sections are implemented using *mutexes* (commonly also called *locks*). Before entering a critical section, a thread must *acquire* the corresponding mutex and hold that mutex for the duration of the critical section. Upon exit, it *releases* the mutex for another thread to use.

If a second thread attempts to acquire the mutex while the first thread still owns it, the call *blocks*, suspending the second thread until the acquisition can safely occur (i.e. after the first thread has released the mutex). A blocked thread makes no forward progress, preventing it from entering the critical section but also preventing it from performing any other independent operations that happen to occur after it in program order. Many mutex implementations also include a way to perform a *non-blocking* acquisition attempt that is guaranteed to return immediately, but does so without acquiring the lock if it is held by another thread. This can allow the calling thread to attempt its own scheduling, the limitations of which were discussed in Chapter 2.

The second failure of composability relates to the inability to defer the acquisition. It makes no sense to acquire a mutex before any other preconditions of the critical section are satisfied, but the only way to achieve this with a standard mutex is to have the caller explicitly poll or wait before attempting the acquisition.

In addition to the issues with respect to composable asynchrony, standard mutexes have another feature that makes them hard to use with task-based runtimes such as Realm. Existing systems associate a held mutex with a particular thread and generally require that the acquiring thread be the one that performs the release. When a critical section involves the execution of more than one task, the final task in the section may be executing in a different thread (or even on a different node) than the first task.

5.6.1 Reservations

To enable this common critical section pattern in a way that both preserves composable asynchrony and meshes better with a task-based runtime, Realm provides the `Reservation`. Reservations are created by the Realm application on demand, and like most other Realm objects, a `Reservation` is described by a portable handle that may be packed into arbitrary data structures and used by any task on any node in the system. The `Reservation` interface looks very similar to that of a standard mutex at first glance, but the semantics are quite different.

The `Reservation::acquire` method is similar to non-blocking acquisition of a normal mutex in that it always returns immediately, but the acquisition of a `Reservation` always succeeds. The only question is *when* that success occurs, and the `Event` returned by the method captures that. The caller uses that event as a precondition for the critical operation(s), guaranteeing their execution will experience the desired mutual exclusion without any further involvement from the caller.

Every call to `Reservation::acquire` requires a matching call to `Reservation::release`. If the exact set of critical operations is known to the caller, it can request the release immediately after issuing those operations, using their completion event(s) as a precondition to the release. However, if any of the operations might launch additional child operations, their execution must also be contained in the critical section and an alternative technique is required. Instead of requesting the release directly, the caller can delegate it to the last operation in the critical section. That operation either performs the release at the end of its execution, or further delegates it to the last of any children operations it has launched, allowing arbitrarily-deep trees of tasks.

Figure 5.4 shows a common usage pattern for a `Reservation`. A parent task launches a large number of child tasks that perform some expensive analysis (e.g. looking for particular features in images) and the results of each analysis are summarized in a shared data structure. If every analysis were guaranteed to take exactly the same amount of time, the updates of the shared structure could be performed in sequential order without loss of efficiency. However, if the analysis has unpredictable execution time, this total ordering over-constrains the scheduling and increases total runtime.

In addition to the basic mutex, many systems offer a *reader-writer lock* that can improve performance by allowing multiple readers simultaneous access to a shared resource, but still guaranteeing exclusive access to writers. Realm's `Reservation` supports this directly through a parameter passed to the `acquire` method. As with the exclusive access mode, each call to `acquire` in shared mode must be matched by a call to `release`. Finally, Realm's reservation matches most other mutex implementations in that it is *non-reentrant* — a second attempt to acquire a `Reservation` by the current holder results in a deadlock. Some systems do offer a *re-entrant lock*, which allows a single thread to perform additional nested acquisitions, but this adds overhead for a very uncommon case, and relies on the association of held mutexes with particular threads.

5.6.2 Distributed Reservations and Fairness

As with all other resources, a `Reservation` may be used by any task in the system. When requestors on multiple nodes contend for the same `Reservation`, Realm keeps track of which process is the current *owner* of the `Reservation`, and sends messages over the network to request, and ultimately execute, transfers of ownership. Although the latency incurred by these messages can be hidden, the average throughput of a single `Reservation` (i.e. how many requests can be satisfied per unit time) is vulnerable to the latency gap, since requests cannot be granted during the actual migration of a `Reservation`.

Realm is therefore designed to minimize the number of times the `Reservation` is migrated between nodes in the system. It does so by favoring local requests from the same node above all requests for migration from other nodes. Migration requests are held until the list of local waiters is

```
Event parallel_analysis(std::vector<RegionInstance> inputs,
                        RegionInstance summary,
                        Event precondition)
{
  // create a Reservation to ensure mutual exclusion of summary updates
  Reservation rsrv = Reservation::create_reservation();

  std::set<Event> child_events;

  // each iteration depends on the precondition but not on each other
  for(RegionInstance i : inputs) {
    // again treating mapping policy decisions as a separate concern
    Processor p = select_target_processor(i);

    // launch expensive analysis task
    Event e1 = p.spawn(ANALYSIS_TASK_ID, ..., precondition);

    // acquire reservation on behalf of summary task
    Event e2 = rsrv.acquire(true /* excl */, e1);

    // summary task launched with successful acquisition as precondition
    Event e3 = p.spawn(SUMMARY_TASK_ID, ..., e2);

    // also release reservation on behalf of summary task
    rsrv.release(e3);

    child_events.add(e3);
  }

  // create a merged event that captures when all summary tasks are complete
  Event all_done = Event::merge_events(child_events);

  // return it to the caller, but also use it to clean up our Reservation safely
  rsrv.destroy(all_done);

  return all_done;
}
```

Figure 5.4: Use of `Reservation` for mutual exclusion

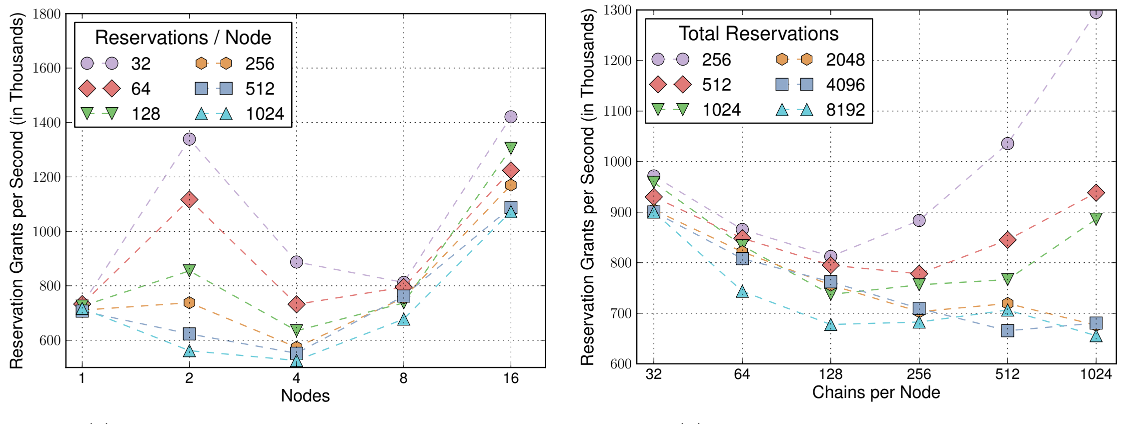
empty, but no longer than that. This guarantees forward progress and that `Reservations` migrate in a timely fashion when contention is low, but significantly improves throughput (and as a direct result, average wait times) when contention on a `Reservation` is high.

Apart from the strict prioritization of local requests over remote ones, arbitration between contending requests is fair. Within a node, requests for a `Reservation` are granted in the order. Between nodes, a round-robin arbiter guarantees that all other nodes with pending requests will be able to satisfy them before the `Reservation` can return to a node. Although any unfairness can lead to starvation in an adversarial environment, Realm's `Reservation` system does not include any avoidance policies. There is no "best" policy, and some are quite complicated. As with many other cases, Realm instead provides mechanisms with predictable behavior in all cases, allowing application or library code to implement custom policies on top of them.

To quantify the benefit of this approach, we use a microbenchmark that measures the rate at which requests can be satisfied by a collection of `Reservation`s at varying levels of contention. A parameterized number of `Reservation`s are created per node and their handles are made available to every node. A task on each node then creates a parameterized number of *chains* of acquire/release request pairs. Each request in the chain attempts to acquire a random `Reservation`, but uses the previous acquisition in that chain as a precondition. Thus the total number of chains across all nodes gives the total number of acquire requests that can exist in the system at any given time. All chains are started at the same time and the time to process all chains is divided into the total number of acquire requests to yield an average grant rate.

Figure 5.5a shows the `Reservation` grant rate for a variety of node counts and reservations per node. The number of chains per node is varied so that the total number of chains in the system is 1024 in all cases. For the single-node cases, the insensitivity to the number of `Reservation`s indicates that the bottleneck is in the computational ability of the node to process the requests. For larger numbers of nodes and larger numbers of `Reservation`s per node, contention for any given `Reservation` is very low, and nearly every acquisition will require the a migration. The speedup with increasing node count suggests the limiting factor is the rate at which `Reservation`-related messages can be sent over the network. However, in nearly all cases, the performance actually increases with decreasing number of reservations. Although contention increases, favoring local reservation requestors makes that contention an advantage, reducing the number of network messages that must be sent per `Reservation` grant.

The benefit of inter-node unfairness is more clearly shown in Figure 5.5b. Here the node count is fixed at 8 and `Reservation` grant rates are shown for a variety of total reservation counts and number of chains per node. At 32 chains per node (256 chains total), contention is low and the grant rate is high. As the number of chains per node increases there is more contention for reservations and the grant rate drops. For smaller reservation counts, further increases in the number of chains result in improved grant rates. On each line, the grant rates begin to improve when the chains per node exceeds the total number of `Reservations`, which is where the expected number of requests per `Reservation` on any given node exceeds one. As soon as there are multiple requestors for the same `Reservation` on a node, they will all be satisfied by a single migration of the `Reservation`, improving overall throughput. This increase only occurs with Realm's unfair migration algorithm — a a more fair approach would still require a migration for nearly every request.



(a) Reservations for Fixed 1024 Chains (b) Reservations for Fixed 8 Nodes

Figure 5.5: Microbenchmark Results.

5.7 Fork-Join Parallelism

The examples used so far have all used a flat task structure in which one parent task launches all of the other operations performed by the application. While convenient for didactic purposes, both scalability and modularity concerns demand that a real application be written hierarchically, with child tasks launching grandchild tasks and so on. This raises the important question of whether the a task's "lifetime" includes the lifetime of all child operations it launches. A programming model in which this containment is guaranteed is commonly called a *fork-join model*. The fork-join model provides an intuitive model of execution that contains effects within subtrees of the call hierarchy, which allows a programmer to reason about the functional behavior of their program as if it were executed sequentially. However, such simplicity comes at the cost of unnecessary synchronization, and amounts to a return to bulk-synchronous programming when the added task hierarchy is being used to address scalability concerns. One of the most well-known examples of strict fork-join parallelism is the Cilk programming language[11].

The alternative extreme, in which a parent task's lifetime is completely decoupled from any child operation(s) it launches, is often called a *fire-and-forget model*. This reverses the pros and cons of the fork-join model. All operations inhabit a single dependency graph and the dependency edges that exist in the graph are those that are explicitly added by the application. However, the intuitive model of scoped execution is lost. The Open Community Runtime[52] falls at this end of the spectrum.

However, there is an asymmetry here that Realm takes advantage of. As long as a way is provided to add dependency edges from child operations back to their parent, an application can start from the fire-and-forget style and opt back in to the fork-join model incrementally. (Legion presents it as starting from scoped execution and *relaxing* dependencies, but that is simply a matter of perspective.) A brute-force way to achieve this is to allow a parent task to wait on an `Event`. Realm allows this, but discourages it, as it has the usual drawbacks associated with implicit dependencies — Realm cannot see what independent work may also be held up by the wait. The `Event::include` method instead adds an explicit dependency between the specified event and the “completion” of the calling task. The calling task can return before the included event has occurred, freeing up the execution resources, but the lifetime of the task will be extended until all included operations have completed as well. Chapter 3 described a similar issue related to the deferred execution of CUDA kernels launched by a task — these use the same mechanism in Realm’s implementation.

Figure 5.6 shows a version of our SAXPY kernel that uses fork-join parallelism to improve performance for very large inputs. If the vector length is larger than some (likely processor-specific!) threshold, it is preferable to split the one task into two or more, allowing the work to be performed concurrently on multiple processors. Recursive bisection is used, with the right branch being forked off into a subtask while the original task follows the left branch. By calling `Event::include` on the completion event for the right branches, the original task’s completion event will correctly encompass all of the subtasks. A similar effect could have been achieved by launching $\frac{N}{K}$ tasks up front, where N is the size of the vector and K is the splitting threshold, but the recursive bisection approach used here is preferable for two reasons. First, it improves modularity. The logic for exposing parallelism is contained in the task’s implementation rather than appearing at the call site, and the caller is able to use a single `Event` to describe dependencies on the computation. Second, it avoids exposing “too much” parallelism, which can increase overhead and memory consumption without improving performance. If $\frac{N}{K} \gg P$, where P is the number of processors in the system, the recursive bisection approach limits the number of tasks being managed by the runtime to $P \log_2(\frac{N}{K})$, while still guaranteeing enough work for every processor.

5.8 External Dependencies

Another synchronization pattern that is often useful is the ability declare one or more operations (possibly with their own inter-dependencies) that should be performed when some arbitrary condition is satisfied in the future. If it is known which operation will satisfy the condition, that operation can simply be used as the precondition, but if the identity of that operation is unknown at the time (due perhaps to dynamic application behavior or simply encapsulation for modularity), Realm provides

```
void nested_saxpy_task(const void *args, size_t arglen,
                       const void *userdata, size_t userlen, Processor p)
{
  // unpack from arguments
  RegionInstance inst = ...;
  Rect<1> bounds = ...;

  // recursive bisection if the data is large enough
  while(bounds.volume() > SPLIT_THRESHOLD) {
    int mid = (bounds.lo + bounds.hi) / 2;
    Rect<1> child_bounds = bounds;
    bounds.hi = mid;
    child_bounds.lo = mid + 1;

    // choose where to run child task and spawn it
    Processor child_proc = select_processor(inst);
    Event child_done = child_proc.spawn(...);

    // now include the child's completion in our own
    child_done.include();
  }

  // create accessors, etc...

  // actual computation on remaining bounds
  for(int i = bounds.lo; i <= bounds.hi; i++)
    ra_y.write(i, ra_y.read(i) + alpha * ra_x.read(i));
}
```

Figure 5.6: Implementation of SAXPY kernel using hierarchical decomposition

the notion of a “dependency to be named later” in the form of a `UserEvent`. A `UserEvent` is created explicitly by the application, but once created, may be used as any other `Event` (it is in fact defined as a subclass). The `UserEvent::trigger` method is called by whatever task in the system determines that the condition associated with the event has been met. Like all other operations in Realm, this triggering of a `UserEvent` is of course deferrable. The `UserEvent` provides a kind of callback mechanism for Realm applications. However, unlike most callback systems in which the initiator directly executes the callback operation (often causing portability hassles when the initiating operation can run on multiple processor kinds), Realm's mediation of callbacks through the event graph allows the operation(s) involved in the callback to be arbitrarily complicated and executed on a part (or parts) of the machine of the application's choosing.

5.9 Barriers

A final synchronization pattern that is very common in large HPC applications is the need to have two or more long-running tasks coordinate subtasks (or copies) related to communication (e.g. a ghost cell exchange) without having to synchronize the entire parent tasks. A task will often be coordinating with multiple subsets of other tasks and the iterative nature of HPC applications results in long succession of these coordinations (often called *rendezvous*) over the lifetime of the task. If these coordinated tasks could exchange the `Event` handles for the appropriate subtasks, Realm's distributed implementation would allow them to directly include the handles as preconditions for their own subtasks, but this creates a new data exchange problem to solve the original data exchange problem. Instead, the peers involved in a given coordination need to have pre-arranged one or more `Event`-like things. The `UserEvent` construct described above is suitable when the coordination is a one-time occurrence, but each participant in a coordination would require a separate `UserEvent` to signal its readiness to the others. Additionally, a `UserEvent` cannot be reused once triggered, so an iterative process would require a unique `UserEvent` for each iteration (again, for each participant). Realm improves the plight of such applications by providing a `Barrier` abstraction, which efficiently describes an unbounded sequence of rendezvous. The completion of each rendezvous *phase* in the sequence requires a specified number of *arrivals* to that phase as well as the completion of all previous phases. Unlike barrier constructs in bulk-synchronous programming models, Realm's `Barrier` is fully composable. Although the `Barrier` handle implicitly refers to the sequence of rendezvous phases, it explicitly refers to a particular phase, and like a `UserEvent`, may be used as a precondition for any other Realm operation. As expected, the `arrive` method performs an arrival on the specified phase, but only once the supplied precondition has been satisfied. As a task completes its involvement with one phase and moves on to the next, it uses the `Barrier::advance` method to obtain a handle that explicitly refers to that next phase.

Figure 5.7 shows how a Realm application might use a `Barrier` to coordinate ghost cell exchange while performing the interior/boundary split optimization described in Chapter 2. This is a simplified version that uses a single barrier across all tasks rather than the generally preferred pair-wise synchronization, but it would be straight-forward to switch to the more optimal version. With a separate `Barrier` per communication partner, the barrier arrival and advance operations (lines 23 and 30, respectively) would be applied to each of the barriers, and the `merge_events` call on line 27 would similarly be modified to include all of the barriers as preconditions. Note also that the code is ordered differently than in Chapter 2. The arrival at a phase (line 23) appears to immediately precede the dependence on it (lines 26-27). In a system based on implicit dependencies, this arrival would have to be moved above the independent work on lines 19-20 to hide the latency involved in the synchronization. However, Realm's use of explicit dependencies allows all the synchronization code to be grouped together for readability (and, in a larger code base, modularity), as the deferred execution of the computation in lines 19-20 does not prevent the runtime from seeing the

```
void ghost_exchange_task(const void *args, size_t arglen,
                         const void *userdata, size_t userlen,
                         Realm::Processor p)
{
  // unpack Barrier object we will use for coordination
  Barrier barrier = ...;

  // as always, decide which processor we're going to run our work on
  Processor p = select_target_processor(...);

  // on each iteration, we'll perform updates of our interior data
  //  and our boundary data - keep an Event that captures the readiness
  //  of the most recent version of each
  Event prev_interior = Event::NO_EVENT;
  Event prev_boundary = Event::NO_EVENT;

  for(int i = 0; i < NUM_ITERATIONS; i++) {
    // interior update requires only our own data
    Event next_interior = p.spawn(UPDATE_INTERIOR_TASK_ID, ...,
                                  Event::merge_events(prev_interior, prev_boundary));

    // our current boundary data contributes to this phase of the rendezvous
    barrier.arrive(prev_boundary);

    // boundary update requires completed rendezvous
    Event next_boundary = p.spawn(UPDATE_BOUNDARY_TASK_ID, ...,
                                  Event::merge_events(prev_interior, barrier));

    // advance our barrier to refer to the next phase
    barrier = barrier.advance();

    // next->prev
    prev_interior = next_interior;
    prev_boundary = next_boundary;
  }
}
```

Figure 5.7: Coordinating ghost cell exchange with a `Barrier`

independent barrier arrival on line 23.

5.10 Futures

Although the main focus in high performance computing is in performing efficient computations on large collections of data, there are often small pieces of data that must be communicated as well. These small pieces of data tend to be accompanied by synchronization operations. The most obvious example of this is the result of a non-void function (e.g. computing the scalar dot product of two large vectors), in which a dependent operation needs to wait for completion of the dot product, but likely also requires the value. Similarly, ghost cell exchanges like the one above may be of variable size, and the producer needs to inform the consumer(s) of that size on each iteration.

A construct that captures both a value and the synchronization necessary to wait for it to be available is commonly called a *future*. Although these pieces of data could be managed and moved in `RegionInstance`s in Realm, the overhead (both programmer and runtime) would be unacceptable. Instead, Realm extends the functionality of the `Barrier` to efficiently carry small pieces of data as part of the messages used for synchronization. When arriving at a barrier, a value is provided by the caller, which can be obtained by a call to `get_result`.

Barriers that are configured for multiple arrivals per phase use an application-specified *reduction operation* (e.g. addition or minimum) to reduce the values provided with each arrival down to a single scalar result for the phase. This captures another common pattern in which a large data-parallel computation that produces a single value (e.g. a dot product or the determination of a maximum allowed time step) is executed in chunks. A separate task for each chunk determines a partial value for that chunk, and the partial results can be reduced with an appropriately-chosen reduction operator to yield the overall result. This form of reduction receives intense attention from implementers of runtimes based on implicit dependencies as they can easily become a performance limiter on large jobs. Many runtimes do not even offer non-blocking versions of *collective* operations such as these, and those that do are still unable to compose them with other deferred operations in the way that Realm is.

5.11 Causality

When explicitly constructing dependency graphs of operations, a correct application must always construct an *acyclic* graph. If two or more operations depend on each other in a cycle, none of them can ever be executed and the application will hang. However, not all applications are written correctly on the first try, and ideally a runtime based on explicit dependencies will provide facilities for detecting and fixing application bugs related to those dependencies. In the absence of `UserEvent` or `Barrier` usage (or race conditions through shared memory), the Realm API has the nice property that all programs are guaranteed to be cycle-free, or *causal*. This arises naturally from the requirement that the precondition for an operation must be provided before the handle for the operation itself exists — the precondition cannot possibly include itself, directly or indirectly.

The use of a `UserEvent` or `Barrier` invalidates this guarantee, as both provide an event handle for whose completion new preconditions can be added. Realm provides two capabilities that help detect and diagnose cycles that result from improper use of these primitives.

The first is a lazy hang detection mode that looks for a condition in which one or more operations are waiting on preconditions, and no operations anywhere in the system are ready or executing. No further progress is possible, so the runtime writes the unexecuted part of the operation graph to a file, and terminates with an error. A post-processing script can then be used to analyze the graph and identify instances of two patterns:

1. Any cycles in the graph are clear indications of a problem, although it may not always be obvious which edge(s) in the cycle are the erroneous ones.

2. Any `UserEvent` or `Barrier` for which there are no incoming edges. The lack of incoming edges indicates that one or more calls to `trigger` or `arrive` did not occur, either because the programmer forgot them or because the operation in which it would be performed has not been executed. At least one of these is usually the problem, but many others will be false positives if the task that will perform the missing call is in the transitive fanout of another incomplete `UserEvent` or `Barrier`.

The second cycle detection technique provided by Realm is an eager algorithm that can detect the formation of a cycle at the instant it is formed, allowing a trap into the debugger. This detection adds overhead for searching the operation graph, but only for calls to `UserEvent::trigger` or `Barrier::arrive`, as those are the only two that can complete a cycle in the graph. Still, the overhead is large enough that a user will not enable the eager detection algorithm until a hang has been detected by the lazy algorithm and the output of the post-processing script was not sufficient. The eager algorithm is also unable to detect hangs caused by the "untriggered" event case.

To further improve the situation, and also reduce the number of false positives from the lazy detection technique, Realm provides a way for the application to tell the runtime where these hidden dependencies lie. A call to the `Event::advise_event_ordering` method creates an *advisory dependency* between the specified `happens_before` and `happens_after` events. For example, an application that intends to trigger a `UserEvent` somewhere during the execution of a given task can use advisory dependencies to indicate that the `UserEvent` will happen after the precondition for the task, and happen before the task's own completion event. Advisory dependencies have no impact on the scheduling of operations, but they are reported in operation graphs and are used by the eager cycle detector. An additional check that can be enabled on request is a consistency check between advisory dependencies and the actual scheduling of operations. In particular, if any `happens_after` event occurs before its corresponding `happens_before` event, an error is reported to the user.

Chapter 6

Generational Events

As we have seen in the examples in the previous chapter, `Event`s are the glue of any Realm application. They are used to capture *explicit dependencies* between not just tasks and data movement, but all of Realm's *composably asynchronous* operations. `Event`s constrain execution order just enough for correct behavior and allowing Realm the flexibility to dynamically order tasks based on when their preconditions are satisfied.

