PROGRAMMING MANY-CORE SYSTEMS WITH GRAMPS

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

Jeremy Sugerman
August 2010

This dissertation is online at: http://purl.stanford.edu/cd589ky4265

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.

**Patrick Hanrahan, 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.

**Mendel Rosenblum**

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.

**Kurt Akeley**

Approved for the Stanford University Committee on Graduate Studies.

**Patricia J. Gumport, Vice Provost Graduate Education**

*This signature page was generated electronically upon submission of this dissertation in electronic format. An original signed hard copy of the signature page is on file in University Archives.*

# Abstract

The era of obtaining increased performance via faster single cores and optimized single-thread programs is over. Instead, a major factor in new processors' performance comes from parallelism: increasing numbers of cores per processor and threads per core. At the same time, highly parallel GPU cores, initially developed for shading, are increasingly being adopted to offload and augment conventional CPUs, and vendors are already discussing chips that combine CPU and GPU cores. These trends are leading towards heterogeneous, commodity, many-core platforms with excellent *potential* performance, but also (not-so-excellent) significant *actual* complexity. In both research and industry run-time systems, domain-specific languages, and more generally, *parallel programming models*, have become the tools to realize this performance and contain this complexity.

In this dissertation, we present GRAMPS, a programming model for these heterogeneous, commodity, many-core systems that expresses programs as graphs of thread- and data-parallel stages communicating via queues. We validate its viability with respect to four design goals—broad application scope, multi-platform applicability, performance, and tunability—and demonstrate its effectiveness at minimizing the memory consumed by the queues.

Through three case studies, we show applications for GRAMPS from domains including interactive graphics, MapReduce, physical simulation, and image processing, and describe GRAMPS runtimes for three many-core platforms: two simulated future rendering platforms and one current multi-core x86 machine. Our GRAMPS runtimes

efficiently recognize and exploit the available parallelism while containing the footprint/buffering required by the queues. Finally, we discuss how GRAMPS's scheduling compares to three archetypal representations of popular programming models: task-stealing scheduling, breadth-first scheduling, and static scheduling. We find that when structure is present, GRAMPS's adaptive, dynamic scheduling provides good load-balance with low overhead and its application graph gives it multiple advantages for managing the run-time depths of the queues and their memory footprint.

# Acknowledgements

My parents, Sharon and Art Sugerman, were enthusiastic and supportive when I ended more than five years of working full-time to seek a Ph.D. Nearly seven years later, they are still supportive, though in their own way each has made it clear that while they respected my disinclination to hurry, one is expected to graduate at some point. My advisor has made a similar point.

Speaking of which, I am very glad to have had Pat Hanrahan as my advisor. We have not always had an easy time understanding one another—despite both being fluent in Technical-Person English, our personal dialects are all but disjoint in some places—but we generally worked our way to common ground and I learned far more from the exposure to a mind and character I respected that was so different from my own. He indulged and abetted my masquerading as a graphics person, and even sort-of becoming one.

I am also grateful to Kurt Akeley and Mendel Rosenblum for serving on my reading committee and being excellent resources and role models. Kurt is an amazing engineer. It is a pleasure and an education to watch him digest a topic, talk, or paper and hear the content and phrasing of the questions he asks in the process. I owe Mendel an enormous debt of gratitude for many, *many* things: all his signatures my first pass at Stanford; VMware, plus his friendship and teaching there; support applying to grad school; and all the conversations and advice throughout school, even though we barely talked about my actual research.

Eric Darve and Christos Kozyrakis rounded out my orals committee. Eric unhesitatingly agreed to be my chair despite our only casual interactions. Christos not only welcomed my crashing of his group, but requested it, promoted GRAMPS, and

# Contents

# List of Tables

# List of Figures

# Chapter 1

# Introduction

Commodity computing and computers are in the midst of transition: the era of the single fast core is over. Multi-core CPUs are now prevalent and core counts are increasing in accordance with Moore's Law. Five years after the first dual-core x86 chips appeared, eight and twelve-core versions have been released. The rapid rise of independent cores per chip has made software parallelism critical to performance: *scale-up*, maximizing a core's utilization, was once paramount, but it is now rivaled or surpassed by *scale-out*, maximizing incremental performance as more cores are employed. Quite simply, with eight or twelve cores, realizing 80% performance from all of them dwarfs realizing 95% performance from only one.