This heavy use of explicit dependencies in a performance-critical environment demands an implementation that is scalable and minimizes overheads in time, space, and programmer effort. The time required to trigger an event places a lower bound on the size of operation that can be efficiently scheduled by the runtime. The space required to store dependencies between operations comes a the expense of the application's ability to store more data and can prevent scaling to large systems. Finally, any programmer effort that is involved in managing the dependencies takes away from other efforts and adds further opportunities for bugs in the application.

In this chapter, we describe the implementation of `Event`s in Realm and show how they satisfy all of these needs. We begin with a description of a basic distributed event that yields good performance and scalability properties. We then extend to the novel *generational event* structure that preserves these properties and addresses the competing concerns of space management and programmer overhead.

6.1 Basic Events

Figure 6.1 shows a small Realm operation graph. This graph was obtained from a single iteration of the Legion circuit simulation mini-app (discussed briefly in Section 4.5). Several different types of operations appear. Rectangles are used for tasks, parallelograms for data movement, and diamonds for operations involving `Reservation`s (see Section 5.6.1). Recall that Legion uses a dynamic mapping process to transform an implicitly-parallel application into one that runs on Realm, but

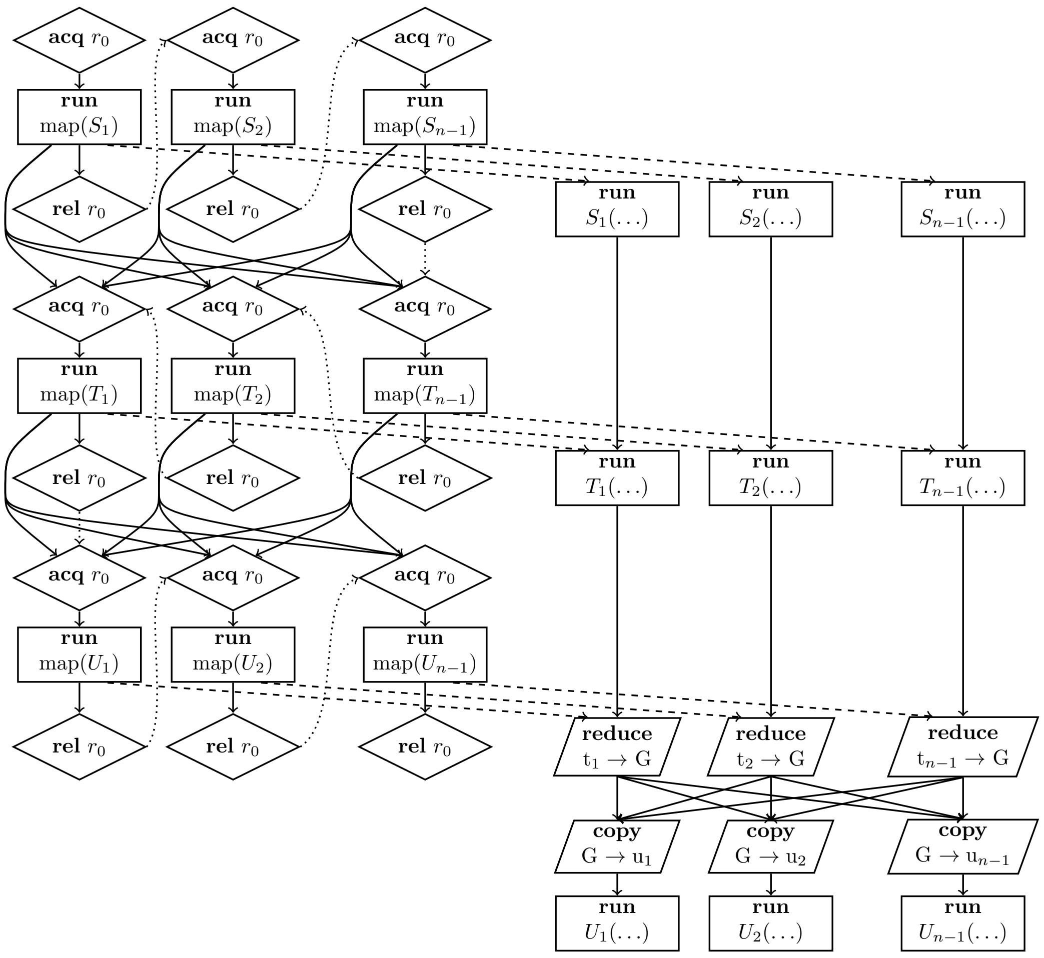


Figure 6.1: A Realm Event Graph

from Realm's perspective, there is no difference between the Legion "meta-tasks" that perform this mapping (organized on the left of Figure 6.1) and the actual application tasks themselves (on the right). The solid lines in the graph indicate explicit dependencies provided by the Legion runtime, while the dashed lines show parent-child relationships between tasks. Finally, the dotted lines show one possible ordering of the acquisitions of the `Reservation` r_0.

Even when executed on a cluster, the Realm operation graph is a global and dynamic construct. Every node may add operations to the graph during the course of execution, and dependencies may be added between any pair of operations, regardless of which nodes added them. As a result, even a basic event implementation needs to function correctly in a distributed environment and avoid the scalability bottlenecks that would arise from having a complete operation graph on any node in the cluster.

In most cases, `Event`s are created by Realm itself as part of handling a application request for asynchronous operations. (Applications can also create them explicitly in the form of a `UserEvent`.) An `Event` is a light-weight handle that can be freely transported around the system. An `Event` is *owned* by the node that created it, and the space of these handles is divided statically across all the nodes by simply including the node ID in the upper bits of the handle. This allows any nodes to create new handles without risk of collisions and without requiring inter-node communication. It also permits any node to immediately determine the owner of an `Event` without any communication.

The basic event is implemented as a distributed object, spread across one or more nodes. Each node uses the `Event` handle to look up their piece of the object as needed. This lookup uses a *monotonic* data structure allows wait-free queries even when updates are being performed. When a new `Event` e is created, the owning node o allocates a data structure to track e's state, which is initially *untriggered* but will eventually become *triggered* (or *poisoned* — see Section 8.5). The data structure also includes initially empty lists of *local waiters* and *remote waiters.* Local waiters are dependent operations (e.g., copy operations and task launches) that will be executed on node o. Remote waiters are other nodes that are *interested* in the `Event`, likely because they have operations that depend on it as well.

Once created, the `Event` handle may be passed around through task arguments or shared data structures and eventually used as a precondition for operations to be executed on other nodes. The first reference to e by a remote node n allocates the same data structure on n, sets the state to *untriggered*, and adds the dependent operation to its own local waiters list. An *event subscription* active message is then sent to node o indicating node n is interested and should be added to the list of remote waiters so that it may be informed when e triggers. Any additional dependent operations on node n are added to n's list of local waiters without further communication with the owner node. When e eventually triggers, the owner node o notifies all local waiters and sends an *event trigger* message to each subscribed node on the list of remote waiters. If the owner node o receives additional subscription messages after e has triggered, o immediately responds to the new subscribers with a trigger message as well.

The triggering of an event need not occur on the owner node. (The most common situation in which this occurs is with `UserEvent`s, but inter-node data movement often takes advantage of this as well.) When the `Event` e is triggered on a non-owner node t, a trigger message is sent from t to o, which forwards that message to all other subscribed nodes. The triggering node t notifies any local waiters immediately; no message is sent from o back to t. While a remote event trigger results in the latency of a triggering operation being at least two active message flight times, it bounds the number of active messages required per event trigger to $2N - 2$ where N is the number of nodes interested in the event (which is generally a small fraction of the total number of machine nodes). An alternative is to share the subscriber list so that the triggering node can notify all interested nodes directly. However, such an algorithm is both more complicated (due to race conditions) and requires $O(N^2)$ active messages. Any algorithm super-linear in the number of nodes in the system

Nodes	1	2	4	8	16
Mean Trigger Time (μs)	0.329	3.259	3.799	3.862	4.013

Figure 6.2: Event Latency Results.

will not scale well, and as we show in Section 6.2, the latency of a single event trigger active message is very small.

By limiting the instantiation of tracking data structures to just the owner node and other interested node, the Realm operation graph is effectively partitioned across the nodes of the cluster. Each node tracks subgraphs containing operations to be executed on that node, and dynamically connects its subgraphs to those on other nodes based on the dependencies between operations. By keeping both the storage and communication costs for a single node proportional to the number of operations (and their dependencies) that will execute on that node, Realm applications are able to scale to very large node counts without the overhead associated with the operation graph becoming a bottleneck.

6.2 Event Latency and Trigger Rates

We evaluate our Realm implementation using microbenchmarks that test whether performance approaches the capacity of the underlying hardware. All experiments were run on the Keeneland supercomputer[64]. Each Keeneland KIDS node is composed of two Xeon 5660 CPUs, three Tesla M2090 GPUs, and 24 GB of DRAM. Nodes are connected by an Infiniband QDR interconnect.

We use two microbenchmarks to evaluate basic event performance. The first tests event triggering latency, both within and between nodes. Processors are organized in a ring and each processor creates a `UserEvent` and performs a deferred trigger that is dependent on the triggering of the previous processor's `UserEvent`. Once the whole chain is constructed, the first event in the chain of dependent events is triggered and the time until the triggering of the chain's last event is measured; dividing the total time by the number of events in the chain yields the mean trigger time. In the single-node case, all events are local to that node, so no active messages are required. For all other cases, the ring uses a single processor per node so that every trigger requires the transmission (and reception) of an event trigger active message. (The event subscription messages are all sent during the setup of the chain, and are not included in the measurement.)

Table 6.2 shows the mean trigger times. The cost of manipulating the data structures and running dependent operations is shown by the single-node case, which had an average latency of only 329 nanoseconds. The addition of nearly 3 microseconds when going from one node to two is attributable to the latency of a GASNet active message; others have measured similar latencies[8]. The gradual increase in latency with increasing node count is likely related to the point-to-point

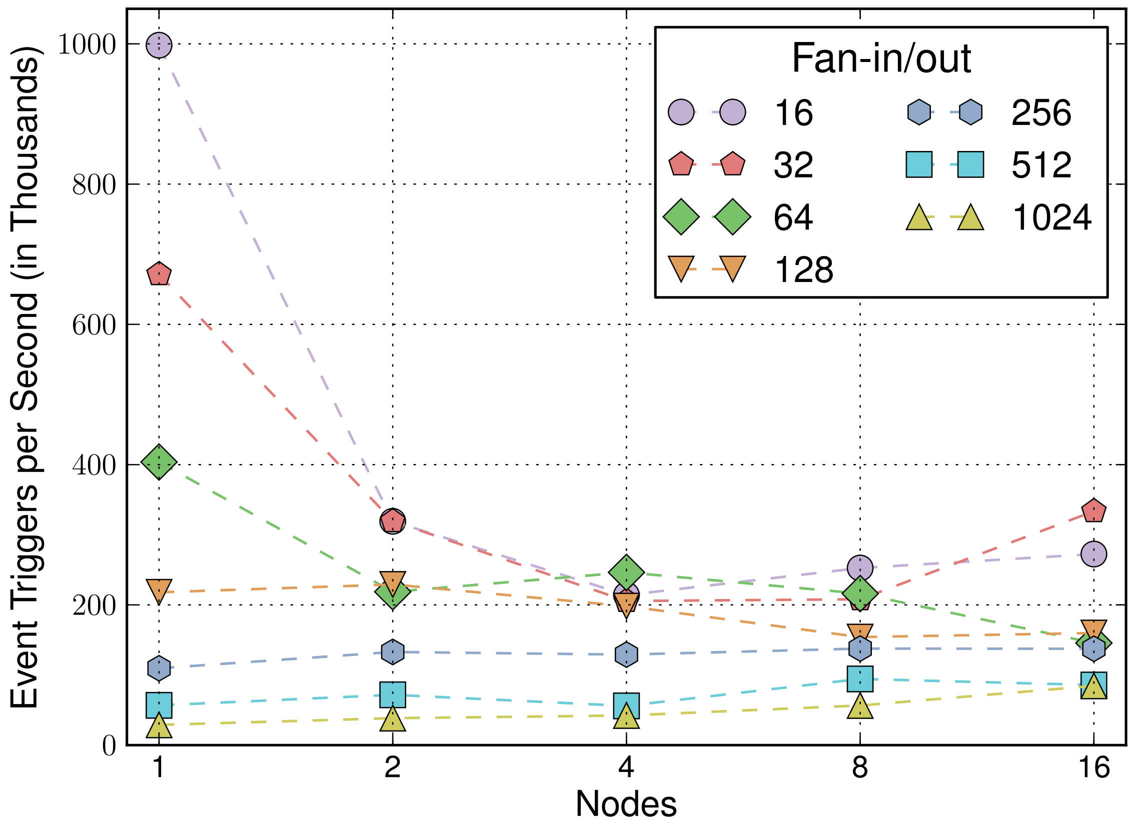


Figure 6.3: Event Trigger Rates

nature of Infiniband communication, which requires GASNet to poll a separate connection for every other node.

Our second microbenchmark measures the maximum rate at which events can be triggered by our implementation. Instead of a single chain, a parameterized number (the *fan-in/out factor*) of parallel chains are created. The event at step $i + 1$ of a chain depends on the ith event of every other chain. The events within each step of the chains are distributed across the nodes. When the fan-in/out factor exceeds the node count, there are multiple dependent operations for each remote event, but the use of a local tracking structure keeps the number of messages sent limited to two (one subscription and one trigger message) per event per node.

Figure 6.3 shows the event trigger rates for a variety of node counts and fan-in/out factors. For small fan-in/out factors, the total rate falls off initially going to two nodes as active messages become necessary, but increases slightly again at larger node counts. Higher fan-in/out factors require more messages and have lower throughput that also increases with node count. Although the number of events waiting on each node decreases with increasing node count, the minimal scaling indicates the

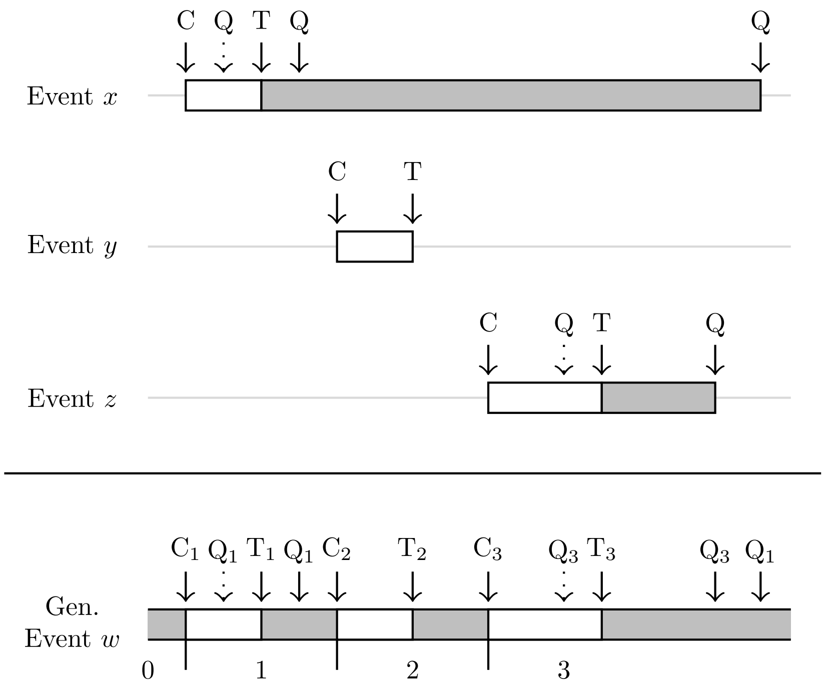


Figure 6.4: Generational Event Timelines

bottleneck is in the processing of the active message each node must receive rather than the local redistribution of the triggering notification.

The compute-bound nature of the benchmark shows that active messages do not tax the network and leave bandwidth for application data movement. The measured latencies and event trigger rates in these microbenchmarks suggest that Realm can handle operation as short-running as 10s of microseconds. That may be too large to handle the finest-grained (e.g. per-element) data-parallelism, but it is one or two orders of magnitude smaller than what is commonly seen in HPC applications.

6.3 Generational Events

The storage overhead associated with explicit dependencies depends both on the number of dependencies being tracked and the duration for which that tracking must be maintained. The distributed implementation of the basic event discussed above helps to limit the number of dependencies tracked by a given node, but does not address the temporal aspect. High-performance computing applications tend to be very long running, and if the memory used for tracking a dependency is not reclaimed somehow, the application will likely run out of memory and crash. However, there are several constraints on the lifetime of the data structure used to represent an `Event` e in a dynamic system. Creation and triggering of e can each happen only once, but any number of operations may depend on e. Furthermore, some of the operations depending on e may not even be requested until long after e has triggered. Therefore, the data structure used to represent e cannot be freed until it is certain that no more operations depending on e will be requested.

This problem is common to both implicit and explicit dependency systems, and most systems ask the program to manage dependency lifetimes themselves, creating and destroying them manually. This approach is used by StarPU[3], but also implicit systems such as MPI[58] and CUDA[50]. However, it can be very challenging to do this correctly, especially when dependencies cross between software module boundaries. When the application behavior is uncertain (e.g. data-dependent), an application must be conservative and potentially wastes space for dependencies which could have been reclaimed earlier. The Open Community Runtime[52] offers another option in the form of *once events*, which are automatically reclaimed by the runtime after they trigger. In exchange for the immediate reclamation, the application has to guarantee that all dependent operations are requested before event triggers, which requires constructing the operation graph backwards, from bottom to top. This is straight-forward for the source-to-source compilers that are the main target user of OCR, but is very unpleasant for manual programming.

The traditional alternative to manual memory management is garbage collection. OpenCL[38] offers reference counted events, which greatly simplifies the programmer effort and ideally reduces memory costs to exactly the events that are still live. However, the programmer must still correctly insert calls to adjust the reference counts and these calls add considerable runtime overhead, especially in a distributed memory implementation.

Realm offers what we believe is a novel alternative to the problem — one that automatically and aggressively recycles dependency structures. Like OCR's once events, programmer overhead is nearly eliminated, but there is no restriction on the order in which operations are requested. The Realm system similarly avoids the runtime overhead of reference counting, while using an amount of storage that is often *less* than the number of live events. The key observation is that one *generational event* data structure can simultaneously represent one untriggered event and a large number (e.g., $2^{32} - 1$) of already-triggered events.

We extend the `Event` handle to include a generation number in addition to the unique identifier for its generational event. Each generational event records how many generations have already triggered. A generational event can be reused for a new generation as soon as the current generation triggers. If an operation is requested that is dependent on a earlier generation, it is know to have already triggered and the new operation can immediately be executed (unless it has other preconditions as well).

To create a new `Event`, a Realm node finds a generational event it owns in the triggered state, creating a new one only in the very rate circumstance that all existing generational events owned by that node in the untriggered state, increases the generation by one, and sets the generational event's state to untriggered. As before, this can be done with no inter-node communication.

An example of how multiple `Events` can be represented by a single generational event is shown in Figure 6.4. Timelines for events x, y, and z indicate where creation (C), triggering (T) and queries (Q) occur. Queries that succeed (i.e. the event has triggered) are shown with solid arrows, while those that fail are dotted. The lifetime of an event extends from its creation until the last operation (trigger or query) performed on it. Although the lifetime of event x overlaps with those of y and z, the untriggered intervals are non-overlapping, and all three can be mapped on to a single generational event w. Event x is assigned generation 1, y is assigned 2, and z is assigned 3, according to the order in which they were created. A query on the generational event succeeds if the generational event is either in the triggered state or has a current generation larger than the one associated with the query.

The distributed tracking structure described for basic events in Section 6.1 is extended to support generational events through the addition of two fields. Both the owner and the remote node track the latest generation known to have triggered (this can be stale on remote nodes), while remote nodes also track the generation (if any) to which they are currently subscribed. There are no additional messages required on top of those used for the basic events — the subscription and trigger messages just carry a few more bytes in the form of the generation number. Remote generational events enable an interesting optimization. If a remote generational event receives a query on a later generation than its current generation, it can infer that all generations up to the requested generation have triggered. Even though it has not yet received the trigger message for the older generation, it knows the owner would not have created a newer generation unless the older had triggered, and it can get a head start on notifying local waiters for the older generation.

6.4 Event Lifetimes

To illustrate the both the need for some way to reclaim dependency tracking storage and the superiority of the generational event approach, we instrumented Realm to capture information about the lifetime of events, and looked at the three Legion applications discussed in Section 4.5. The usage of events by all three applications was similar, so we present representative results from just one. Figure 6.5 shows a timeline of the execution of the Fluid application on 16 nodes using 128 cores. The dynamic events line measures the total number of event creations. A large number of events are created—over 260,000 in less than 20 seconds of execution—and allocating separate storage for every event would clearly be difficult for long-running applications. The full S3D application (see Chapter 11) performs over 100 million Realm operations per run.

An event is *live* until its last operation (e.g., query, trigger) is performed. After an event's last operation, it is safe to reclaims the event's associated storage. The live events line in Figure 6.5 is therefore the minimum number of needed event tracking structured needed for either precise manual management of dependencies or a reference counting implementation. In this example, the number of live events also grows steadily over the run, but is less than 10% of the dynamic events count, a

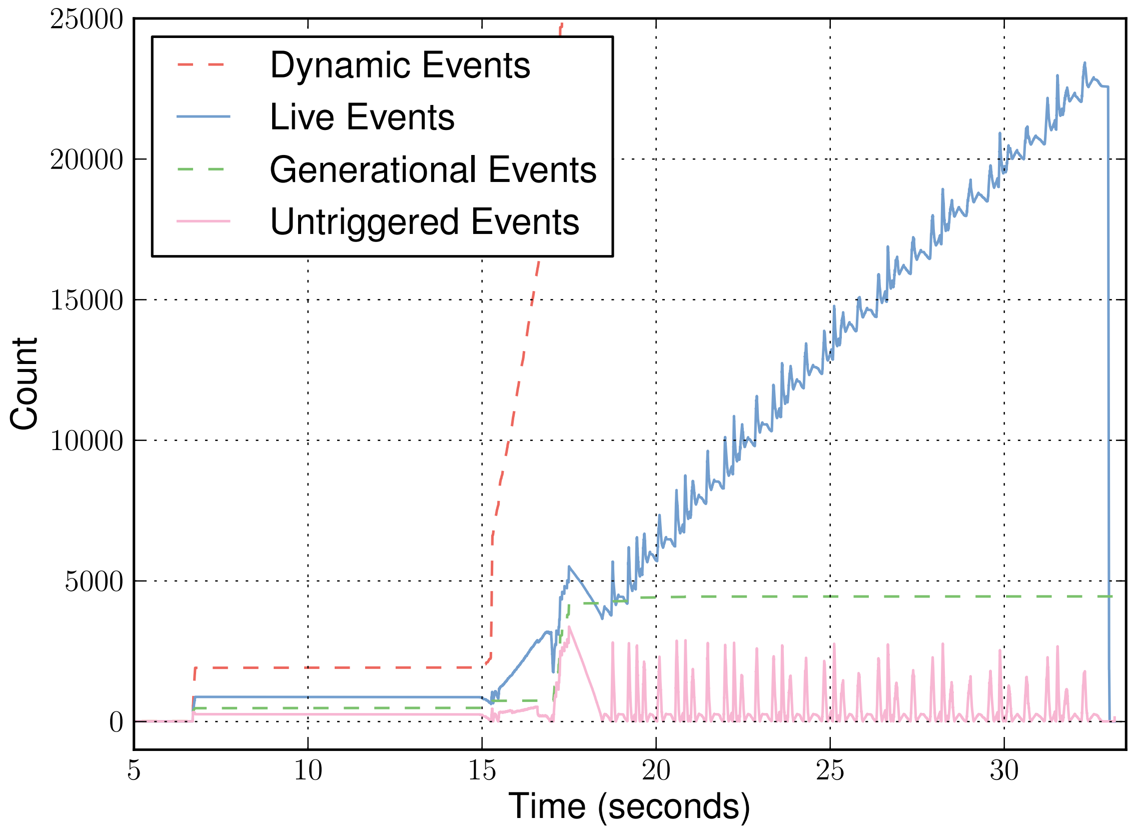


Figure 6.5: Event Lifetimes in Fluid Application

significant reduction in storage requirements. However, it comes at the cost of either programmer effort (for manual management) or runtime overhead (for reference counting).

As discussed in Section 6.3, our implementation requires storage that grows with the maximum number of untriggered events. Multiple live events can share a single generational event, provided no more than one is untriggered. The number of untriggered events is a further 10X smaller than the maximal live event count for this run, a gap that would likely continue to increase for longer runs. The green generational events line in Figure 6.5 shows the actual number of generational events created by Realm for this run. The maximum number of generational events needed is slightly larger than the peak number of untriggered events because nodes must create a new event locally if they have no available (i.e. triggered) generational events, even if there are available generational events on remote nodes. Overall, our implementation uses 5X less storage than existing manual management or reference counting implementations and avoids any related overhead. These savings would likely be even more dramatic for longer runs of the application, as the number of live events is steadily growing as the application runs, while the peak number of generational events needed appears to occur during the start-up of the application. Overall this demonstrates the ability of generational events to represent large numbers of live events with minimal storage overhead.

Chapter 7

Machine Model

The last few chapters have focused on one of the key contributions of Realm: a set of abstractions that provide functionally portable abstractions for launching computations on any processing element in a cluster, moving data between memories in the cluster, and composing these asynchronous operations in a way that maximizes the ability of Realm to hide the latencies inherent in any large system. These abstractions are carefully designed to provide the necessary *mechanisms* for adapting an application to any given system, while leaving all control of the *policy* in the hands of the application. No algorithm (nor any finite set of algorithms) can possibly compute the optimal mapping for all combinations of applications and target systems. In contrast, for a particular application and a particular target system, the optimal algorithm is often either obvious or easily discovered, and in such cases, an HPC user is willing to expend programmer effort and/or incur some initial analysis overhead to gain the benefits of that optimal mapping for a long running application. By explicitly delegating all policy decisions to the application, Realm allows the programmer to decide exactly how much effort or overhead they wish to expend, and guarantees that no time is wasted fighting or undoing any less-informed policy decisions being made by the runtime.

However, this delegation is not a complete abdication of responsibility, and we now turn to another key contribution of Realm: a collection of tools that help inform the application's policy decisions. This chapter discusses Realm's *machine model*, which provides a portable interface for discovering the (possibly dynamic) "shape" of the system on which the application is running. Chapter 8 covers the profiling capabilities of Realm, which address performance debugging needs in an asynchronous task-based runtime and allow an application's mapping policy to adapt to the observed real-time execution behavior. Finally, Chapter 9 describes Realm's dynamic code generation capabilities, that allow that adaptation to extend into the computational tasks themselves.

7.1 Machine Model

Chapter 5 described how a Realm `Processor` is used for any type of execution resource that can run a task, and a `Memory` is used for any memory resource that can store application data. A `Processor` may be queried for its *kind* (e.g. host CPU core, CUDA-capable GPU) as well as more architecturally-specific features (e.g. clock rates, supports for particular sets of instructions). Similarly, a `Memory` may be queried for its kind (e.g. system memory, memory that has been registered for remote DMA operations, or memory attached to a GPU), its capacity, and whether it supports features such as error detection or correction.

This per-resource information can be useful in deciding where to run a task or where to place data, but it is not sufficient. Modern supercomputers have complex and distributed memory hierarchies, and having data in a memory that is “close” to a given processor is critical. The Realm *machine model* provides this information by constructing a graph in which each resource (a `Processor` or a `Memory`) is a node. An edge between two nodes represents an *affinity* between those two resources. A processor-memory affinity represents the ability of a task running on that processor to directly access instances in that memory (using a `RegionAccessor` as described in Chapter 5). Similarly, a memory-memory affinity represents the ability to directly transfer data between that pair of memories using the `Domain::copy` method. Each affinity edge is annotated with an estimate of the best-case bandwidth and latency of data movement between the two resources. The affinity graph generated by Realm is flat and does not explicitly represent the boundaries between the nodes in a cluster. This approach permits inter-node heterogeneity to be described as easily as intra-node heterogeneity. The affinity graph is also a dynamic entity. It may change due to system events beyond the application’s control (e.g. the loss of a node or performance throttling due to power constraints), but this also anticipates future systems that may be allow a job to dynamically adjust its resource usage during execution.

Access to the machine model is provided by the singleton `Machine` object (obtained by a call to `Machine::get_machine`). The `Machine` object can be used to get a list of all the processors or memories in the system, which can be useful when determining an initial distribution of data in an application. It also provides methods for getting the list of neighbors of a given `Processor` or `Memory` in the affinity graph. Figure 7.1 shows an example of using these interfaces to generate a `graphviz` input file for visualizing the model of a machine, and Figure 7.2 shows the resulting graph for a single node of `sapling`, a development cluster used in this work. The graph is quite complicated even for a single node, and some manual tweaking of the output was required to produce a legible diagram.

These interfaces are essentially providing a raw snapshot of the affinity graph, and are intended for use by mapping code that wishes to perform its own analyses on the graph (e.g. clustering based on relative affinities) in preparation for making future mapping decisions. Such an analysis obviously needs to be prepared to handle large graphs when run at full-scale on modern supercomputers.

```
void print_affinity_graph(void)
{
  Machine machine = Machine::get_machine();

  std::cout << "graph␣M␣{\n";

  std::set<Processor> procs;
  machine.get_all_processors(procs);
  for(Processor p : procs)
    std::cout << "␣␣p" << p << "␣[␣style=bold;"
              << "␣label=\"" << pkind2str[p.kind()] << "\"];\n";

  std::set<Memory> mems;
  machine.get_all_memories(mems);
  for(Memory m : mems)
    std::cout << "␣␣m" << m << "␣[␣shape=box;␣style=bold;"
              << "␣label=\"" << mkind2str[m.kind()]
              << "\\nSize␣=␣" << (m.capacity() >> 20) << "␣MB\"];\n";

  std::vector<Machine::ProcessorMemoryAffinity> pmas;
  machine.get_proc_mem_affinity(pmas);
  for(Machine::ProcessorMemoryAffinity a : pmas)
    std::cout << "␣␣p" << a.p << "␣--␣m" << a.m << "␣["
              << (is_best(a, pmas) ? "␣style=bold;" : "") << "␣];\n";

  std::vector<Machine::MemoryMemoryAffinity> mmas;
  machine.get_mem_mem_affinity(mmas);
  for(Machine::MemoryMemoryAffinity a : mmas)
    std::cout << "␣␣m" << a.m1 << "␣--␣m" << a.m2 << ";\n";

  std::cout << "}\n";
}
```

Figure 7.1: Routine to print Realm machine model affinity graph

7.2 Address Spaces

A Realm application will ideally put all of its shared data into instances which can be copied between memories as necessary. This permits the boundaries between the processes running on different nodes to be completely ignored, and maximizes the options available when making mapping decisions. However, many support libraries used by HPC applications have been designed for the MPI style of running a separate process per core. They tend to make heavy use of global variables or other one-per-process constructs. As a result, concurrent operations executing in that process must coordinate properly when accessing them. The actual coordination can usually be achieved through the use of a separate `Reservation` in each process, but the application needs to know which

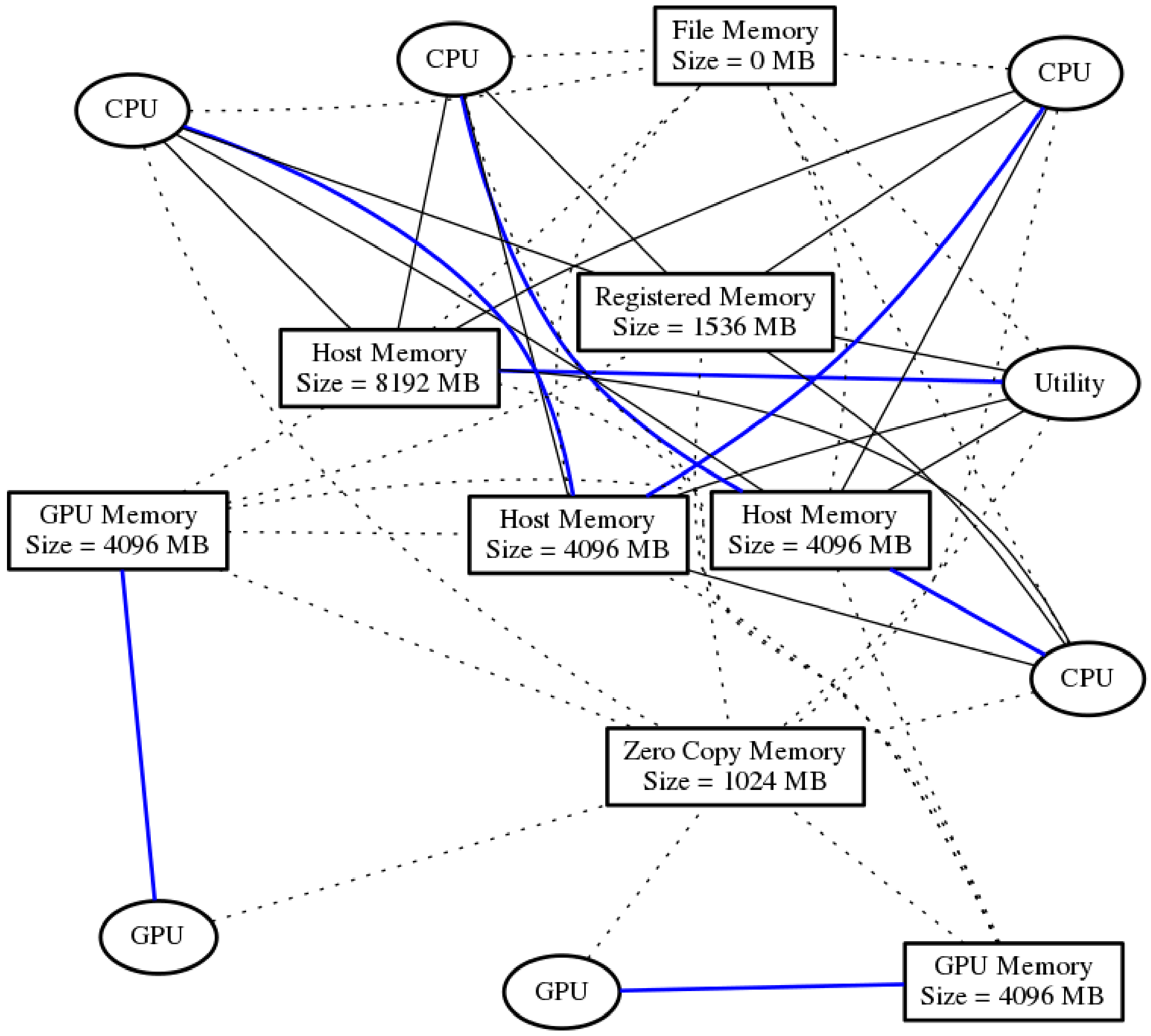


Figure 7.2: Visualization of affinity graph for single node of sapling

one should be used by tasks running on a given `Processor`. To support such cases, each resource in the machine model is associated with an *address space*, and methods are provided to list all the processors that are *local* to a given address space.

7.3 Query Interface

As described above, the machine model provides interfaces for enumerating the nodes and edges in the affinity graph, but this is not the most efficient way to answer common questions such as “how many processors are there in the system?” or “what’s the best memory to place data in for this processor?” Such queries can often be answered with both time and space complexities that are smaller than the size of the whole graph.

Queries are *prepared* by constructing either a `ProcessorQuery` or a `MemoryQuery` object and applying predicates to it. Common predicates are:

- `only_kind`(*kind*) limits the query to just processors (or memories) of the specified *kind*.
- `has_affinity_to`(*resource* [, *min_bandwidth*, *max_latency*]) filters out resources that do not have affinity to the specified other *resource*. If the optional *min_bandwidth* and *max_latency* arguments are provided, weaker affinities can be ignored as well.
- `best_affinity_to`(*resource* [, *bandwidth_weight*, *latency_weight*]) restricts the query to resources whose best affinity is the one to the specified *resource*. This can often return multiple resources (e.g. all the CPU cores in a NUMA domain prefer that domain's memory pool) or none at all (e.g. no processing element would prefer to interact with data directly in disk storage). By default, affinities are ranked purely based on their peak bandwidth, but the optional *bandwidth_weight* and *latency_weight* parameters can used to control this.
- `best_affinity_from`(*resource* [, *bandwidth_weight*, *latency_weight*]) reverses the above, restricting the query to the resource (or resources, in case of a tie) that has the best affinity from the specified *resource*. If this predicate is composed with others before it, only resources that satisfy the earlier predicates are considered for this test.

Once a query has been prepared, it can be executed by calling one of the following methods on it:

- `count` returns the number of resources satisfying the query.
- `first` returns the first matching resource. Sorting is based on the resource handles, so the ordering is consistent on all nodes.
- `nth`(n) returns the the nth matching resource (counting from 0).
- `random` selects a random matching resource.
- `begin` and `end` provide a standard C++ iterator for enumerating all matching resources.

For example, a simple implementation of the `select_saxpy_target` policy function (used in Figure 5.2) might select a random host processor that has the ability to access the data in the `initial_mem`, but then choose to move the data to another memory with better affinity to that processor, as long as the data can be copied (i.e. the two memories have an affinity).[1]

[1]This code relies on the property that every memory has an affinity to itself.

```
void select_saxpy_target(Memory init_mem, Processor tgt_proc, Memory tgt_mem) {
  Machine m = Machine::get_machine();
  tgt_proc = ProcessorQuery(m).only_kind(HOST_PROC).has_affinity_to(init_mem).random();
  tgt_mem = MemoryQuery(m).has_affinity_to(init_mem).best_affinity_from(tgt_proc).first();
}
```

As another example, the dumping of the affinity graph can be performed in a much more space-efficient fashion using iterators:

```
for(Processor p : ProcessorQuery(machine)) {
  std::cout << "  p" << p << ";\n";
  for(Memory m : MemoryQuery(machine).has_affinity_to(p))
    std::cout << "  p" << p << " -- m" << m << ";\n";
}
```

7.4 Subscriptions

A prepared query may be executed more than once, and Realm maintains a cache of results that can be reused as long as no changes to the affinity graph have occurred. If the application wishes to cache the results (or something derived from the results) itself, it needs to know when a stored result should be invalidated.

The Realm machine model allows for *subscriptions*, which request that a task be launched if a change is made to the affinity graph for any reason. As part of the subscription request, the caller specifies the target processor for any callbacks, the task ID, and any *user data* to be delivered to the task. A prepared query is an optional argument to the subscription request — if provided, callbacks will only occur if the change to the affinity graph is likely to change the resources matched by the query. The arguments to the callback task contain a list of the changes to the affinity graph. The task can either make changes to its cached results based on that data or simply execute the corresponding query again. The subscription request returns a `Subscription` handle that is useful only for canceling the subscription.

7.5 Manipulating the Machine

In its current form, the Realm machine model is strictly read-only. Any changes to the affinity graph are the caused by the system, and the application is a passive observer. Ongoing work is examining how an application might be able to make requests to modify the system, and the Realm `Machine` object is the natural place to supply such an interface. A number of different types of requests are being considered. The simplest ones would request to change processor clock speeds or allocated memory sizes. More complicated requests might involve merging two or more single-core CPU `Processor` resources into a single multi-core one that uses OpenMP for fine-grained data parallelism as described in Section 5.4. Going even farther, some have already imagined requests implementing elasticity by adding and removing whole nodes to match the application workload.

7.6 Application Models

We began this chapter by arguing that an optimal mapping depends on both the application and the target system. However, the discussion since has been exclusively about a model of the target system, and some readers may wonder whether there should be an "application model" as well. Many other runtimes do use such models, either inferring them or by extracting them from a higher-level description of the application. In particular, Legion does the latter using its system of logical regions and privileges, and defines a *mapping interface* that allows mapping policy decisions to be made based on models of both the application and the machine[7].

Nearly every such runtime defines the application model in a different way, which poses a challenge to any attempt to port an application from one runtime to another. Realm's implementation of the machine model tries to avoid this difficulty by defining a very general model that is completely application-agnostic, but in reality, the performance of a system depends to some extent on the exact workload being run. The profiling tools described in the next chapter help an application map out some of these grey areas where the application and machine models collide.

Chapter 8

Profiling

Even for the rare programmer who can write functionally correct code on the first try, getting that code to run at peak efficiency on a modern supercomputer is an iterative process. Static analysis can identify only the most egregious performance issues, so dynamic analysis is used instead. The application is run and *profiling* measurements are performed. These measurements can be used to identify problem areas in the application code. Comparisons between measurements from two different runs can determine sensitivity to different factors or be used to test the effect of a change to the code.

There are many existing tools for profiling of applications, but none are well suited to profiling a Realm application. The ideal profiling framework for a Realm application will be able to:

1. perform measurements on heterogeneous processor types,
2. analyze data movement and scheduling as well as task execution,
3. impact the application's behavior as little as possible, and
4. provide real-time results to the application itself allowing dynamic adaptation.

In keeping with Realm's self-designated role in the overall solution to performance portability, Realm's profiling interface focuses just on the problem of making measurements, leaving the questions of what measurements to perform, when to perform them, and what to do with the results to the applications (or perhaps a profiling library being used by the application).

Every request to create a resource or perform an operation in Realm accepts an optional `ProfilingRequestSet`. As the name suggests, this argument comprises zero or more `ProfilingRequest`s, which are requests from the application to perform measurements on the resource being created or the operation being launched. Each `ProfilingRequest` consists of a set of the measurements requested along with callback task information (a task ID, target processor, and optional user data, identical to the subscription callback used by the Realm machine model — see

Section 7.4). The use of a set of sets of measurements may seem redundant, but it allows more than one profiler to be active in the application without entanglement. (This can be especially useful when an application uses a third-party library that does its own dynamic tuning.)

8.1 Measurements

Each kind of measurement supported by the Realm profiling system defines a C++ type for its result. For example, the `OperationTimeline` measurement returns the times at which various things occurred for that operation:

```
struct OperationTimeline {
  typedef long long timestamp_t;

  timestamp_t create_time;   // when was operation created?
  timestamp_t ready_time;    // when was operation ready to proceed?
  timestamp_t start_time;    // when did operation start?
  timestamp_t end_time;      // when did operation end (on processor)?
  timestamp_t complete_time; // when was all work for operation complete?
};
```

The timestamps are measured in units of nanoseconds from application startup. (When running on multiple nodes, Realm synchronizes the timers at the beginning of execution.)

Not all measurements are fixed-size. For example, if an application is performing calls to `Event::wait` (despite constant discouragement), it can profile when they occur and how long they take with `OperationEventWaits`:

```
struct OperationEventWaits {
  struct WaitInterval {
    timestamp_t wait_start;   // when did the interval begin?
    timestamp_t wait_ready;   // when did the event trigger?
    timestamp_t wait_end;     // when did the interval actually end
    Event       wait_event;   // which event was waited on
  };

  std::vector<WaitInterval> intervals;
};
```

As mentioned above, resources may be profiled as well as operations. The exact spatial and temporal footprints of an instance are measured with `InstanceMemoryUsage` and `InstanceTimeline`. Contention on `Reservation`s can be analyzed with `ReservationContention`:

```
struct ReservationContention {
  size_t num_grants;     // total number of times reservation was granted
  timestamp_t average_held_time;  // average/longest time between
  timestamp_t longest_held_time;  //   grant and release
  size_t num_contended; // grants that had to wait due to contention
```

```
    timestamp_t average_wait_time; // average/longest time between
    timestamp_t longest_wait_time; //   request and grant
  };
```

The list of measurements is intentionally open-ended, and designed to make use of existing profiling tools when possible. If the PAPI[48] library is available, Realm will attempt to use it to perform measurements based on hardware counters, such as behavior of the various processor caches (hits and misses), instructions per clock (IPC) broken down by broad instruction type, TLB misses, and branch mispredictions.

Profiling measurements are *best-effort*. If a requested measurement is nonsensical (e.g. counting GPU texture cache misses on a task running on the host CPU) or if the necessary resources such as hardware counters are unavailable, that measurement is silently dropped. The callback for a given `ProfilingRequest` is always called exactly once, when all measurements have either been performed or dropped. The absence of a result for a requested measurement indicates that it was dropped.

By asking the application to request profiling measurements separately for each operation, the profiling overhead is ideally limited to just what is required to provide the data actually desired by the application. (This is not always possible — e.g. the CUDA driver currently profiles either all kernels or none.) However, if an application wishes to make the same measurements for multiple operations, the same `ProfilingRequestSet` may be supplied for each operation.

8.2 Example: Task-Level Execution Trace

The most obvious thing that can be done with the Realm profiling interface is to gather data to feed a traditional offline analysis. The Legion runtime does this if requested, simply asking for `OperationTimeline` measurements for every Realm operation it launches (both application tasks and internal Legion tasks). The callback tasks adds the timeline for each operation to a memory buffer, which is written out to a file at the end of the run. (Like Realm, Legion's profiling tools try to minimize the impact that profiling has on the application's execution timing.)

This data can then be analyzed and shown in an interactive viewer, allowing the programmer to look at the execution time of color-coded tasks, but also identify when processors are idle due to scheduling issues or lack of parallelism. Figure 8.1a shows a screenshot of a Legion application similar to the heat diffusion example from Chapter 2. The job was run on 8 compute nodes (profiling data for only the first 4 are shown in this screenshot), using 8 cores for application task and 1 *utility processor* for internal Legion runtime tasks. (Recall that Legion is just part of the application from Realm's perspective.) Each row is a Realm `Processor` and they are grouped by *address space* (i.e. node).

Looking at the overall runtime, things look pretty good — the application cores are mostly full and the utility processor is mostly idle. However, the per-operation data provided by Realm allows the programmer to zoom in (Figure 8.1b) and identify a number of areas for improvement. For example, although the average utilization of the utility processors is quite low, their usage appears to come in bursts and may be partially to blame for the gaps around the blue tasks. Similarly, we can see that occasionally some of the orange tasks are delayed in their start — this fact would be lost in the noise when averaged over the whole execution.

8.3 Example: Empirical Work Distribution

Our second example demonstrates one way in which the Realm profiling and machine query interfaces can be used to make an application's mapping policy decisions. Consider the call to `select_target_processor` in the `Reservation` usage example (Figure 5.4) in Section 5.6.1. The policy call is given a `RegionInstance` and returns the `Processor` on which the task should be run. Assuming we wish to pick a processor that can access that instance directly (this is itself a policy decision), we still need to select one. Different processors may run the task at different speeds (because they are of different kinds, or have non-uniform paths to the instance's memory, or any number of other reasons), so a simple round-robin distribution is unlikely to suffice.

Our approach instead uses a weighted random distribution. If the weights are inversely proportional to the expected runtime of the task on each processor, this will maximize the overall throughput. These expected runtimes will be based on real-time profiling feedback. Figure 8.2 packages up the needed functionality into a class. For simplicity, it is specific to a single task type and uses STL containers.

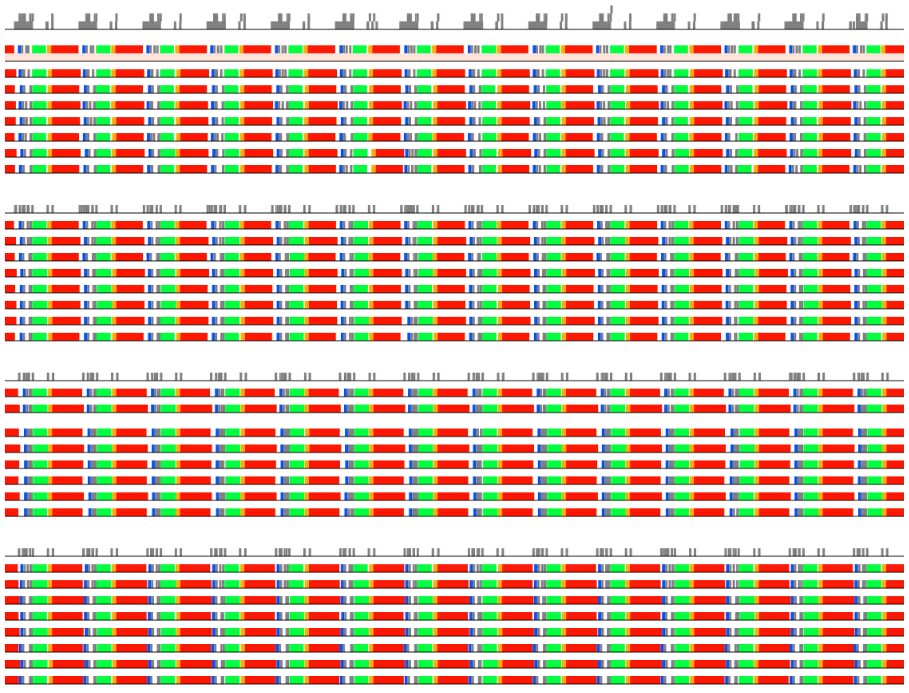
Lines 8-32 define the `select_target_processor` method. In addition to the `RegionInstance` argument that is used to determine the candidate processors, we've added a reference to a `ProfilingRequestSet` from the caller. As we'll see, this allows the question of whether or not to do profiling to be answered dynamically as well.

1. Line 11 prepares a `ProcessorQuery` that limits results to those that have affinity to the `Memory` in which *inst* resides. The query is not executed yet, as it might be used in two different ways.

2. The first possible use is on line 15. If we are still gathering our initial data, a random `Processor` is requested of the query, allowing us to hopefully gather samples for all candidate processors.

3. However, once the load balancer is done with the "warmup" stage, it switches to the weighted random distribution. Lines 19-24 performs a single iteration over all candidate processors, choosing (and often re-choosing) a processor based on how its weight compares to all the previously observed weights.

4. Lines 27-29 form the `ProfilingRequest` that will obtain another `OperationTimeline` sample for the load balancer's runtime estimator, but only if it is worthwhile. This code uses a simple sample count limit, but a better algorithm would examine the current variance of its samples


(a) Zoomed out (most of the application)

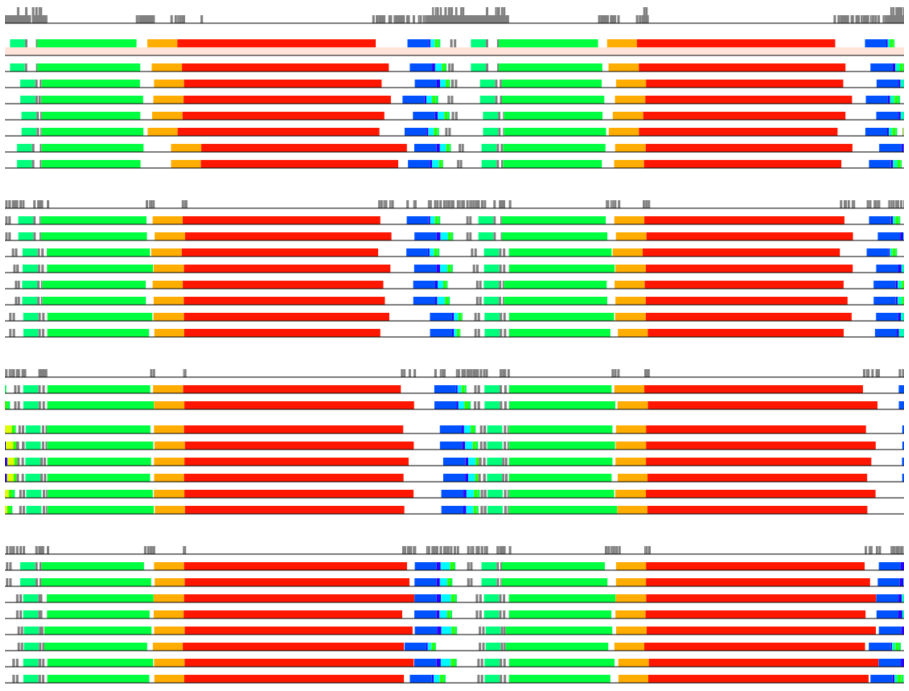

(b) Zoomed in (two iterations)

Figure 8.1: `legion_prof` execution timeline

and/or reduce the sampling rate gradually instead of all at once. This also demonstrates the benefit of allowing multiple independent requests in a `ProfilingRequestSet`, as the code here can request what it needs without coordinating with any other code that is also performing measurements (either the same or different ones) elsewhere.

5. Finally, line 31 returns the chosen processor to the caller.

Lines 34-45 define the profiling callback task body. The code to register it is not shown, but since it is a static method, all instances of `DynamicLoadBalancer` can use the same `profiling_task_id` (line 6).

1. Profiling measurement results are passed to the callback task in a compressed bitstream. Line 37 converts this into a `ProfilingResponse` object. This conversion does not need to unpack all of the measurements — the compression format is designed for random access.

2. Line 39 requests the `OperationTimeline` measurement result and tests to make sure it exists. (This is good practice even though there is currently no case in which an OperationTimeline request will be dropped for a task.)

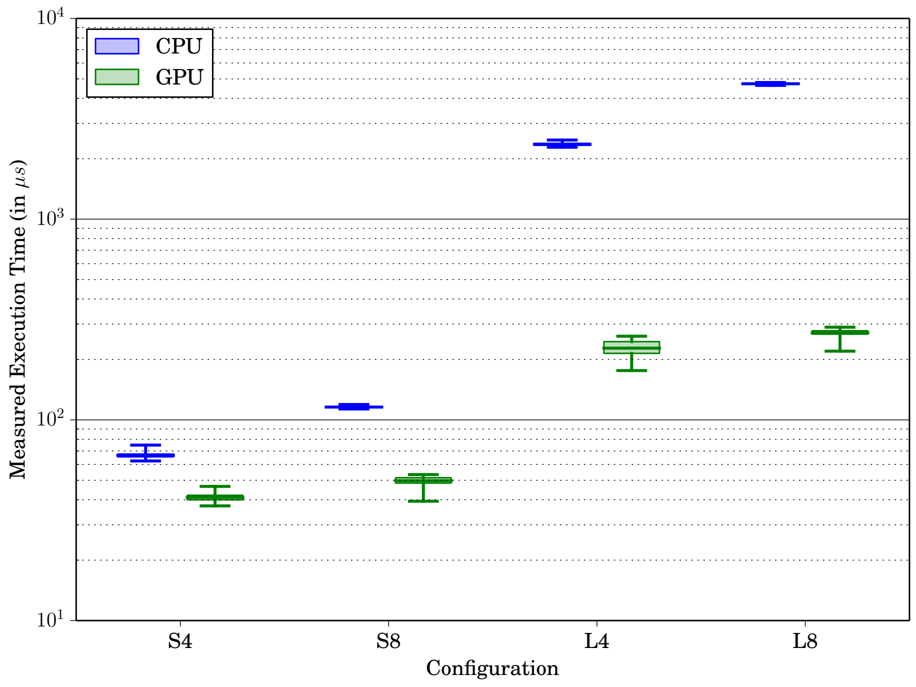
3. Since `profiling_callback` is a static method, we need a pointer to the right `DynamicLoadBalancer` object to update. Line 40 fishes that out of the task's *user data* (which was supplied on line 28).

4. Lines 41-43 update the statistics for the specified processor. Note that that the parameter p is the `Processor` on which the callback task is called, which is only the same as the one that executed the profilee because we demanded it on line 28.

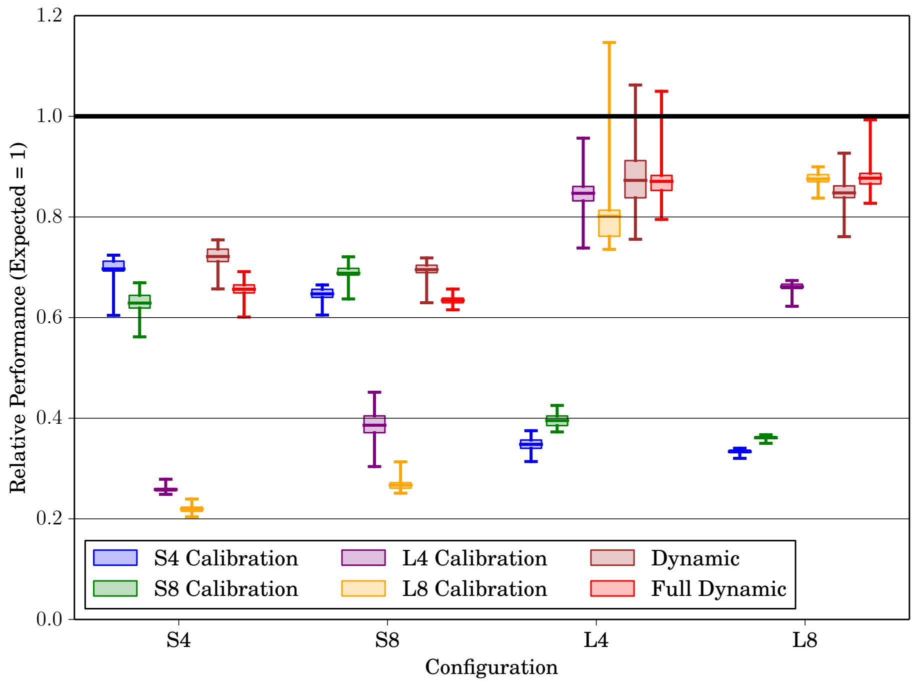
We explore the benefits of the empirical work distribution enabled by Realm's profiling interface by looking at a microbenchmark based on the SAXPY kernels described in Chapter 5. The test performs repeated SAXPY operations on a vector that is 100's of MB in size — too large to fit in processor caches. The vector is divided into pieces of a programmable size, and each piece is computed by a separate task. The target system is heterogeneous, containing a GPU as well as two CPU sockets with a non-uniform memory architecture. By varying two parameters, we obtain four different execution configurations, and are interested in obtaining the best possible performance for each. The first parameter is the size of the vector pieces given to each SAXPY task, and may be "small" (32k elements) or "large" (1M elements). Tasks based on larger pieces will obviously take longer to run, but the more important impact is that GPUs tend to be significantly less efficient when operating on smaller pieces of data. Similarly, the second parameter tries to capture variability in CPU core performance by using either 4 or 8 total CPU cores (i.e. 2 or 4 per socket). Although each core has its own datapath resources, they share the same memory access path. Doubling the execution resources often results in slower execution on a given core due to competition for the memory.

```
class DynamicLoadBalancer {
public:
  size_t samples_taken;
  std::map<Processor, size_t> proc_samples;
  std::map<Processor, double> proc_total_runtime;
  static TaskID profiling_task_id;

  Processor select_target_processor(RegionInstance inst, ProfilingRequestSet& prs)
  {
    // build a query of processors that have access to the instance
    ProcessorQuery pquery = ProcessorQuery(Machine::get_machine())
                                                               .has_affinity_to(inst.get_location());
    Processor best;
    if(samples_taken < WARMUP_SAMPLES) {
      best = pquery.random();      // not enough data yet - just pick randomly
    } else {
      // iterate over the choices, choose based on weight
      double total_weight = 0;
      for(Processor p : pquery) {
        double weight = proc_samples[proc] ? (proc_samples[proc] / proc_total_runtime[proc]) : 1e10;
        double p_switch = weight / (weight + total_weight);
        if(drand48() < p_switch) best = p;
        total_weight += weight;
      }
    }

    if(samples_taken < MAX_SAMPLES)
      prs.add_request(ProfilingRequest(best, profiling_task_id, &this, sizeof(this)))
                .add_measurement<OperationTimeline>());

    return best;
  }

  static void profiling_callback(const void *args, size_t arglen,
                                 const void *userdata, size_t userlen, Processor p)
  {
    ProfilingResponse resp(args, arglen);
    OperationTimeline info;
    if(resp.get_measurement(info)) {
      DynamicLoadBalancer *me = *(DynamicLoadBalancer **)userdata;
      me->total_samples++;
      me->proc_samples[p]++;
      me->proc_total_runtime[p] += 1e-9 * (info.end_time - info.start_time);
    }
  }
};
```

Figure 8.2: Load balancing based on real-time Realm profiling



(a) Calibration Results



(b) Application Performance

Figure 8.3: Benefits of Empirical Work Distribution

We expect each of these configurations to have different ratios of CPU and GPU performance, but the exact ratios (which are essential for optimal load-balancing) are nearly impossible to predict from first principles. The standard approach to this problem is to use offline profiling tools to characterize the application's performance and use that to compute a static load-balancing ratio for a given configuration. We start with this approach as well, running each configuration 100 times and using the Realm profiling tools to report the average time taken for the CPU and GPU tasks on each run. The distributions across the runs are shown in Figure 8.3a. The configurations are named based on whether they use Small or Large pieces and the number of CPU cores. (i.e. S4 is the small piece size with 4 CPU cores.)

As expected, task runtimes are longer for the large pieces, with the CPU tasks slowing down more than the GPU tasks. The difference between the 4 and 8 CPU core configurations is a little more surprising. First, the CPU tasks are almost twice as slow in the 8 core case, suggesting that the memory system was already near its limit with only 4 cores active. Second, there is an unexpected slowdown in the GPU tasks when going from 4 to 8 cores, perhaps due to the CUDA driver running on the CPU and trying to use the same oversubscribed memory access path. This unexpected influence highlights the danger of hidden variables when attempting to characterize a workload in an environment that differs in any way from the actual execution environment. A second warning sign that there may be hidden variables at work is the variability in the results from run to run on the same configuration, which exceeds 20% in several cases.

Using the results of the profiled calibration runs, "optimal" load-balancing ratios are computed for each of the four configurations. To these four static load-balancing modes, we add two dynamic modes. The first uses the strategy discussed in this section, profiling just the first 5% of the actual execution and using that to drive load-balancing for the rest of the run. The second is similar, but keeps profiling enabled for the full execution, allowing the ratio to be continually adjusted. Each of the configurations was then run 100 times for each of the six modes. The overall runtime of each run was compared to an estimate of the best possible performance for that configuration, based on just the task execution times from calibration runs without considering costs of synchronization or other overheads. The resulting distributions for each mode and configuration are shown in Figure 8.3b. Unsurprisingly, the best static load-balancing mode for each configuration is the one native to that configuration. The enormous performance hit for using a load-balancing ratio based on the wrong piece size suggests strongly that several intermediate sizes should be characterized for this application if other sizes are to be used.

The two dynamic modes perform very well in all cases, with the algorithm described in this section usually out-performing even the static load-balancing ratio computed for the exact configuration being used. This is likely due to the dynamic mode's ability to capture the performance properties of the exact execution environment. The mode that profiles for the entire run suffers a little for the smaller piece size due to the profiling overhead. When the tasks being profiled are very short, the more granular control of partial profiling is important. However, for the larger runs, the benefits of additional adjustments to the load-balancing approach can also be seen. As mentioned above, a partial profiling approach that gradually decreases the number of tasks being profiled over the course of the run would likely yield the best of both of these dynamic modes.

8.4 Profiling Realm Itself

Although it is not exposed directly to the programmer, Realm includes some infrastructure for profiling the runtime implementation itself. It uses some similar techniques to those described here for internal "tasks" (e.g. handling inter-node messages of a given type), but also uses more traditional techniques for the "glue" code that holds everything together. Standard execution time profiling such as `gprof` or VTune have been very effective at finding slow code. However, to debug memory use bugs, a custom sampling profiler was implemented, allowing memory use by different subsystems in Realm (e.g. task scheduling, memory transfer, event handling) to be easily distinguished.

8.5 Fault Tolerance

An increasing concern in the HPC community is the occurrence of hardware faults that occur during long-running applications. The standard practice is to periodically save the entire application state to a *checkpoint* on disk. When a fault of any kind occurs, the application is allowed to crash and the user restarts it from a previous checkpoint. If fault rates continue to rise (and they are expected to), the overhead of this approach becomes intractable. Instead, applications will need to be able to contain faults and recover from them without having to restart the entire application. Both containment and recovery are only possible with information about the nature of the fault that has occurred. And while recovery methods fall very much in the category of application policy that Realm tries to stay out of, both the reporting of faults and their containment are areas that Realm assists with.

Faults can occur in many forms. Execution faults that occur while a task is running are trapped by Realm and reported via the profiling interface. The basic result (pass or fail, with a small amount of failure-specific detail) is measured using `OperationStatus`, whereas a detailed backtrace of the precise error location (if available) is measured using `OperationBacktrace`.

Most systems that report faults or exceptions take advantage of a program's call stack to propagate unhandled errors to the calling function, and terminate the application only if no handler exists anywhere in the stack. Unfortunately, Realm's flat operation graph lacks that hierarchy, and if an `OperationStatus` measurement is not requested for a specific operation, there is nowhere else to report a fault that occurs in that operation. An unobserved execution fault cannot possibly be recovered from, so Realm terminates the application in this case. It detects this condition at the point of the fault so that a core dump (if enabled) will allow post-mortem debug of the offending code.

Memory faults can occur that corrupt the contents of a single `RegionInstance` or, if the error is not to a specific address, all instances in a `Memory`. A memory fault is reported directly in an `InstanceStatus` measurement, but can also be observed indirectly via an execution fault on any task that has an active `RegionAccessor` or requests one in the future. As a result, an unobserved abnormal `InstanceStatus` is not considered to be a fatal error.

The final category of fault reported by Realm is the loss of hardware resources — a `Processor` or a `Memory`. An individual resource can be lost in some cases (e.g. a GPU can fall "off the bus"[61]), but the most common cause of hardware resource loss today is a node crash, which causes the loss of all `Processor`s and `Memory`s on that node. The loss of a resource, whether one or many, is reported as a change to the machine model, and possibly through the abnormal status of any operations or instances using the resource(s) in question.

With execution faults contained to the task in which they occurred and failures reported to the application for a recovery policy to be chosen, Realm's involvement is nearly complete. The remaining question is what should be done with operations that are dependent, either directly or transitively, on a failed operation. Realm addresses this by *poisoning* the completion event of a failed operation and propagating that poison forward through the operation graph. Whereas a normal `Event` signifies the successful completion of an operation at some point in (finite) time, a poisoned `Event` represents one that will *never* successfully complete. An operation whose precondition will never occur can clearly never occur either, so it is discarded and its completion event is also poisoned. The transitive fanout of the failed operation is effectively stripped out of the operation graph, relieving the application of that burden (and of the bookkeeping required to even implement it). The application is then free to re-attempt the computation with an isomorphic operation graph or to choose a completely different recovery path.

If a task interacts directly with a poisoned event (i.e. using `has_triggered` or `wait`), it automatically becomes poisoned as well, allowing most code to written without directly considering fault tolerance. A task that is able to contain faults in some way may use *fault-aware* versions of those methods that inform the caller if an event is poisoned without propagating it automatically.

Chapter 9

Dynamic Code Generation

This chapter continues the discussion of the mechanisms Realm provides to applications (or libraries) to obtain performance portability on modern supercomputers. We started with an introduction to the machine model in Chapter 7, which allows an application to understand what types of processors are available and where they sit relative to each other in the system's memory hierarchy. This can form the basis for static distributions of the application's work. We then added the profiling framework in Chapter 8 which allows the application to measure the quality of that initial distribution and make dynamic changes to it to improve performance, switching to different task implementations and/or different processor kinds.

However, it may be that the profiling data indicates that none of the available task implementations are ideal. An implementation that has been specialized for one or more properties of a specific workload can often run significantly faster than a more generic implementation. Specialization might be based on the data size or its layout in memory. It may also take advantage of input-specific runtime constants or mode settings. These specialized tasks can be pre-built in some cases, but in others the set of possible combinations is too large to enumerate at compile-time. In this chapter, we explore Realm's support for dynamic code generation.

Earlier examples have used two different ways to register tasks on processors. The `Runtime::register_task` method only accepts a pre-compiled function pointer and must be called before any top-level tasks are launched, making it unsuitable for dynamic code generation. Once a top-level task has been started, tasks must be registered with either the `Processor::register_task` or the `Processor::register_task_by_kind` method, the latter of which was used in Section 5.4. Instead of a function pointer, these methods accept a `CodeDescriptor`, which allows the task's code to be described in one or more ways.

A particular description is supplied by an object that implements the `CodeImplementation` interface. The simplest implementation of this interface is the `FunctionPointerImplementation`, which holds a pointer to an arbitrary function that is executable on the host CPU. A key attribute of any code implementation is whether or not it is *portable* — can it be transferred from one node to another safely? Raw function pointers are not portable on many systems due to *address space layout randomization* (ASLR), an OS feature commonly enabled for its system security benefits. Realm does not attempt to detect the presence of ASLR, but accepts a command-line argument (`-realm:portable_fp`) that treats function pointers as portable.

A `CodeDescriptor` also specifies the *type* of the described function. At present, all tasks have the same prototype, so the explicit type field is primarily for catching application bugs. However, other interfaces that use application-specified code (e.g. reduction functions) are expected to move to the use of `CodeDescriptor`s as well. Additionally, future `Processor` kinds may specify require additional arguments to tasks (e.g. a *rank* parameter for an MPI processor kind).

A given kind of Realm processor specifies which `CodeImplementation` variants it can accept directly for a registered task. For example, the host CPU kind (and the CUDA kind, for the reasons described in Section 5.4) accepts only a `FunctionPointerImplementation`. If the set of implementations in the application-supplied `CodeDescriptor` does not include one that is suitable for the target processor, a *code translator* must be used. A code translator may be necessary for task registration even if the preferred implementation is provided, if the target processor is remote and implementation is not portable.

Several examples of code translators (and often accompanying code implementations) will be covered in subsequent sections, and the interface itself is described in Chapter 10. Code translators are not directly visible to application code, but this anonymity may not last. It is likely that cases will arise in which there are multiple viable code translation paths, and Realm lacks the knowledge required to select the right one. A faithful adherence to the separation of policy and mechanism demands that the control over the translation path, and any options for the translators along that path, be chosen by the application at task registration time.

Like all other Realm operations, task registration is composably asynchronous. The registration of a task can require compilation and/or communication, so an `Event` is used to describe its (successful) completion. Task launches may be immediately requested, as long as they use the registration's completion event as a precondition. Also like any other operation, task registration may be profiled. The `OperationTimeline` and `OperationStatus` measurements are currently supported, and it is expected that path-specific measurements will be in the future. For example, the CUDA driver can provide information about the generated code (e.g. number of registers used) that are important for any load-balancing decisions the application may wish to make.

Task registration operations may fail for a variety of path-specific reasons (e.g. ill-formed LLVM IR). Such failures are reported via the `OperationStatus` measurement, and Realm uses the same rules as for execution faults in tasks — an unobserved abnormal status is considered to be a fatal error. Task launches preconditioned on a failed registration will be skipped due to the poisoned precondition.

9.1 Dynamic Loader

The first code translator, available in nearly all cases, provides the ability to load shared libraries (also known as *dynamic shared objects* or DSOs) and map from symbol names to pointers into the shared library image loaded in memory. (This uses the standard POSIX `dlopen` and `dlsym` library calls.) A reference to code in a DSO is captured by a `DSOReferenceImplementation` which is composed of two string fields: *dso_name* should be the name of the shared library on disk, and *symbol_name* should be the symbol name to look up. The dynamic loader translator provides to Realm a path to convert from a `DSOReferenceImplementation` to a `FunctionPointerImplementation`. This translation may fail if either the specified file cannot be opened or is not a shared object, or if the specified symbol name does not exist in the object.

Although nonstandard, many systems provide a `dladdr` library call that attempts a reverse mapping, taking a pointer to a function in memory and returning a symbol name and the filename of the DSO from which it was loaded. If this call is available, the translator also offers the ability to convert from a `FunctionPointerImplementation` back into a `DSOReferenceImplementation`. This capability, in conjunction with the right compiler option to generate shared linkage information for functions in the main executable, can allow a `FunctionPointerImplementation` to appear to be portable even with ASLR is available — Realm will convert it to a `DSOReferenceImplementation` to send from one node to another and then convert back on the target processor's node.

9.2 LLVM

A second code translator, and one that enables truly dynamic code generation, is based on the LLVM compiler[42], which can be embedded in other applications (or in this case, runtimes) to provide *just-in-time* (JIT) compilation capabilities. The compiler accepts input in the assembly language of its low-level virtual machine (hence the name LLVM), commonly known as the LLVM Internal Representation (IR)[44]. Compilation is organized as a series of *passes*, including most standard (and some more esoteric) optimizations. The generated machine code for a variety of architectures is competitive with that of other (ahead-of-time) compilers.

Realm's LLVM code translator defines a new code implementation, the `LLVMIRImplementation`. It consists of two main components, the first being a block of data containing the IR. It may be in either the human-readable text format or the binary *bitcode* format supported by LLVM. The IR should represent a self-contained module — external references to standard library calls are permitted, but all other code dependencies should be included. (In particular, the code should not assume symbols from previously-compiled modules are available.) However, the code need not be flattened. Multiple functions may be included as part of the module, and a second field of the `LLVMIRImplementation` object specifies the *entry symbol* — the place where the registered task's execution should start.

Although designed to be faster than traditional ahead-of-time compilers, the JIT compilation in LLVM is by no means instant. To avoid stalling other runtime activities, the compilation is performed on a separate worker thread. By default, one worker thread is used per node, but this number can be increased through the use of a command-line parameter. Unlike some of the more latency-sensitive Realm background threads, compilation worker threads are suspended when idle and will not compete for execution resources with other Realm internal operations.

Realm's task registration API often leads to usage models in which the same task is registered with different processors at different times. Although the LLVM code translator compiles each module independently from all others, it does include a *memoization* optimization, recognizing the module in a compilation request exactly matches a previous one and reusing the previous result. To avoid keeping around the entirety of potentially-large module inputs, hashes are computed and compared instead.

The LLVM compiler is constantly evolving, but at present provides both eager compilation (MCJIT) and lazy compilation (ORC), in which the actual optimization is delayed until the target function is called. While this initially sounds like a good thing for a deferred execution model, it is problematic for two reasons. First, the compilation will take some time, and if that latency is incurred as part of executing the task, Realm lacks the ability to schedule some other operation while waiting for the compilation to complete. Second, a compiled task will often be used by several different processors of the same kind (at least on the same node), which can introduce issues of thread-safety. Although the LLVM maintainers often discuss whether or not the compilation framework should be thread-safe, it is not currently, and Realm would be forced to add the synchronization itself, causing coupling between supposedly independent `Processors`. As a result, Realm's LLVM code translator uses the MCJIT path. However, if it is deprecated as some are suggesting, additional work may be needed on the Realm side to hide JIT compilation latency in other ways.

9.3 CUDA

The LLVM code translator works well for host CPU code, but is not sufficient for tasks that will be run on CUDA-capable `Processors`. There are two reasons for this. First, the LLVM back-end code generator does not generate actual machine code for GPUs. The instruction set used by GPUs can vary greatly from one architecture to the next, so NVIDIA does not publish the details of the instruction sets. Instead it defines its own *parallel thread execution* (PTX) virtual machine and publishes its instruction set instead. Tools such as LLVM must generate PTX output, which is then lowered to a specific GPU's instruction set by the CUDA toolkit. This lowering can be done ahead-of-time by the compiler, or just-in-time through the CUDA driver API.

The second challenge in dynamically generating code for tasks on CUDA involves the CUDA programming model's use of the host CPU. A Realm task for a GPU processor must consist of

GPU machine code for the kernels but also host CPU machine code that launches those kernels. A `CodeDescriptor` that describes code for a CUDA processor kind must include implementations for both of these somehow.

Rather than define its own code implementation format, the approach used by the CUDA code translator is use the `LLVMIRImplementation`, and capture both the CPU and GPU code in a single module. The LLVM IR format allows the creation of *attributes*, and the CUDA code translator expects the functions within the module that are to be run on the GPU to be tagged with a special attribute. The translator looks for these attributes and splits the input module into two pieces. The first contains just the tagged functions (and functions called by them) and is given to one of the LLVM background threads to be compiled to PTX. The PTX is then given to the CUDA driver to be lowered into GPU machine code and the location of the resulting code is requested. The second piece of the original module has the tagged functions removed. Further, calls to those removed functions are replaced with the necessary code to perform a kernel launch, using the address of the corresponding kernel returned by the CUDA driver. This modified code can finally be given an LLVM worker thread to be compiled into host CPU code that will form the entry point for the new task.

Having both the host CPU and CUDA GPU code translation paths start from an `LLVMIRImplementation` should be of benefit to application code generators that want to generate code for both paths — much of the generated structure can be the same. (In fact, the degenerate case of the CUDA path in which no functions are tagged becomes identical to the host CPU code generation path.) However, many existing code generation tools for CUDA choose to generate PTX directly, and the easiest way to incorporate such tools into a Realm application remains a subject of debate. One option would be to explicitly split the host CPU code from the GPU code, supplying two different `CodeDescriptor`s and the GPU one to use a PTX-specific format (and perhaps allowing the use of pre-compiled function pointers for the host CPU side). Another option might be to define another style of task for CUDA-capable `Processor` kinds in which a PTX kernel can be executed directly, without the necessity of host CPU code for the task.

9.4 Other Code Translators

Many other code translators are planned or envisioned. OpenCL[38] is a programming language similar to CUDA, and uses its own LLVM-inspired internal representation, SPIR[39]. A code translator for OpenCL would likely be structurally similar to the CUDA code translator.

Increasingly, the “top level” task of scientific applications is written in scripting languages such as Python or Lua. Dynamic code generation is very common in such languages, and the Realm dynamic code generation path can be used to implement `eval` (or `loadstring`, ...) as an asynchronous and distributed operation. Additionally, numerous tools exist in these languages for generating efficient machine code (for CPUs, GPUs, or both) from the source scripting language[20, 41]. Their incorporation into Realm's dynamic code generation flow could greatly simplify the porting of these applications to Realm.

There are also a number of code generation tools that are used by some applications for generating machine code for the host CPU. These include JIT engine in Chrome V8[34] or the Pin framework from Intel[45]. Pin is particularly interesting because it provides the ability to modify existing machine code and can be used to add custom instrumentation to tasks that have been ahead-of-time compiled or generated through an unrelated flow. A library module using Pin and linked in with Realm would be able to provide a translator from `FunctionPointerImplementation` to some sort of `InstrumentedImplementation` and add its own profiling measurements for application use. This modular internal structure of Realm is discussed in the next chapter.

Finally, note that the asynchronous nature of Realm allows the consideration of code generation techniques much more aggressive than are used in traditional latency-sensitive JIT engines. As long as other work can be done in the meantime, expensive optimization passes can be used for a given task if the payoff is expected to be worth it. Such an effort can even be speculative. Once an initial implementation of a task is available, a search for a better implementation using a tool such as STOKE[55] can proceed concurrently with use of the initial version. Should an improvement be found, it can be registered in place of the original and future task launches will automatically use the better version.

Chapter 10

Extensibility

A runtime system that intends to provide abstractions for portability must be easily extended to support new types of execution or storage, new ways of moving data through a distributed memory hierarchy, and new paths for loading or generating code for tasks. Realm uses a modular internal structure in which even many of the basic capabilities (e.g. host CPU `Processor`s and main system memory) are provided by *modules*.

Figure 10.1 shows the internal structure of the Realm runtime. We will discuss the interface for modules momentarily, but we first cover the components that make up the foundation of Realm. The machine model was discussed in detail in Chapter 7. Its interactions with other Realm components consist primarily of being notified of the addition or removal of resources or changes to affinities between them. Similarly, the role of the `Runtime` object in application setup and cleanup was covered in Chapter 5. During the actual execution of the application, the main job of the `Runtime` is to maintain tables of all existing objects (resources, instance, generational events, and reservations) and map the portable handles supplied by the application for those objects into pointers to the corresponding implementation objects (in the current process). The remaining components are not application-visible and have, until now, escaped description.

10.1 Events and Scheduling

Events and scheduling form the heart of the Realm runtime. The event subsystem manages the generational event, barrier, and reservation objects in the local process and sends and receives the network messages necessary to keep them consistent with the corresponding objects in other processes. It also keep a table of all operations known to the local process, which enables deadlock detection and enumeration of “lost” operations if a node fails.

The internal implementations of generational events provide a `trigger` method that is called by a completing operation. The event subsystem defines an abstract `EventWaiter` class that other

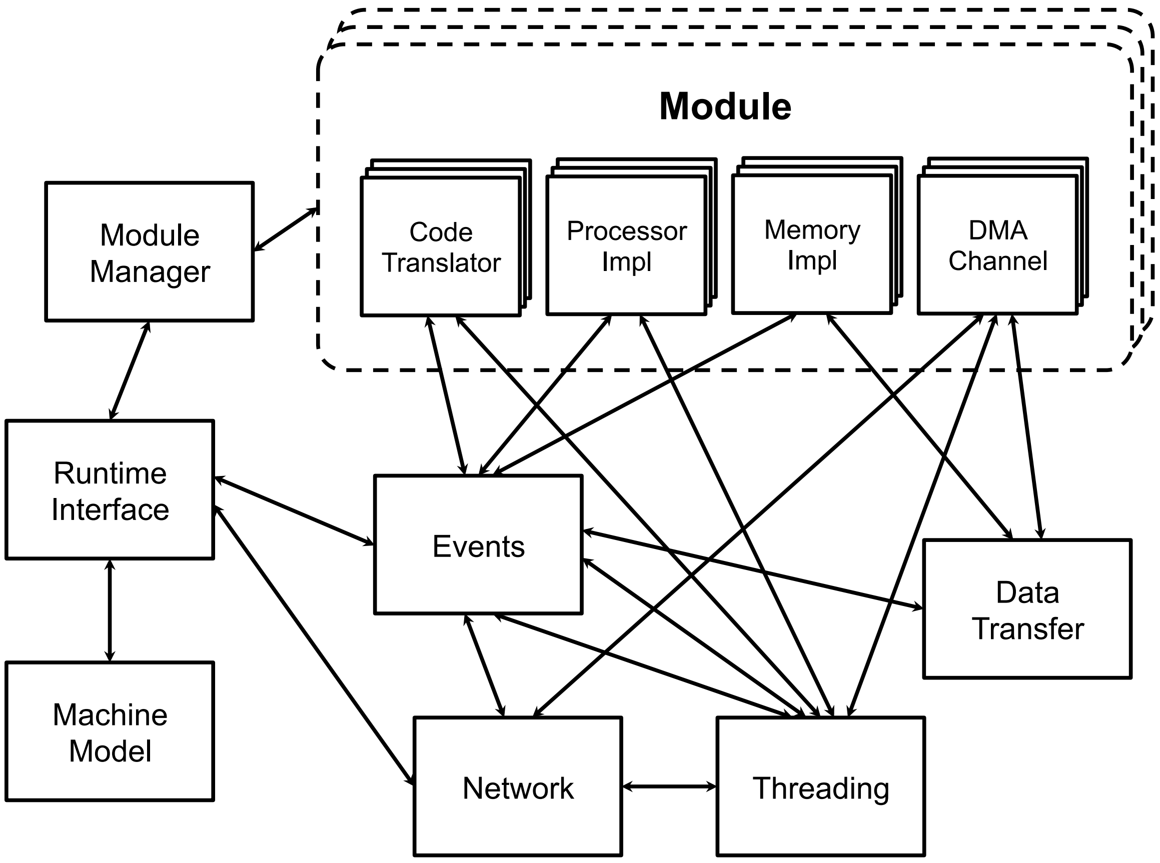


Figure 10.1: Modular Internal Structure of Realm

components use to receive notifications when a particular generation of a generational event or barrier is resolved. Resolution is either a successful triggering of the event or a poisoning, and is indicated by a boolean parameter in the `event_triggered` method that must be implemented in any concrete subclass of `EventWaiter`. When the parameter indicates a poisoning case, it is the responsibility of the `event_triggered` method to skip the corresponding operation and propagate the poison to the operation's finish event. (Subclasses must also implement `print` and `get_finish_event` methods, which are used for the detection and printing of dependency cycles, as discussed in Section 5.11.)

10.2 Threading

The threading subsystem in Realm solves two related problems. It provides a system-independent interface for `KernelThreads` and, if supported, `UserThreads`. A `KernelThread` is a thread that is scheduled and context switched by the kernel. In all systems currently supported by Realm, kernel thread support is provided by the standard POSIX threading API (pthreads).

In contrast, a `UserThread` represents an execution context that is managed entirely in user space. One or more `UserThread`s can be created on top of a host `KernelThread` and the switches between the host and user thread(s) are requested explicitly with the `Thread::user_switch` method. A user-level context switch can be significantly faster that one that crosses into kernel space (185 ns vs. 1060 ns on the Legion development cluster), and will not be preempted (as long as core assignment is done correctly), but user threads must be used with care. If a user thread makes a system call that causes a wait in the kernel (e.g. file I/O), that same inability to preempt the thread prevents the execution of other work on the core while the first thread waits.

The second problem solved by the threading subsystem is the assignment of threads to physical execution resources in the system. By default, the operating system choose from all available CPU cores when deciding where to run a new or awakened thread, suspending whatever thread is running on that core. As it has no information about the application, it uses heuristics, which are often exactly contrary to Realm's preferences. For example, a thread that has had its "fair share" of CPU cycles (e.g. one executing a long-running, performance critical application task) will almost always be switched out in favor of a thread that has just woken up from performing a lower-priority background operation.

However, most operating systems (macOS is a notable exception) provide a way to restrict the set of cores on which a thread may run, and therefore which other threads it competes with for execution resources. In particular, if the the ability to run on a particular core is restricted to a single thread (and that thread is similarly limited to run only on that core) it will never be switched out by another thread in the same application. The job management systems of most supercomputers also guarantee that at most one application has permission to run on a given core, eliminating all application interference. The operating system's background tasks often remain a source of interference, although the asynchronous execution model provided by Realm allows Legion to handle it better than traditional bulk-synchronous models[56].

The process of assigning threads to cores consists of three steps. The first step builds a *core map*, a list of the execution cores available to the Realm process in each NUMA domain and whether any share physical datapaths. Processor architecture can get very complicated, but the core map simplifies the matter, looking at three main resources: the arithmetic logic unit (ALU) that performs integer and logic operations, the floating-point unit (FPU) that performs floating point math operations, and the load/store unit that handles access to memory. Some processor architectures have dedicated units of each type for each core, but others are more complicated. On Intel CPUs with HyperThreading enabled, each HyperThread appears as a core to the operating system, but pairs share a single ALU, FPU, and load/store unit. On the AMD Bulldozer CPU used in Titan, each core has its own ALU and load/store unit but shares an FPU with one other core. On Linux, the core map can be discovered via the `sysfs` pseudo-filesystem, but a more portable approach uses the `hwloc` library if it is available[13].

	Exclusive	Shared	Minimal	None
Exclusive	N	N	Y	Y
Shared	N	Y	Y	Y
Minimal	Y	Y	Y	Y
None	Y	Y	Y	Y

Figure 10.2: Core Reservation Compatibility Matrix

The second step is to determine how many threads are needed and their willingness to share execution resources. Realm's modular structure makes it hard to know the complete list up front, as each module needs to have a chance to examine the system state and command-line parameters. A greedy core assignment is infeasible (there is no way to know how many more threads to come will demand their own cores), so all requests are accumulated before any decisions are made. The requests come in the form of `CoreReservation` objects that are created by subsystems or modules that intend to launch kernel threads. Each `CoreReservation` requests a certain number of cores, whether they need to come from a particular NUMA domain, and what level of use they intend to make of each of the generalized ALU, FPU, and load/store datapaths in the core. The usage may be `EXCLUSIVE` if the thread is expected to make heavy use of the datapath and wants to avoid interference, `SHARED` if significant use will be made but interference is acceptable, `MINIMAL` if such little use will be made that it is unlikely to cause noticeable interference, and finally `NONE` if absolutely no use will be made. (It is virtually impossible to avoid the use of ALU or load/store instructions, so it is expected that this would only be used for the FPU datapath.) Eventually, the `CoreReservation` objects will be filled in with the allowed set of cores for that reservation, and these objects are passed to the `KernelThread` construction routines to pass the allowed set to the operating system. To avoid having to perform two passes over the modules, the interface allows a `KernelThread` to be constructed before the `CoreReservation` is filled in. In this case, the `KernelThread` object is created immediately, but the actual thread creation is deferred until the allowed set of cores is known.

Once all `CoreReservation` objects have been created (i.e. all subsystems and all modules have had a chance to make their requests), the threading subsystem attempts to find a satisfying assignment of cores and reservations. The current implementation uses a heuristic of sorting the reservations in order of decreasing restrictiveness, favoring reservations that require exclusive access for any datapath over those that do not, and then further splitting the first group to favor those that request cores from a specific NUMA domain over those that have no preference. Assignments are then made iteratively, with each reservation initially starting out with either the set of all cores in the core map or just the cores in a particular NUMA domain, if the reservation has requested it. Then for each datapath, the new reservation's compatibility with each previously assigned reservation is determined based on Figure 10.2. If two reservations are incompatible, the set of cores used by the previous reservation and any cores that share the datapath in question with cores in that set are removed from the new reservation's potential set. After all previous reservations have been considered, the size of the remaining set is compared against the number of cores requested. If it is inadequate, the whole assignment process is considered to have failed — there is no backtracking. If sufficient cores are available, the exact assignment made depends on whether the reservation had exclusivity demands. If it did, it is given exactly then number of cores requested — any more is likely to be a waste. A reservation with no exclusivity demands is simpler, being given permission to use any cores that remain in the allowed set. This algorithm assumes that any satisfying assignment of cores is equally good, and can be easily broken (i.e. fails even though a satisfying assignment exists) for contrived inputs, but it produces the "right" answer for all hardware and software configurations that have come up in practice.

10.3 Data Transfer

The DMA subsystem in Realm handles the execution of copy, reduce, and fill operations. Although the actual movement of bits between storage locations in the memory hierarchy is handled by the DMA channels provided by modules, there is a considerable amount of common effort that must be performed. When a new copy or reduction request is made, the first step is often for the Realm process to fetch information about the instances involved from the nodes on which they were created. These requests, if necessary, are performed concurrently with waiting for the transfer's precondition to be satisfied.

Once that precondition is satisfied and all instance metadata is available, the DMA subsystem can split a transfer request into subrequests for each unique pair of source and destination `Memorys`. For each such pair, the available DMA channels are examined to find one that will provide a `MemPairCopier` object that encapsulates the functionality required to copy (or reduce) data. Finally, if a copy involves multiple fields, the DMA system determines whether to perform the copy for one field at a time (i.e. an SOA order) or whether to interleave the fields (SOA), with the goal being to make the smallest number of distinct requests to the copier object. Fill operations follow a similar path, but are simplified as they do not have source instances and memories to consider.

The DMA subsystem also tracks the completion of transfer requests. It requests the insertion of *fences* into each of the DMA channels it uses, and triggers the transfer's completion event once all fences have reached their destinations.

10.4 Network

The network subsystem handles requests and responses that go between Realm processes as part of the distributed execution. These requests are either *active messages*, which must be processed by a message handler on the receiving node, and *direct memory transfers* (often called Direct Memory

Access) that directly read or write contents of the memory in another process without any involvement from the process itself. The core capabilities are provided by the GASNet communication library, but there are several rough edges that must be smoothed out by the network subsystem.

The first relates to the library calls that send and handle active messages. They are designed for maximum compatibility with C calling conventions, which makes them very clunky for C++ code. The parameters for an active message must be chopped up into 32-bit chunks which are passed as separate arguments and must be reassembled in the handler. To keep this clunkiness from obfuscating other Realm code that works with messages, the network subsystem provides templated wrappers that allow the active message arguments to be any trivially copyable structure and use inlining to avoid making an extra copy of the arguments in most cases.

A second issue has to do with the way GASNet makes use of threads. Many communication libraries create their own *progress threads* that run in the background, polling the network interface and performing asynchronous operations requested by the application. Such background threads are problematic for Realm, as they often cannot be managed, or in some cases even understood, by the threading subsystem. This can cause strong interference with application tasks when the network is heavily utilized. GASNet does count on background activities but does not create a progress thread. Instead, it temporarily "hijacks" threads that make calls into the GASNet library and uses them to perform background activities in addition to whatever was required for the library call itself. These activities can include the handling of an arbitrary number of incoming active messages, and the potential (and unpredictable) delay to application threads is undesirable.

As a result, the networking subsystem creates what amounts to progress threads to send and receive active messages, but does so using `CoreReservation`s so that interference with performance-critical threads is avoided. In a bulk-synchronous model, every process tends to be sending and receiving its messages at about the same time, but the traffic patterns are often more asymmetric or skewed in time in an asynchronous runtime. In an effort to further avoid coupling between sending and receiving active messages, Realm uses two classes of progress threads: one for sending messages and one for receiving. The handlers that are registered with GASNet are always performed by the sending thread, so they are changed to only be responsible for putting the received message in a queue. The queue is read by the receiving thread(s), which call the "real" handler code.

The final impedance matching performed by the network subsystem is related to memory management. To be efficiently offloaded to the network interface hardware, any active message larger than 128 bytes must have its payload placed in a pool of memory that has been registered with GASNet, and must provide an address in similarly-registered memory in the target process to which the payload should be copied. These pools must be shared between all possible communication peers, ideally in a way that prevents *head-of-line blocking*, in which messages to one peer are delayed because congestion prevents an older message to a different peer from being sent.

An open question remains with respect to the management of the pool used for sending messages.

It is not uncommon for Realm applications to have situations where a large number of operations are preconditioned on the same `Event`. When that event triggers, all of these operations become ready and can initiate network requests much faster than the network can actually process them. If a burst of network requests would overflow that source data pool, the network subsystem must either stall requests, reject them, or spill their payloads into a temporary buffer. Stalling the sender of a message re-introduces all the problems of implicit dependencies. The rejection of messages is also problematic in an asynchronous environment, as it's not clear to the requestor when to retry the request. Finally, spilling only reduces the problem, as a sufficiently large burst of messages will exhaust the entire memory of a node. (This is currently being observed for some Legion applications when running with over 1000 nodes.) At time of writing, the Realm implementation uses a moderate amount of spill space to weather small bursts and stalls when that is exhausted, but other options are being explored, as are ways to use the source data pool more efficiently. One promising approach is to provide "soft" back-pressure that does not stall or reject messages, but instead gives the application feedback that it is nearing the pool's capacity and it might wish to prioritize other work.

10.5 Module Interface

The last subsystem that makes up the foundation of Realm is the module interface that is responsible for loading the modules that provide the necessary implementations for processor and memory resources, memory-specific functionality for moving data, and/or code translators for dynamic code generation.

The module loader discovers modules either by virtue of them being statically linked into the main application executable or by dynamically loading shared objects requested by command line parameters or environment variables. To be discovered, a module must define a subclass of the abstract class `Module`[1] and use a special `REGISTER_MODULE` macro that defines a module registration object appropriate for the linkage type. The `Module` interface (shown in Figure 10.3) is simple enough that we can describe all the methods here. The `get_name` is self-explanatory and used primarily for diagnostic messages.

The static method `create_module` is the one first called by the module loader. It receives a pointer to the main `RuntimeImpl` object (not the opaque version seen by the application) as well as the list of command-line arguments. The job of `create_module` is to analyze the system state and the command-line arguments and then create and return the actual module object. This creation is optional — the `create_module` may choose to return a null pointer instead. For example, the CUDA module does not load if no GPUs are found in a given system. The *cmdline* argument to the `create_module` method is intentionally mutable. A module is expected to consume any command-line arguments it understands, allowing Realm to warn the user about any unrecognized arguments.

[1]Recall that all Realm classes are placed in the `Realm` namespace, avoiding any conflicting definitions using such a common name.

```
class Module {
  const std::string& get_name(void) const;

  // a 'pure virtual' static method - must be defined in subclasses
  // static Module *create_module(RuntimeImpl *runtime, std::vector<std::string>& cmdline);

  virtual void initialize(RuntimeImpl *runtime);

  virtual void create_memories(RuntimeImpl *runtime);

  virtual void create_processors(RuntimeImpl *runtime);

  virtual void create_dma_channels(RuntimeImpl *runtime);

  virtual void create_code_translators(RuntimeImpl *runtime);

  virtual void cleanup(void);
};
```

Figure 10.3: Realm Module Interface

A module is also encouraged to use a unique prefix for command-line arguments to avoid name collisions.

The module interface adds any non-null pointers returned by `create_module` calls to a list of active modules in the runtime. The first use of the list is to iterate over it, calling the `initialize` method on each one. By deferring initialization until after all modules have been loaded, any interactions between them (or the network subsystem) can be understood.

10.5.1 Memory Implementations

Once all module and subsystem initialization has completed, each module is invited to create any memory resources it plans to provide as part of its `create_memories` method. The functionality for a memory resource is encapsulated in a subclass of `MemoryImpl`, which provides the ability to create and destroy instances in that memory. This allocation must support the deadlock avoidance algorithm described in Section 5.5.1, but in most cases this is simply a matter of inheriting the default allocation methods from `MemoryImpl`. A memory implementation must also either provide access to the raw contents of the memory for the standard `RegionInstanceImpl` or generate custom instance implementation objects that perform accesses directly. A unique `Memory` handle is obtained for each implementation through a call to `RuntimeImpl::next_local_memory_id` and the implementation is then added to the runtime with `RuntimeImpl::add_memory`.

10.5.2 Processor Implementations

After all memory resources are created, the `create_processors` method is called on each module. Similar to the creation of memories, a module may define subclasses of `ProcessorImpl` that implement a `spawn_task` method that is nearly identical to the application-visible `Processor::spawn` one. The only difference is that a completion event has been created for the task and is passed to the processor implementation. The implementation is responsible for using `EventWaiter`s to watch for resolution of preconditions and for maintaining some sort of scheduler for tasks whose preconditions have successfully triggered.

A helper class `LocalTaskProcessor` is provided that uses a priority queue for ready tasks and schedules either multiple `UserThread`s or multiple `KernelThread`s to execute tasks to completion if possible, but context switch if they explicitly wait on an unresolved event. The "core" Realm module uses these to provide the standard *application*, *utility*, and *IO* processor kinds, which differ only in the reservations made for their threads (application processor threads demand exclusivity, while the others share) and whether they take advantage of user-level context switching if it is available (IO threads do not, as they will frequently wait in system calls).

As part of the creation of processors, a module scans the list of memories that were created in the previous step to determine which ones can be accessed by each processor resource. The `RuntimeImpl::add_proc_mem_affinity` method is used to add these affinities to the machine model.

10.5.3 DMA Channels

The next step allows each module to add DMA channels, which provide the ability to move data between a pair of memories. Whereas a module's `create_processors` step needs to be after every module has performed `create_memories` so that the right processor-memory affinities can be reliably found, the `create_dma_channels` step is ordered after `create_processors` strictly for simplicity.

The instances of `DMAChannel` that are added to the runtime as part of this step are not directly visible to the application, but their presence is implied by the memory-memory affinities that are also added to the runtime as part of this step. As described in Section 10.3 above, a DMA channel exposes its copying (or reducing) ability through the instances of `MemPairCopier` it provides to the DMA subsystem.

10.5.4 Code Translators

The final step in which a module extends the existing functionality of the Realm runtime is by adding any code translators in the `create_code_translators` step. The functionality of a particular `CodeTranslator` is again accessed through a very simple API — responding to a `can_translate` method to indicate if it can translate a `CodeImplementation` of one type into one of another particular type, and then a `translate` method that actually initiates the translation if desired by the runtime. As discussed in Chapter 9, a code translator will almost always create its own thread to perform these translations asynchronously, providing a translation result to the supplied *callback* once it is ready.

10.5.5 Application Interaction

Nearly all the capabilities provided by a Realm module are used by the runtime internals rather than the application itself. However, a few constructs are used directly by the application. Two items that have already come up in earlier discussions about extensibility are profiling measurements or code implementation types that are specific to the kind of processor, memory, or code translators provided by a module. However, a module may also provide custom accessor types for more efficient access to instances.

10.5.6 Inter-Module Dependencies

The Realm module interface is designed with the expectation that modules are largely independent of each other. (The entanglement between the foundational components in Figure 10.1 suggests why they are part of the foundation rather than modularized.) However, carefully-considered dependencies between modules can be preferable to the duplication of code and effort that might result from strict enforcement of the independence of modules. For example, as described in Section 9.3, the code translator for CUDA uses LLVM for some of its translation steps, and uses the code (and worker threads) from the LLVM module rather than creating its own threads and LLVM contexts.

Entanglement between modules causes two complications, but both are fairly easily addressed. (One could argue that it causes a third complication in the form of increased maintenance cost of the software, but we do not consider that here.) The first is the requirement that the loading of a module also load any dependencies, and that they be initialized in the correct order. The dependent module necessarily uses symbol names contained in its dependencies, so the inclusion is automatic in the case of static linking and ensured with a linker flag (e.g. `-z defs` for GNU ld) for dynamic modules. The correct initialization order is obtained in both cases by including the name(s) of the dependency in the dependent module's registration macro, ensuring they are registered (and therefore initialized) in the correct order.

The second complication is that a module that intends to use runtime resources (i.e. not just code) of a second module must handle the case in which that second module is not created. The dependency's `create_module` is guaranteed to have been already called, but recall that it may choose to return a null pointer instead of creating the module. In such a case, the first module must find a way to adapt, or it can choose not to create a module either. In the example given above, the CUDA module is still created if the LLVM module is not, but the CUDA code translator is disabled. The processors, memories, and DMA channels all remain usable in this case.

Chapter 11

Case Study: S3D

It is difficult to quantify claims of improvements to portability or programmer productivity. Benchmarks and mini-apps are often simple enough that the best way to implement them for a given machine is obvious to somebody familiar with the architecture. Similarly, the code is usually small enough that it can seem quicker for a good programmer to just write a new version for each machine in a known model than to learn a new programming style. However, assumptions that hold true for benchmarks and mini-apps may not apply to real HPC applications. Not only are most applications too complicated to keep a full model in one's head, and too long to consider rewriting for each new architecture, the authors of most HPC applications are scientists in some other domain (e.g. chemistry, nuclear physics, or biology), not experts in programming and computer architecture.

In this chapter, we will tell the story of porting a real HPC application to Legion, its tuning for several different systems, and some extensions that were made to it. The effort required the use of several tools in addition to Realm, most notably the Legion, so while credit for these results is most certainly shared, we will argue that this effort would not have been so successful, or likely even possible, without Realm's support for composable asynchrony and performance portability.

11.1 S3D

The application in question is S3D, which simulates combustion of hydrocarbon fuels in turbulent environments. It performs *direct numerical simulation* (DNS) of all of the chemical and physical phenomena involved in combustion. Whereas other simulation techniques use approximations for smaller physical structures (e.g. eddies in the turbulent flow), a DNS application is fully resolved, explicitly modeling the behavior at the smallest physical and time scales. A direct numerical simulation approach is very expensive in terms of computational power required, easily filling the largest supercomputers in the world for days at a time.

S3D is a very old application that has seen several generations of users, maintainers, and developers. The original version was written in Fortran and used MPI for all parallelism, following the "rank per core" approach described in Chapter 1. In many ways, S3D is an ideal bulk-synchronous application. It performs a very regular sequence of computations on a regular grid, using nearest neighbor communication patterns.

S3D is designed to work with different chemical *mechanisms*, effectively the list of atoms and molecules that will be tracked and the chemical reactions that convert between them. The function that computes the rates of these chemical reactions represents a large fraction of the computational cost in S3D. However, the chemistry is embedded in an explicit solver for the Navier-Stokes equations, which consists of a very large number of small kernels that model the *transport* of the various chemical species through the physical volume represented by the grid. These transport kernels have a fixed computational intensity, and as the latency gap grew, they became the bottleneck for scalability on large systems. Initially, enough of the latency could be hidden by the use of non-blocking MPI transfers (i.e. implicit dependencies), but it was clear that would be insufficient on a system of the size of Titan.

The second issue faced by S3D with respect to Titan was its heterogeneity. Recall that the bulk of Titan's computational power is in its CUDA-capable GPU, not in the CPU cores where the Fortran application runs. To obtain the performance promised by Titan, most of the S3D computational kernels had to be moved over to the GPU. Additionally, the introduction of a single GPU per node effectively broke the rank per core model that had been used until then. As these same challenges were faced by many other HPC application, S3D was chosen to be one of the six applications that were used to judge Titan's application readiness.

The strategy chosen for porting S3D to Titan was to switch from the rank per core model to the hybrid approach discussed in Chapter 3, using one process per node. The processes on different nodes still used MPI to communicate, but the scheduling of and communication between the heterogeneous processors within a Titan node was handled by OpenACC[53], a directives-based programming model similar to OpenMP that supports the offloading of sections of code to a GPU accelerator.

The ability for directives to be added to otherwise-unchanged application code is a major selling point of directives-based models. Unfortunately, the porting of S3D to OpenACC required extensive manual code transformations as described in [43], which reported a speedup of approximately 2.3X compared to the original MPI version that did not use the GPU at all. The code transformations were performed for only two mechanisms, and involved major changes to the mechanism-agnostic code as well, a major reason that the code was unable to be merged back into the main S3D source code. Hybridization naturally reduces the number of MPI ranks, and the OpenACC version of S3D included additional optimizations such as the interior/boundary split described in Chapter 2 to better tolerate the network latencies on Titan. Experiments run on Jaguar (Titan's predecessor at

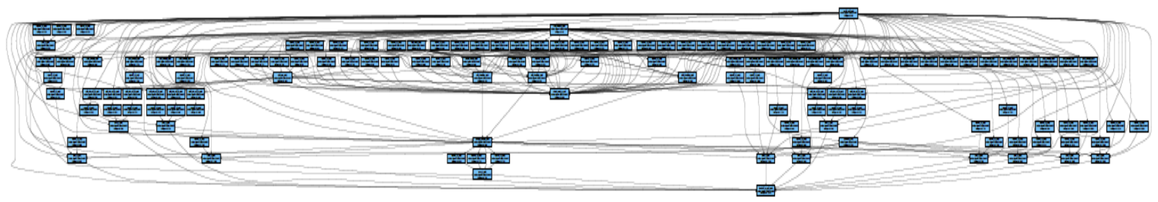


Figure 11.1: Task Graph for a Single Iteration of S3D's Main Loop

Oak Ridge) showed significantly better latency tolerance at 7200 nodes (86400 cores), but no scaling data was reported for Titan itself.

Around this time, the Legion project was looking for a "real application" to port as a way of testing whether the initially promising results reported for mini-apps in Legion[5] would hold up at scale. Back-of-the-envelope math suggested that significantly more performance and latency tolerance could be obtained from S3D on Titan, but only by exposing the ample task-parallelism within S3D.

The initial port of S3D to Legion required about two person-months and focused on the top-level structure of the application, getting partitioning and task launches correct. The use of Fortran code was not possible, so the effort initially used simple debugger-friendly transliterations of the Fortran computational loops into C++ code. Based on the design of Legion and Realm, the team was confident that vectorized CPU and GPU kernels could be swapped in later without impacting overall functional correctness or creating communication bottlenecks. In particular, the ability to use Legion's mapping interface and Realm's portability abstractions to move one leaf kernel at a time from the CPU to GPU allowed the optimized version of each kernel to be tested in isolation, greatly reducing the debugging time necessary.

Code for the important mechanism-specific kernels was produced using Singe[4], which treated the input files that describe the mechanism as a simple domain-specific language and generated optimized code for both CPU and GPU execution. Although performed as an ahead-of-time compilation process in this case, a just-in-time version of Singe that uses Realm's dynamic code generation as well as profiling (for automatic tuning of the code) capabilities is being explored. This would allow Singe to take advantage of additional input-dependent properties (e.g. the exact size of the local subgrid for loop unrolling purposes) to produce even better code.

Figure 11.1 shows a portion of the *logical task graph* for the Legion version of S3D. (The logical task graph is lowered into the Realm operation graph, including all data movement and any necessary synchronization, during Legion's mapping process. This lowering can only further increase the width of the graph.) This portion only includes one iteration on one node, and uses the simplest chemical mechanism with only 9 species, but the width of the graph makes clear the amount of task-parallelism that was hiding in the original Fortran version. It also provides a sense of how many code permutations might have to be considered to achieve the optimal scheduling for a given

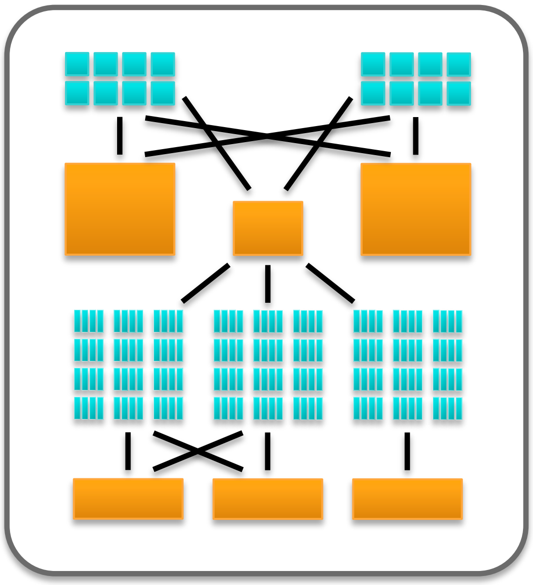


Figure 11.2: System Architecture of Keeneland

system in a model based on implicit dependencies. Legion's data model automatically extracts this parallelism for Realm, with significant benefits that we will see in the rest of this chapter.

11.2 Keeneland

Initial development work on the Legion port of S3D was done on a small development cluster, but it was soon time to start testing on larger systems. Rather than moving immediately to Titan, we first did significant testing on a smaller system called Keeneland. Keeneland was an experimental system designed specifically to anticipate the heterogeneity likely to be seen in future systems[64], giving the authors of applications, compilers, runtimes, and tools a chance to assess their readiness for Titan and machines like it.

Figure 11.2 shows the architecture of Keeneland. Each node contained two CPU sockets and three GPUs. It was a NUMA system — each CPU had its own attached memory and the ability to access the other's at reduced performance. However, the non-uniformity extended as well to the connectivity between the CPUs and GPUs. Two GPUs were on the same PCI-Express bus, allowing them to access each other's memory, but also forcing them to share a common path to the CPUs' memories. The third GPU was on a separate PCI-Express bus, and could only communicate with the first two by going through the special "zero copy" portion of system memory.

Based on the Keeneland system architecture, the correct mapping to use seemed obvious. The most math-intensive kernels should be spread across the three GPUs, each handling one third of the

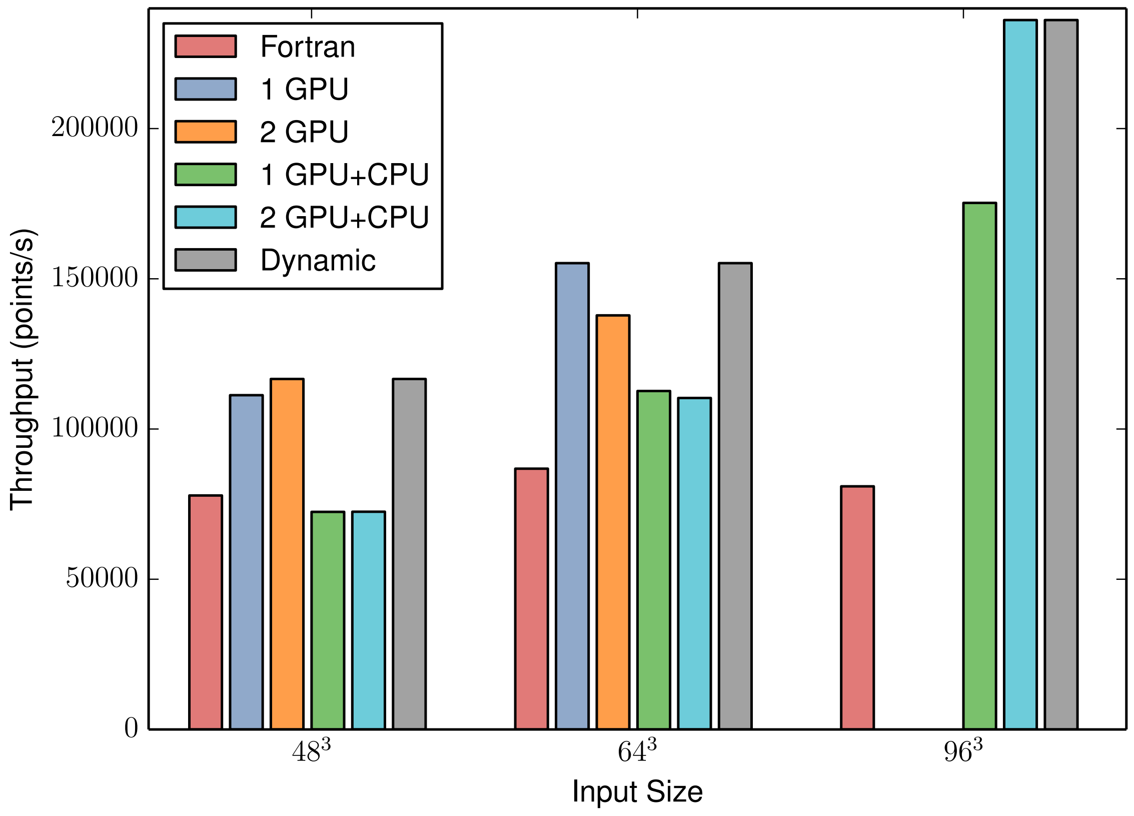


Figure 11.3: Mapping Alternatives for S3D on Keeneland

local subgrid, and the CPUs should carry their share of the load as well. The necessary changes to the application's mapping code were straight-forward due to the Realm machine interface. However, the results were surprising. The "obvious" mapping for such an architecture performed terribly. Using the output obtained from Realm's profiling interface, it was clear that the bottleneck was due to copies that were crossing the PCI-Express buses in the system. Spreading the work across all the CPUs and GPUs simply resulted in too much communication.

Communication could be reduced by using fewer of the CPUs and/or GPUs, but the correct subset to use was unclear. However, the effort required to try the various permutations in Legion was minimal and we simply tried them all, for a variety of different grid sizes. The results can be seen in Figure 11.3. For the smallest grid size, it was better to keep all the work on the GPUs, and splitting it across the two GPUs that could access each other's memory was slightly better than using one GPU. For the medium grid size, the increase the amount of data that had to be moved between the GPUs became a problem, and now it was slightly better to use a single GPU. However, the largest grid size exceeded the capacity of the GPU framebuffers and communication with the CPU became unavoidable. Once again the right answer was to use the two GPUs that shared a

PCI-Express bus.

The dependence of optimal mapping strategy on the grid size can be problematic in many programming models, as the size is not known until execution time. The ideal performance would be the grey "Dynamic" bar at the right of each section of Figure 11.3, picking a different policy for each grid size. Realm's portability abstractions and asynchronous scheduling allow exactly that. Legion's mapping interface is able to make dynamic policy decisions based on grid size, or any other parameter that may differ from one execution to the next. Although not currently in use, the techniques described in Chapter 8 could use the first few timesteps to try each of the choices and then use the best performing alternative for the thousands or more timesteps to follow.

The fact that the optimal mapping for S3D on Keeneland involved leaving one GPU (and sometimes two) completely idle is worth some further consideration, as it is nearly every programmer's instinct to spread the application's workload across all available processors. However, increases in heterogeneity and especially the latency gap are making it obvious that some machine architectures are simply not good fits for some applications. Other applications are able to use all of Keeneland's GPUs (e.g. the Legion circuit simulation in Figure 4.1b), and given Keeneland's experimental nature, it being a bad fit for some important applications would arguably be a positive result. In contrast, a mismatch between an important application and the architecture of a system like Titan would be seen less favorably. We will return to this question in the conclusion.

11.3 Titan

As access to Titan became available, we moved much of our testing and performance tuning to the larger machine. Apart from issues with the build system, there were no portability concerns. The same application code that ran on Keeneland ran without change on Titan as well. However, both the CPU and GPUs on Titan supported additional instructions compared to their Keeneland counterparts, and new variants of the most important leaf kernels significantly improved performance.

A major difference between Keeneland and Titan was in the optimal mapping strategy. Compared to Keeneland, Titan only had one GPU, but it was significantly faster than the GPUs on Keeneland. In contrast, there were fewer CPU cores on a Titan node. The combination of these two factors resulted in an architecture in which the vast majority of the computational horsepower was on the GPU. The bottleneck of the PCI-Express connection between the CPU and GPU led to the optimal mapping being one that performed nearly all computation on the GPU, regardless of the grid size. As this was one of the mapping strategies used for Keeneland, the changes required were minimal.

A second performance-related change was forced by the relatively slower CPUs on Titan. The dynamic analysis performed by Legion that ran on a single *utility processor* (see Section 10.5.2) became the performance limiter on Titan. Realm was easily configured to create multiple utility processors, but some additional changes were required in the Legion runtime to load-balance its analysis tasks across them.

In addition to the better single-node performance that Titan offered, the move to Titan allowed us to better test the scalability of S3D (along with Legion and Realm), and compare directly with the OpenACC version that had been developed for Titan. (It only compiled with the Cray Fortran compiler, which was not available on Keeneland.) Figure 11.4 shows the weak scaling performance of all three versions of S3D: the original Fortran version that used only the CPU, the OpenACC version designed to offload the bulk of the computation to the GPU, and the Legion version.

Our experiments replicated the 2.2X performance improvement of the OpenACC code over the Fortran, but told a different story for scaling. The OpenACC version slowed down considerably more at 1024 nodes. Despite the optimizations made to overlap more computation with communication, the latency tolerance was inadequate. This effect was exacerbated by Titan's network design, which allows network traffic from one job to interfere with traffic from another. (Recall from Chapter 3 that fat tree networks are used in large part to avoid these effects.) The effect is more easily understood by looking at the wall clock time per timestep in Figure 11.5. The absolute increase in execution time with growing node count is nearly the same for both the Fortran and OpenACC versions. The larger performance hit for the OpenACC version is then a direct result of Amdahl's Law.

The Legion version performed significantly better, providing speedups of 60-90% over the OpenACC version (and 3-4X over the original Fortran). It also scaled significantly better, despite the adverse effect of Amdahl's Law. With the more complicated chemical mechanism, the logical task graph is over six times wider than the one in Figure 11.1, and Realm used this available task parallelism to hide over 90% of the latency of network communication when going from 512 to 1024 nodes, compared to only 30% for the OpenACC version (Figure 11.5).

11.4 PRF and RCCI

The performance and scalability of the Legion version of S3D on Titan opened the door to new scientific exploration, which offered further tests of Realm's scalability and composable asynchrony. The search for an internal combustion engine that is both clean and efficient has been a long one, and one promising approach is *homogeneous charge compression ignition* (HCCI), in which the fuel and air are allowed time to mix well as in a common spark-ignition cylinder, but (as the name suggests), ignition occurs due to the heat of compression rather than a spark. This causes the mixture to *auto-ignite* throughout the cylinder volume rather than requiring the flame to *propagate* from one end of the cylinder to the other. Figure 11.6 shows an example of these auto-ignition kernels (red) within the overall volume. This allows combustion with the fuel efficiency of a diesel engine, but at much lower temperatures, reducing pollutants significantly.

The main challenge with compression ignition is controlling when the ignition occurs. Timing within an internal combustion engine is precisely choreographed, and the variability in auto-ignition

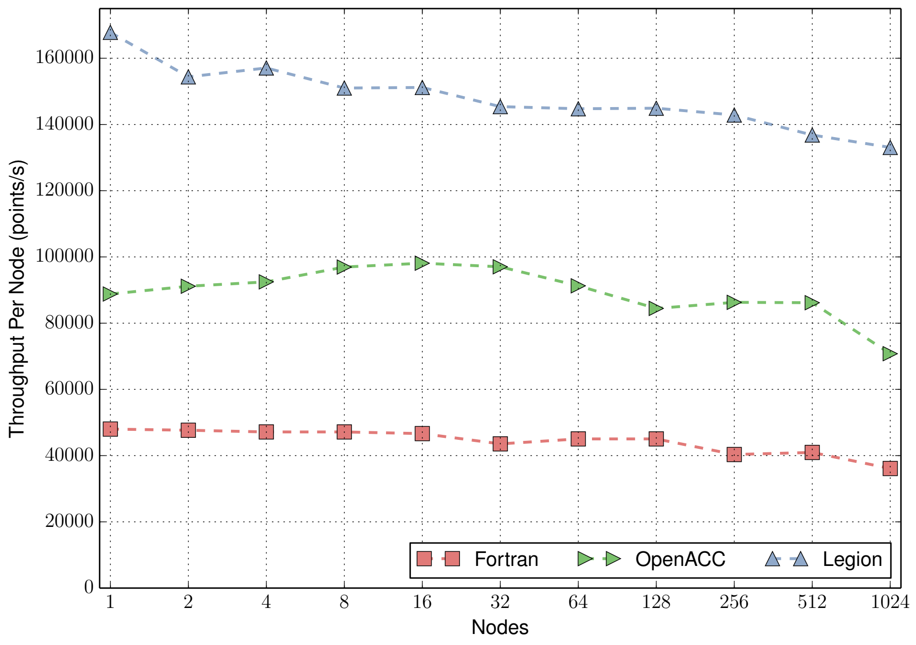


Figure 11.4: S3D Performance Comparison on Titan (heptane mechanism)

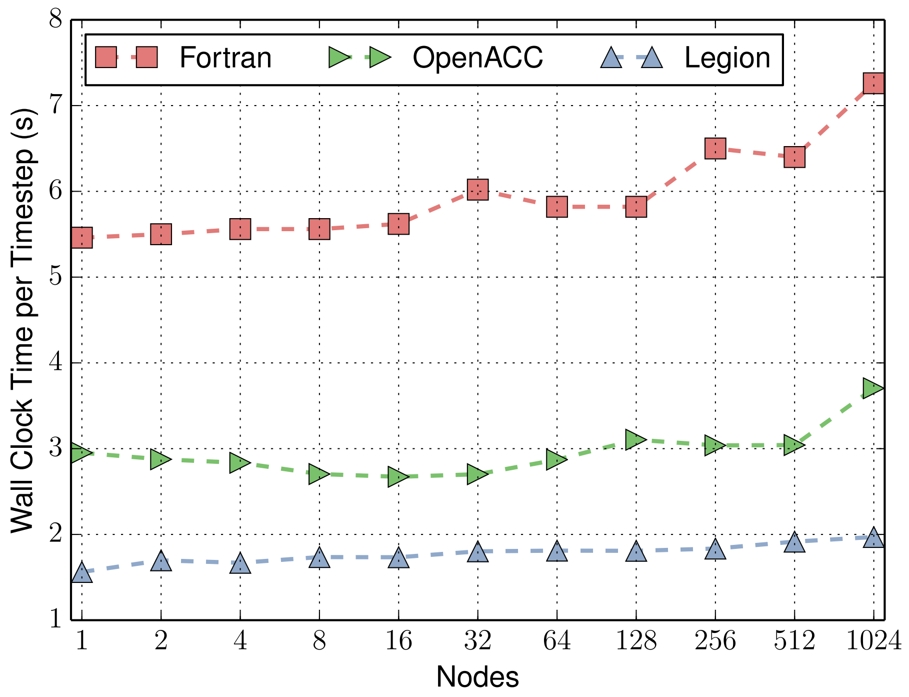


Figure 11.5: S3D Latency Hiding Comparison on Titan

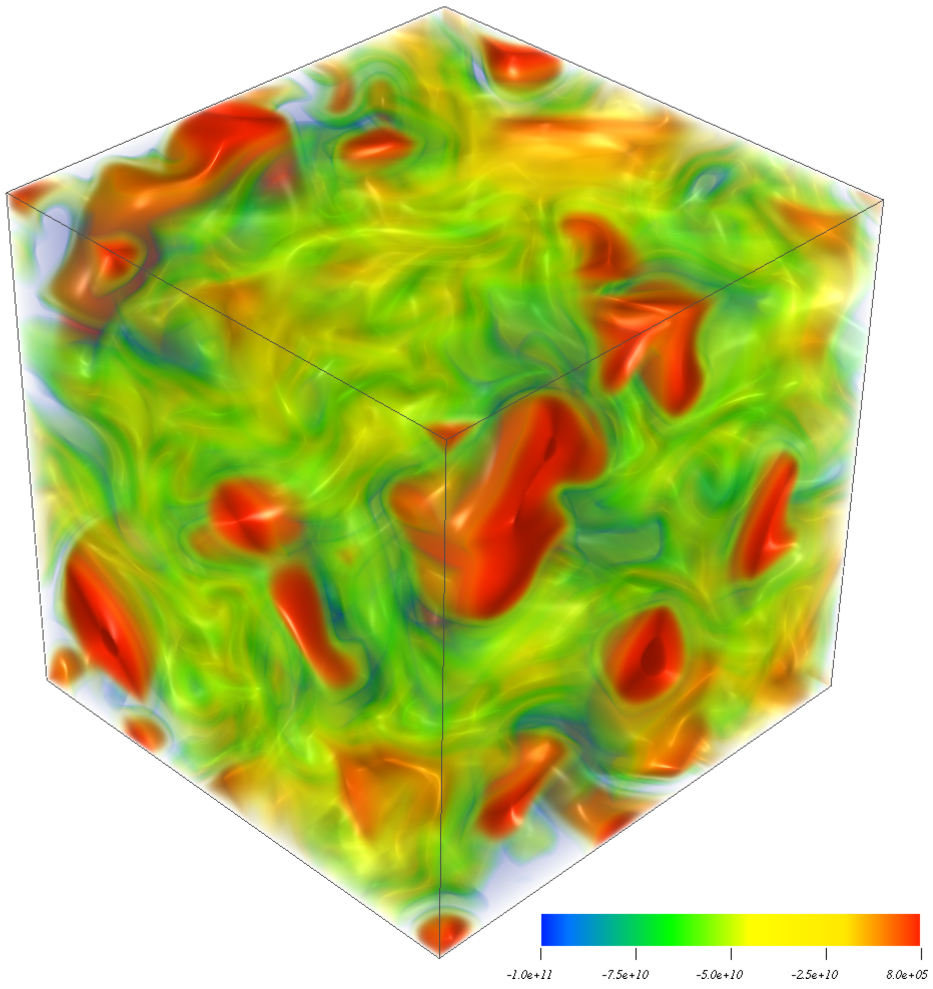


Figure 11.6: Auto-ignition kernels resulting from compression ignition

times that results from imperfect mixing of the fuel and air can decrease efficiency and increase engine wear. An area of current research is in *reactivity-controlled charge ignition* (RCCI), which tries to improve matters by using a mix of two fuels: a more reactive one (e.g. n-heptane) that ignites at a reliable time, and a less reactive one (e.g. iso-octane) that provides the bulk of the energy from the combustion.

There are many open questions with respect to RCCI, and many are related to the structure of the auto-ignition kernels, how they evolve and interact, and the sensitivities of this behavior to parameters such as the fuel mixture ratios or compression rates. Direct numerical simulation can explore many of these aspects, but DNS is incredibly expensive for RCCI. First, the simulated volume must be large enough to include multiple auto-ignition kernels and the grid resolution high enough to resolve the necessary details. Second, the simulation time must include a signification portion of the compression stroke (multiple milliseconds at least) at a temporal resolution fine enough for highly-reactive fuels (i.e. nanoseconds per timestep). Finally, the *primary reference fuel* (PRF)[40] chemistry for RCCI includes not one, but two, complicated fuels, and even a greatly reduced mechanism includes over a hundred chemical species that must be tracked per grid point and nearly one thousand reactions whose rates must be calculated multiple times per timestep.

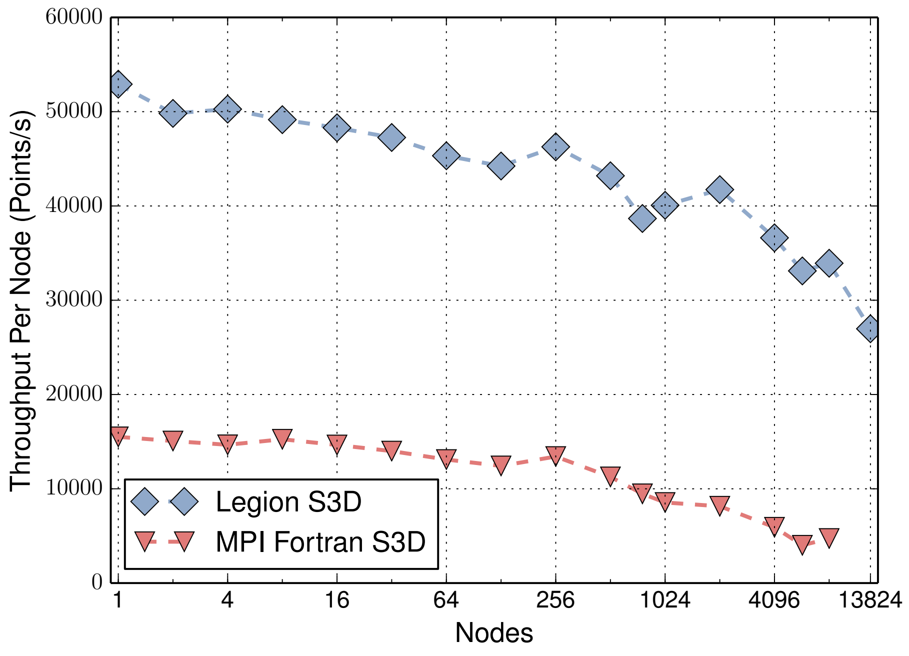
Based on extrapolations of the OpenACC version's performance for the more complicated mechanism involved, it was estimated that a full run of an RCCI simulation would require dedicated access to 75% of Titan's nodes for over a month, well outside the annual allocation provided to any domain scientist on Titan. The performance improvements of the Legion version offered to cut this to two weeks. While still very expensive, this was tractable, and the hope was that a single full run might provide a reference point from which shorter perturbation simulations could be performed, enabling much more rapid scientific exploration.

Supporting the new PRF mechanism within the Legion version of S3D was simple. Singe was used to generate code for the new leaf kernels, and a quick tuning pass determined that, unsurprisingly, the best mapping on Titan for PRF was still to perform nearly all work on the GPU. (We remind the reader that the mere ability to do such a tuning pass at negligible programmer effort is itself a contribution of Legion and Realm.)

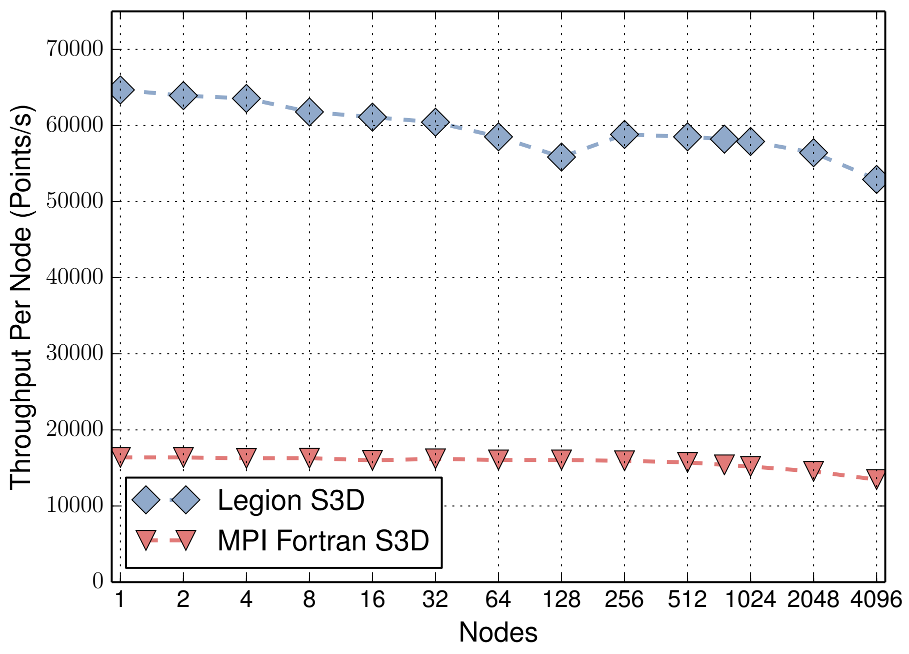
A more challenging change had to do with the modeling of the compression ignition. S3D uses a fixed grid for its simulation domain and models compression by uniformly scaling the density of every chemical species to increase the pressure within the system. Recent work had shown that this approach is greatly improved if this external source term takes into account internal sources of pressure increase (e.g. combustion)[10]. This requires the average pressure to be computed and shared with all nodes, creating an *all-to-all* communication pattern in an application that used strictly nearest-neighbor communication patterns. All-to-all communication patterns have a latency that is unavoidably logarithmic in the number of communicating peers, and the programmer's typical approach in a bulk-synchronous model is to grit his or her teeth and accept it, hoping that the implementers of the network hardware and software have done their best to reduce the constant factors. Some loss in efficiency is guaranteed as the node count increases however.

The Legion programming model provides these all-to-all communication patterns through the use of *dynamic collectives*, which are based on Realm's `Barrier`s. The composable asynchrony allows the collective to be performed concurrently with computation, as long as there are no dependencies between them. Some analysis by the scientists indicated that the simulation fidelity was not compromised if the pressure average was *lagged* by a timestep, allowing an entire timestep's computation to be independent of the collective operation. This lagging was implemented in S3D by simply transposing two lines of code (reading the result of the collective before advancing it), further improving the scalability of the Legion version of S3D with respect to the Fortran version.[1]

[1]MPI-3 introduces the ability to perform non-blocking collective operations, but requires the use of a progress thread. As discussed in Section 10.4, interference between progress threads and a computation thread with which



(a) Titan



(b) Piz Daint

Figure 11.7: PRF Mechanism Performance in S3D

Figure 11.7a shows a comparison of S3D performance, using the PRF mechanism and the HCCI pressure calibration code. Only the original Fortran and the Legion versions are compared.[2] On a single node, the Legion version was 3.9x faster than the Fortran version, by virtue of its (efficient) use of the GPU. Titan was in heavy use during this period, and network latencies were both large and variable. Although the Legion version did not scale as well as we hoped, it was still significantly better than the Fortran version, and the relative performance gap grew to 7.2x. Using the Amdahl's Law argument from the previous section, we conclude that Realm's composable asynchrony was able to hide over 93% of the network latency experienced by the Fortran version.

11.5 Piz Daint

The heavy usage of Titan was a major obstacle to getting the RCCI simulation done. With multi-day waits in the job-queue for each 24-hour execution slot, the run was going to stretch well beyond the targeted completion date. Options were limited, as this simulation was very large, and could only fit on the world's top supercomputers. Fortuitously, one such system actually had spare cycles — Piz Daint, a heterogeneous system at the Swiss National Supercomputing Centre (CSCS)[25].

This was a limited-time offer (the allocation was valid for less than a month), so we had to move quickly. The system architecture was similar to Titan's, using an identical GPU, but also including a much faster CPU, and a better network (Cray's Aries network, the successor to the Gemini network on Titan). Thanks to Realm's use of GASNet, no changes were required due to the different network. Better network latencies resulted in better scalability.

Similarly, the CPUs were functionally compatible, and the Legion version of S3D compiled and ran right away. However, the "all GPU" mapping being used on Titan would leave the extra CPU performance on the table, so we adjusted the mapping to move some of the computational kernels back to the CPU. The entire process was nearly effortless. Within a few hours of receiving the allocation, the RCCI simulation was up and running on Piz Daint, with performance and scalability shown in Figure 11.7b. Merely having a system we could run jobs on was wonderful, and the 20% performance boost from the faster CPUs was icing on the cake.

We ended up using Piz Daint virtually non-stop for three weeks. We actually ran most of the simulation twice, as we discovered that the initial conditions had been incorrect for the first run. Figure 11.8 shows a screenshot of our simulation dashboard for the second run. Each job was limited to 24 hours, with the next job starting immediately after the previous one was terminated. One night, the timeout mechanism failed and an S3D job ran for 32 hours before anybody noticed. These very long runs provided valuable data on the stability and scalability of Realm (and Legion). Several

they are intended to overlap is an open concern.

[2] Attempts were made to add the PRF mechanism to the OpenACC version. The effort was complicated significantly by the transformations that had been applied during the OpenACC port. After several days of work, the code compiled but crashed during execution, and the effort was abandoned.

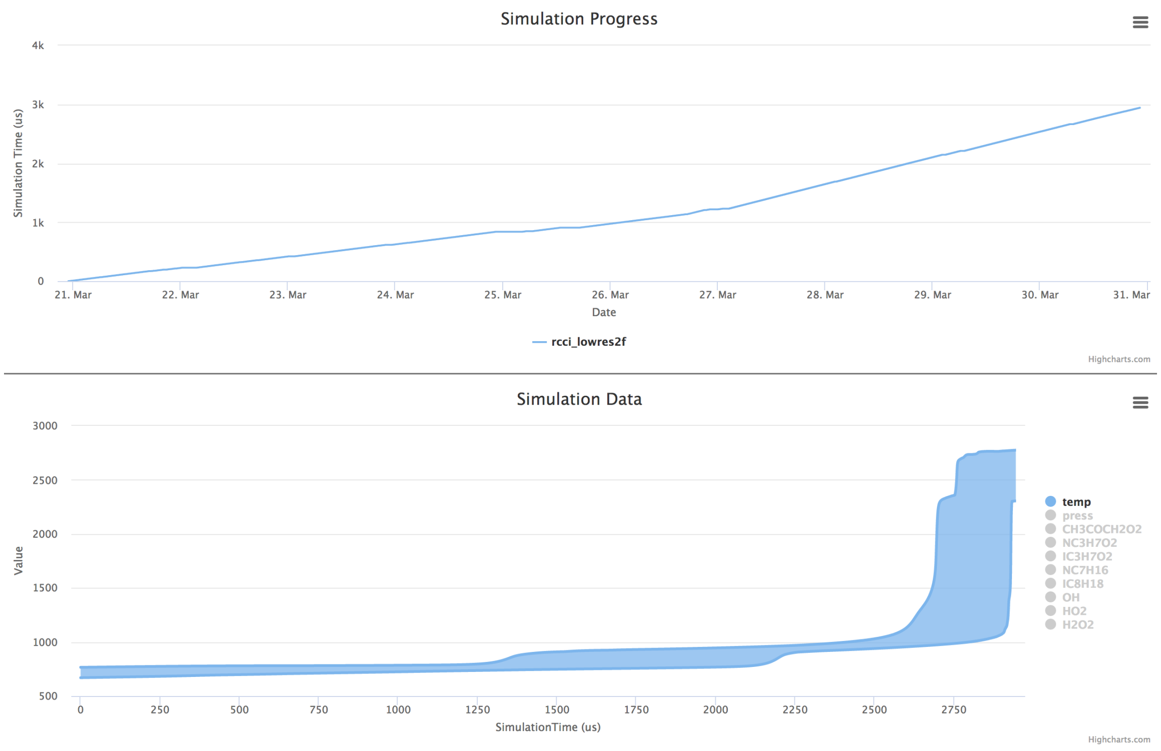


Figure 11.8: Progress of RCCI Simulation over Time

jobs failed due to hardware errors (the remaining Fortran code in the Legion version of S3D is unable to deal with the loss of a node, even if the Legion part could), but there were no failures that could be attributed to the runtime software, despite all the internal asynchrony and complexity.

Each 24 hour run of S3D involved the successful execution of over 100 million Realm operations. Realm's generational events (Chapter 6) easily handled an operation graph of this size, with no discernible increase in either memory usage or scheduling overhead observed over the course of a day-long job.

11.6 CEMA

Even with the significant performance improvements of the Legion version of S3D, the full RCCI simulation required nearly 700,000 node-hours of system time on Piz Daint. As mentioned above, one of the justifications for such a large expenditure is that the results of a full run can be used as a guide for where smaller and/or shorter additional runs can provide interesting insights. The data analysis to find such areas of interest is onerous. The simulation generated 189 GB of data *per second*, only $\frac{1}{1000}$ of which could be written to disk. (This still consumed 16 TB of disk space per day.)

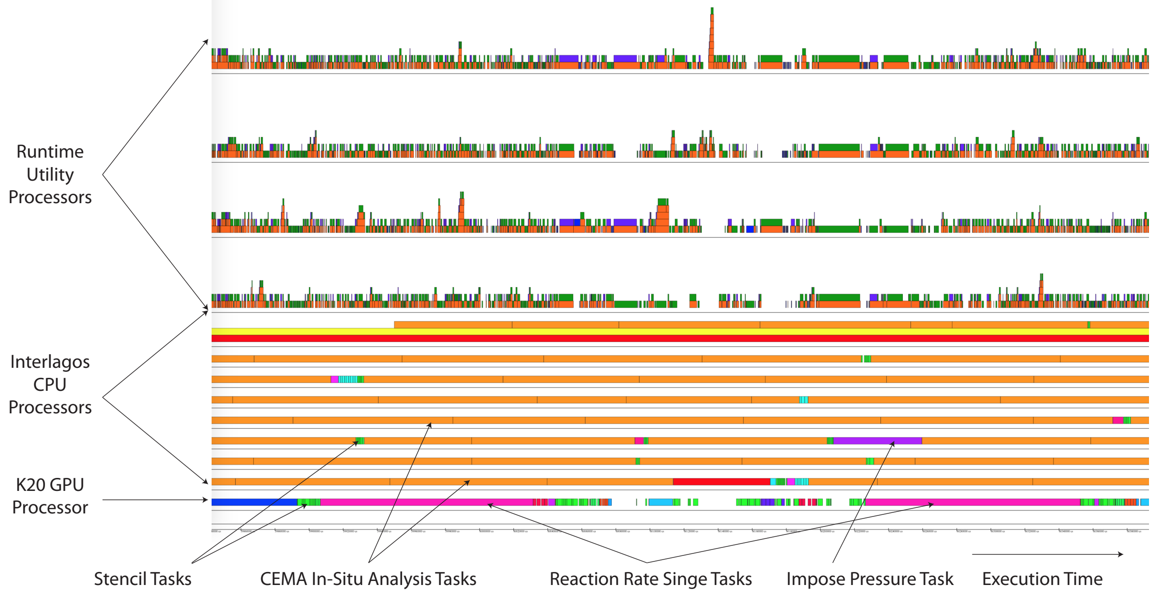


Figure 11.9: Scheduling of CEMA Tasks in S3D

Some sort of automated analysis is clearly needed. A relatively simple measure of *heat release* can easily identify auto-ignition kernels after they have ignited (this is what is actually being plotted in Figure 11.6) but it is much better to recognize such kernels before they ignite. One technique for this is *chemical explosive-mode analysis*, which uses the eigen decomposition of the Jacobian of the per-species reaction rates to determine which species, if any, are about to undergo (literally) explosive growth[46]. The analysis also indicates which species are driving that growth, providing a picture of which pathways in the overall chemical mechanism are active in which portions of the simulation volume.

While powerful, CEMA is too expensive to be performed on every grid point for every timestep. An offline analysis using just the data files written by the main simulation every thousandth timestep would have trouble keeping up, and would completely miss any event that occurred between those thousand-timestep snapshots. An alternative approach is to perform the analysis *in-situ* (i.e. as part of the main simulation), using random sampling of grid points on each timestep to keep the cost low while maintaining a high probability of capturing all the areas of interest[9].

This sampling approach had already been implemented in the Fortran version of S3D and was ported to the Legion version in less than a day. Rather than porting the CEMA code to C++, the Fortran code was used as-is, with a small C wrapper around it to provide the right task prototype. Legion's data model and apparently-sequential semantics allowed the task calls for the CEMA code to be inserted at an intuitive place for the programmer (i.e. immediately after the data that makes up the CEMA input is generated) while still exposing the independence of the CEMA tasks from

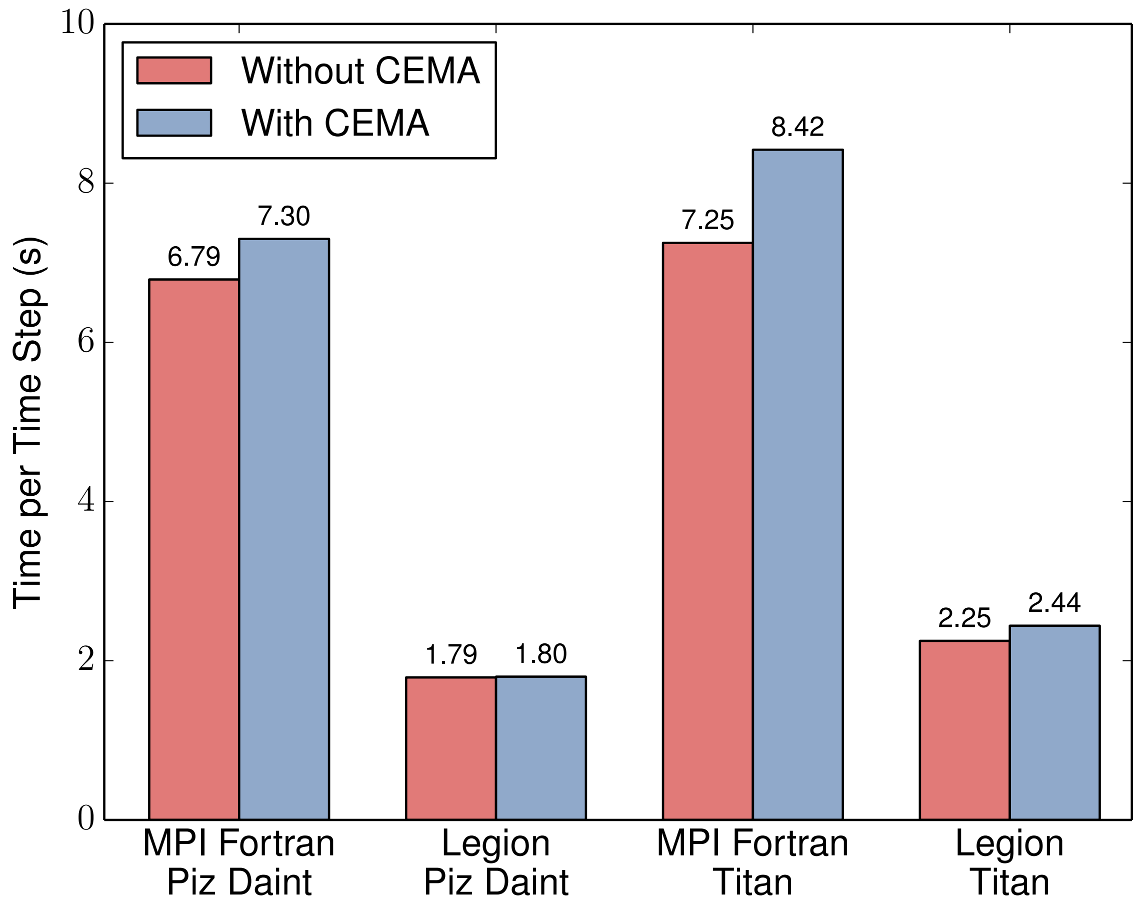


Figure 11.10: Execution Time Impact of CEMA in S3D

the main timestep loop. The CEMA tasks were mapped onto the CPU, for two reasons. First, the eigen decomposition code is very serial and not well suited for execution on a GPU. Second, and more importantly, the GPU was already being fully utilized for the main simulation code, while both Titan and Piz Daint had idle cycles on the CPU due to the bottleneck of the PCI-Express bus. Figure 11.9 shows an annotated version of the Legion profiler output for a CEMA-enabled run on Titan.

With the S3D mapper setting priorities on the CEMA tasks lower than those of the main simulation, the Realm task scheduler was able to make use of those idle CPU cycles while having minimal scheduling impact on the few (but very critical) simulation tasks that run on the CPU. Figure 11.10 shows the measured overhead for both S3D versions on both systems. The bulk-synchronous Fortran code exposes the full cost of the sampling-based CEMA, whereas the Legion version hides 83% of the overhead on Titan and over 98% on Piz Daint. In a sense, Realm's composable asynchrony is also able to provide overlap between two different computations with minimal investment from the programmer. This indicates that Realm is positioned well for future HPC applications, many of which are expected to incorporate multiple subsystems, modeling different physics or using different grids or meshes, or both.

11.7 Ongoing Work

The Legion version of S3D remains in active use and development. Although the initial porting and performance tuning was done by the Legion team, the code is back in the hands of the domain scientists. They are running their own simulations, and more excitingly, making their own modifications to the application that enables them to explore new areas.

A second effort is under way to port S3D to systems based on the Knights Landing architecture from Intel, such as Trinity (Figure 3.3). The best way to make use of a non-uniform many-core is an open question, and S3D (as well as other applications) may benefit from grouping the large number of cores into a smaller number of "OpenMP processors" or the like, as described in Section 7.5. Additionally, new variants of performance-critical kernels will need to be generated to take advantage of the 512-bit wide vector datapath in these processors. Ideally, this will be done using Realm's dynamic code generation capabilities, reducing rather than increasing the number of variants that appear in the source code.

Finally, the success with the in-situ CEMA has encouraged the implementation of in-situ versions of several other analyses. These might include visualization aids, online measurements of the size and shape of flame structures, the use of "tracer particles" to understand flows within the simulated volume. Thanks to Legion's automatic extraction of data dependencies and Realm's ability to efficiently schedule and prioritize both the tasks and any necessary data movement, these analyses can remain modular in the source code, avoiding quadratic increases in maintenance cost as more analyses are added.

Chapter 12

Conclusion

In this thesis, we have described the programmability crisis facing high performance computing, with the widening latency gap and growing heterogeneity of systems stretching the predominant bulk-synchronous programming model up to (or perhaps already past) its breaking point. We present Realm, which plays a key role in the Legion programming model's strategy for providing performance and portability on both current and planned future supercomputers.

We compare Realm to existing alternatives, and argue that Realm is currently unique in its offering the combination of:

- the use of *explicit dependencies* allowing the runtime to adapt to machine-to-machine (or even moment-to-moment) variations in the latencies of various operations while allowing application code to be written in a modular and maintainable way,
- a set of *functional portability abstractions*, allowing the code that initiates computation and communication within the application to remain agnostic to the exact kind(s) of processors or memories being used,
- an explicit design for how higher-level runtimes, libraries, tools, and ideally domain-specific languages can be mapped onto it,
- support for *task-parallelism* in addition to data-parallelism, as the computational intensity of most kernels cannot be arbitrarily increased to match the growing latency gap,
- a clear *separation of policy and mechanism* that prevents the application from having to fight with the underlying runtime to obtain a desired task scheduling or data placement,
- a *low overhead* implementation that provides performance and scalability equivalent to current bulk-synchronous programming yet handles the complex operation graphs that result from even apparently-simple applications,

- an online *machine model* and real-time *profiling framework* that allows an application to adapt to dynamic behavior of the application or underlying machine, or simply to characterize a machine in terms of its behavior on the application's specific workloads and their interactions rather than standard benchmarks or theoretical performance numbers, and
- support for *extensibility* of the runtime itself, to easily accommodate new types of processors, memories, communication networks, and ways of generating optimized code.

We do not claim that Realm's specific implementation is the only way to deliver these features. We expect and encourage others to look into runtimes that make different trade-offs between these features. For example, a runtime for "big data" applications might weight extensibility and adaptability via profiling more and relax the separation of policy and mechanism to some extent. However, we do strongly believe that any runtime that intends to provide performance portability should start from this set of offerings and reduce or remove them only if there is some other great gain to be had as a result.

Tools such as Realm that permit writing applications in a performance portable way make it easier, or perhaps make it even possible, to run fast on existing (and planned) supercomputers. This near-term benefit alone is probably sufficient reason to use such tools. In closing, we argue for a second benefit. Writing high performance computing applications that are performance portable will make future supercomputers faster.

In the past, deciding on the architecture of a new supercomputer was primarily a budgetary exercise. The main architectural knobs available were the speed of the processors, the size of the memory, and the number of nodes in the cluster. More was clearly better for each, so one simply made choices to match the financial, thermal, and/or physical constraints in effect at the time. The architectural design space has now become much more complicated. Heterogeneous processors and complicated memory hierarchies force trade-offs, as more of one processor or memory type often means less of another. Each new processor or memory type also brings new micro-architectural knobs, and especially for more specialized processor types, these knobs often have more impact on performance than the clock speed of the processor.

It is highly likely that there are points in this giant new design space that are significantly better than our current supercomputer designs. In some cases, incremental improvements have been identified, but hardware vendors choose not to deploy them because existing applications would not benefit. For example, the performance benefits of being able to turn part of a CPU's cache into a scratchpad memory have been explored by [24] and others, and while the capability exists in GPUs and in the new Knights Landing many-core CPUs, it has yet to become available in standard x86 CPUs. If existing applications were written in a way that allowed them to easily take advantage of such new features, hardware vendors would have a much stronger incentive to deploy them in their products.

The alternative to making incremental improvements to a known point in the architectural design space of supercomputers is to search completely new areas of the space. Although potentially very rewarding, it is currently infeasible to perform any kind of systematic search of the space. Research efforts have selected specific points in the space that seem interesting, but an evaluation requires the manual porting of applications of interest to the proposed architecture so that it can be characterized and/or simulated. The cost involved limits such efforts, and many promising ideas are probably missed due to differences in opinions regarding which applications are of interest.

Our hope is that by writing applications in a performance portable way, they will be able to make good use of not just existing systems, but also potential future systems. Functional portability abstractions and dynamic code generation would allow an application to be run on a simulator just as easily as it is run on an actual system. And applications that dynamically tune their behavior for the system on which they run can be used to start automating the search of the architectural design space, effectively tuning the architecture as well as the applications. There are clearly many further challenges that would need to be solved to reach this ideal of application/architecture co-design, but performance portability is an important first step.

Bibliography

[1] Emmanuel Agullo, Olivier Aumage, Mathieu Faverge, Nathalie Furmento, Florent Pruvost, Marc Sergent, and Samuel Thibault. *Achieving High Performance on Supercomputers with a Sequential Task-based Programming Model.* PhD thesis, Inria Bordeaux Sud-Ouest; Bordeaux INP; CNRS; Université de Bordeaux; CEA, 2016.

[2] Arvind, Rishiyur S Nikhil, and Keshav K Pingali. I-structures: Data structures for parallel computing. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 11(4):598–632, 1989.

[3] Cédric Augonnet, Samuel Thibault, Raymond Namyst, and Pierre-André Wacrenier. StarPU: A unified platform for task scheduling on heterogeneous multicore architectures. In *European Conference on Parallel Processing*, pages 863–874. Springer, 2009.

[4] Michael Bauer, Sean Treichler, and Alex Aiken. Singe: Leveraging warp specialization for high performance on GPUs. *ACM SIGPLAN Notices*, 49(8):119–130, 2014.

[5] Michael Bauer, Sean Treichler, Elliott Slaughter, and Alex Aiken. Legion: expressing locality and independence with logical regions. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, page 66. IEEE, 2012.

[6] Michael Bauer, Sean Treichler, Elliott Slaughter, and Alex Aiken. Structure slicing: Extending logical regions with fields. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 845–856. IEEE, 2014.

[7] Michael Edward Bauer. *Legion: Programming Distributed Heterogeneous Architectures with Logical Regions.* PhD thesis, Stanford University, 2014.

[8] Christian Bell, Dan Bonachea, Rajesh Nishtala, and Katherine Yelick. Optimizing bandwidth limited problems using one-sided communication and overlap. In *Proceedings 20th IEEE International Parallel & Distributed Processing Symposium.* IEEE, 2006.

[9] Janine C Bennett, Ankit Bhagatwala, Jacqueline H Chen, C Seshadhri, Ali Pinar, and Maher Salloum. Trigger detection for adaptive scientific workflows using percentile sampling. *arXiv preprint arXiv:1506.08258*, 2015.

[10] Ankit Bhagatwala, Jacqueline H Chen, and Tianfeng Lu. Direct numerical simulations of HCCI/SACI with ethanol. *Combustion and Flame*, 161(7):1826–1841, 2014.

[11] Robert D Blumofe, Christopher F Joerg, Bradley C Kuszmaul, Charles E Leiserson, Keith H Randall, and Yuli Zhou. Cilk: An efficient multithreaded runtime system. *Journal of parallel and distributed computing*, 37(1):55–69, 1996.

[12] Robert L Bocchino Jr, Vikram S Adve, Danny Dig, Sarita V Adve, Stephen Heumann, Rakesh Komuravelli, Jeffrey Overbey, Patrick Simmons, Hyojin Sung, and Mohsen Vakilian. A type and effect system for Deterministic Parallel Java. *ACM Sigplan Notices*, 44(10):97–116, 2009.

[13] François Broquedis, Jérôme Clet-Ortega, Stéphanie Moreaud, Nathalie Furmento, Brice Goglin, Guillaume Mercier, Samuel Thibault, and Raymond Namyst. hwloc: A generic framework for managing hardware affinities in hpc applications. In *PDP 2010-The 18th Euromicro International Conference on Parallel, Distributed and Network-Based Computing*, 2010.

[14] Ian Buck, Tim Foley, Daniel Horn, Jeremy Sugerman, Kayvon Fatahalian, Mike Houston, and Pat Hanrahan. Brook for GPUs: Stream computing on graphics hardware. *ACM Transactions on Graphics (TOG)*, 23(3):777–786, 2004.

[15] Zoran Budimlić, Michael Burke, Vincent Cavé, Kathleen Knobe, Geoff Lowney, Ryan Newton, Jens Palsberg, David Peixotto, Vivek Sarkar, Frank Schlimbach, et al. Concurrent collections. *Scientific Programming*, 18(3-4):203–217, 2010.

[16] Titan: Early experience with the Cray XK6 at Oak Ridge National Laboratory. `https://cug.org/proceedings/attendee_program_cug2012/includes/files/pap138-file2.pdf`, 2012.

[17] Getting applications ready for Summit. `https://www.olcf.ornl.gov/wp-content/uploads/2014/12/20141210-CAAR-Webinar.pdf`, 2014.

[18] SciDAC PI Meeting - 23 July 2015. `https://www.orau.gov/scidac3pi2015/presentations/ThursdayAM/Lunch_Wells-SciDAC_23072015_FINAL.pdf`, 2015.

[19] William W Carlson, Jesse M Draper, David E Culler, Kathy Yelick, Eugene Brooks, and Karen Warren. Introduction to upc and language specification. Technical report, Technical Report CCS-TR-99-157, IDA Center for Computing Sciences, 1999.

[20] Bryan Catanzaro, Michael Garland, and Kurt Keutzer. Copperhead: compiling an embedded data parallel language. *ACM SIGPLAN Notices*, 46(8):47–56, 2011.

[21] Vincent Cavé, Jisheng Zhao, Jun Shirako, and Vivek Sarkar. Habanero-Java: The new adventures of old X10. In *Proceedings of the 9th International Conference on Principles and Practice of Programming in Java*, pages 51–61. ACM, 2011.

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

[23] Philippe Charles, Christian Grothoff, Vijay Saraswat, Christopher Donawa, Allan Kielstra, Kemal Ebcioglu, Christoph Von Praun, and Vivek Sarkar. X10: an object-oriented approach to non-uniform cluster computing. *ACM Sigplan Notices*, 40(10):519–538, 2005.

[24] Henry Cook, Krste Asanovic, and David A Patterson. Virtual local stores: Enabling software-managed memory hierarchies in mainstream computing environments. *EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2009-131*, 2009.

[25] Piz Daint. `http://user.cscs.ch/computing_systems/piz_daint/index.html`, 2013.

[26] David E. Culler, Seth Copen Goldstein, Klaus E. Schauser, and Thorsten Voneicken. Tam-a compiler controlled threaded abstract machine. *Journal of Parallel and Distributed Computing*, 18(3):347–370, 1993.

[27] Leonardo Dagum and Ramesh Menon. OpenMP: an industry standard API for shared-memory programming. *IEEE computational science and engineering*, 5(1):46–55, 1998.

[28] Trinity: Advancing Predictive Capability for Stockpile Stewardship. `http://www.lanl.gov/projects/trinity/specifications.php`, 2014.

[29] Summit: Oak Ridge National Laboratory's next High Performance Supercomputer. https://www.olcf.ornl.gov/summit/, 2015.

[30] Aurora. `http://aurora.alcf.anl.gov/`, 2016.

[31] Alejandro Duran, Eduard Ayguadé, Rosa M Badia, Jesús Labarta, Luis Martinell, Xavier Martorell, and Judit Planas. OmpSs: A proposal for programming heterogeneous multi-core architectures. *Parallel Processing Letters*, 21(02):173–193, 2011.

[32] H Carter Edwards and Daniel Sunderland. Kokkos Array performance-portable manycore programming model. In *Proceedings of the 2012 International Workshop on Programming Models and Applications for Multicores and Manycores*, pages 1–10. ACM, 2012.

[33] Kayvon Fatahalian, Daniel Reiter Horn, Timothy J Knight, Larkhoon Leem, Mike Houston, Ji Young Park, Mattan Erez, Manman Ren, Alex Aiken, William J Dally, et al. Sequoia:

Programming the memory hierarchy. In *Proceedings of the 2006 ACM/IEEE conference on Supercomputing*, page 83. ACM, 2006.

[34] Chrome V8. `https://developers.google.com/v8/`, 2012.

[35] R.D. Hornung and J.A. Keasler. The raja portability layer: Overview and status. Technical report, Lawrence Livermore National Laboratory (LLNL), Livermore, CA, 2014.

[36] R. Jagannathan. Coarse-grain dataflow programming of conventional parallel computers. In *Advanced Topics in Dataflow Computing and Multithreading*, pages 113–129. IEEE, 1995.

[37] OpenGL 4.4 API Specification. `https://www.opengl.org/registry/`, 2014.

[38] The OpenCL Specification, Version 2.1. `https://www.khronos.org/registry/cl/specs/opencl-2.1.pdf`, 2015.

[39] SPIR-V 1.1 Specification. `https://www.khronos.org/registry/spir-v/specs/1.1/SPIRV.pdf`, 2016.

[40] Seung Ook Kim, Minh Bau Luong, Jacqueline H Chen, and Chun Sang Yoo. A DNS study of the ignition of lean PRF/air mixtures with temperature inhomogeneities under high pressure and intermediate temperature. *Combustion and Flame*, 162(3):717–726, 2015.

[41] Siu Kwan Lam, Antoine Pitrou, and Stanley Seibert. Numba: a LLVM-based Python JIT compiler. In *Proceedings of the Second Workshop on the LLVM Compiler Infrastructure in HPC*, page 7. ACM, 2015.

[42] Chris Lattner and Vikram Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *Proceedings of the 2004 International Symposium on Code Generation and Optimization (CGO'04)*, Palo Alto, California, Mar 2004.

[43] John M Levesque, Ramanan Sankaran, and Ray Grout. Hybridizing S3D into an exascale application using OpenACC: an approach for moving to multi-petaflops and beyond. In *Proceedings of the International conference on high performance computing, networking, storage and analysis*, page 15. IEEE, 2012.

[44] LLVM Language Reference Manual. `http://llvm.org/releases/3.9.0/docs/LangRef.html`, 2016.

[45] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. Pin: building customized program analysis tools with dynamic instrumentation. *ACM Sigplan Notices*, 40(6):190–200, 2005.

[46] Zhaoyu Luo, Chun Sang Yoo, Edward S Richardson, Jacqueline H Chen, Chung K Law, and Tianfeng Lu. Chemical explosive mode analysis for a turbulent lifted ethylene jet fl ame in highly-heated coflow. *Combustion and Flame*, 159(1):265–274, 2012.

[47] Getting started with Direct3D. `https://msdn.microsoft.com/en-us/library/windows/desktop/hh769064(v=vs.85).aspx`, 2015.

[48] Philip J Mucci, Shirley Browne, Christine Deane, and George Ho. PAPI: A portable interface to hardware performance counters. In *Proceedings of the department of defense HPCMP users group conference*, pages 7–10, 1999.

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

[50] CUDA programming guide 5.5. `http://developer.download.nvidia.com/compute/DevZone/docs/html/C/doc/CUDA_C_Programming_Guide.pdf`, Sept. 2013.

[51] NVIDIA DGX SATURNV. `http://www.nvidia.com/object/dgx-saturnv.html`, 2016.

[52] The Open Community Runtime interface. `https://xstackwiki.modelado.org/Open_Community_Runtime`, 2016.

[53] The OpenACC application programming interface, version 2.5. `http://www.openacc.org/sites/default/files/OpenACC_2pt5.pdf`, 2015.

[54] Dragos Sbîrlea, Zoran Budimlić, and Vivek Sarkar. Bounded memory scheduling of dynamic task graphs. In *Proceedings of the 23rd international conference on Parallel architectures and compilation*, pages 343–356. ACM, 2014.

[55] Eric Schkufza, Rahul Sharma, and Alex Aiken. Stochastic superoptimization. *ACM SIGPLAN Notices*, 48(4):305–316, 2013.

[56] Galen Shipman, Patrick McCormick, Kevin Pedretti, Stephen Lecler Olivier, Kurt Brian Ferreira, Jacqueline H. Chen, Ramanan Sankaran, Sean Treichler, Alex Aiken, and Michael Bauer. Dynamic task scheduling to mitigate system performance variability. `http://www.osti.gov/scitech/servlets/purl/1249032`, 2015.

[57] Elliott Slaughter, Wonchan Lee, Sean Treichler, Michael Bauer, and Alex Aiken. Regent: A high-productivity programming language for HPC with logical regions. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, page 81. ACM, 2015.

[58] Marc Snir, Steve Otto, Steven Huss-Lederman, David Walker, and Jack Dongarra. *MPI: The Complete Reference*. MIT, 1998.

[59] Adrian Soviani and Jaswinder Pal Singh. Optimizing communication scheduling using dataflow semantics. In *2009 International Conference on Parallel Processing*, pages 301–308. IEEE, 2009.

[60] Xavier Teruel, Xavier Martorell, Alejandro Duran, Roger Ferrer, and Eduard Ayguadé. Support for openmp tasks in nanos v4. In *Proceedings of the 2007 conference of the center for advanced studies on Collaborative research*, pages 256–259. IBM Corp., 2007.

[61] Devesh Tiwari, Saurabh Gupta, James Rogers, Don Maxwell, Paolo Rech, Sudharshan Vazhkudai, Daniel Oliveira, Dave Londo, Nathan DeBardeleben, Philippe Navaux, et al. Understanding gpu errors on large-scale hpc systems and the implications for system design and operation. In *2015 IEEE 21st International Symposium on High Performance Computer Architecture (HPCA)*, pages 331–342. IEEE, 2015.

[62] Robert M Tomasulo. An efficient algorithm for exploiting multiple arithmetic units. *IBM Journal of research and Development*, 11(1):25–33, 1967.

[63] The TOP500 list. `http://www.top500.org`, 2016.

[64] Jeffrey S Vetter, Richard Glassbrook, Jack Dongarra, Karsten Schwan, Bruce Loftis, Stephen McNally, Jeremy Meredith, James Rogers, Philip Roth, Kyle Spafford, et al. Keeneland: Bringing heterogeneous GPU computing to the computational science community. *Computing in Science and Engineering*, 13(5):90–95, 2011.

[65] Yonghong Yan, Jisheng Zhao, Yi Guo, and Vivek Sarkar. Hierarchical place trees: A portable abstraction for task parallelism and data movement. In *International Workshop on Languages and Compilers for Parallel Computing*, pages 172–187. Springer, 2009.

[66] K. Yelick et al. Productivity and performance using partitioned global address space languages. In *PASCO*, pages 24–32, 2007.

[67] Kathy Yelick, Luigi Semenzato, Geoff Pike, Carleton Miyamoto, Ben Liblit, Arvind Krishnamurthy, Paul Hilfinger, Susan Graham, David Gay, Phil Colella, and Alex Aiken. Titanium: A high-performance Java dialect. *Concurrency Practice and Experience*, 10(11-13):825–836, 9 1998.