Over the same period, the competing design pressures of power consumption and performance have increasingly promoted diverse heterogeneous platforms: different mixtures of different kinds of 'cores'. Programmable GPU cores originally designed for shading are now used to accelerate general purpose (i.e., non-graphics) applications by more than order of magnitude relative to conventional CPU cores in the same cost/design envelope. CPU designers are experimenting with their own trade-offs in core design. The Cell processor and Intel's Larrabee (now Knights) architecture both offer high computational abilities by using many copies of a simple core design in place of fewer copies of a state-of-the-art out-of-order superscalar processor [35, 40]. Additionally, the fact of multiple cores on the same chip, as opposed to multiple chips, has the potential to facilitate heterogeneity: so long as one core on the chip

can bootstrap the machine, the others can be more specialized. This has CPU vendors pursuing future chips that contain both conventional processors and GPU cores [1].

These impending heterogeneous, many-core, commodity machines offer enormous amounts of potential computational resources. Actually exploiting that potential, however, is a nontrivial challenge. Both parallelism and, especially, heterogeneous parallelism are major challenges for software development. Tackling them has created renewed interest in both academia and industry for developing *high-level programming models*, such as OpenCL [23], Cilk [15], and OpenMP [13]. High-level programming models provide two major advantages over low-level interfaces such as threads and locks. First, they provide *constructs* that make it easier for programmers to express parallelism. Second, their runtimes and schedulers *manage concurrency and communication*, simplifying these burdens for the programmer. This thesis introduces GRAMPS, a new high-level programming model for heterogeneous, parallel systems.

High-level programming models achieve their advantages by placing the application-programming model boundary closer to application constructs than hardware interfaces. This gives them more insight into application structure, patterns, and semantics. In exchange, they must make/impose assumptions about the chief types of applications and developers that they target.

Our application focus stems from three beliefs: interesting applications have irregularity; large bundles of coherent work are efficient; and the producer-consumer/pipeline idiom is important. These beliefs are rooted in our experiences, positive and negative, with graphics workloads and wrestling with graphics hardware for GPGPU purposes, but also draw upon our exposure to streaming architectures (and traditional parallel programming for CPUs). Irregularity takes many forms: different input elements may take different lengths of time to process, generate different numbers of outputs or different communication patterns, etc. At the same time, coherency enables amortization: it is more efficient to make one scheduling decision for one thousand items than one for each; to use SIMD execution to process a batch of items at a time; or to group work by access pattern for memory system locality. Finally, many workloads generate significant quantities of temporary/intermediate data during computation

that can be consumed incrementally as it appears. Not only does this allow parallelism of production and consumption, it has a powerful effect on locality: the data can be processed without the bandwidth and storage costs of spilling it to buffer in the higher levels of the memory hierarchy. We will (re)build program coherence dynamically by having the application expose related work to GRAMPS, which will aggregate it.

Our developer focus is systems-savvy developers: programmers who are well informed about their hardware and the best practices for it, but who dislike rote and prefer to use and build tools that systematize those idioms instead. A particular characteristic of this audience that influences our design in multiple places is a preference for the option of trading conservative guarantees for expressiveness. That is, a few sharp edges, if well-identified, optional, and powerful when used appropriately, are preferable to philosophically disallowing them for safety.

This thesis makes three main contributions:

1. The definition and design of GRAMPS (Chapter 3): programs as graphs of independent stages connected via queues; queues with application specified capacities and packet (work) sizes; and stages with can support no, limited, or total automatic intra-stage parallelism with static, conditional, or in-place (reduction) outputs.

2. Validation of GRAMPS with respect to four design goals (Chapters 4, 5, 6):

   - **Broad application scope:** It should be possible to express a wide range of algorithms in GRAMPS and more convenient and effective than roll-your-own approaches.

   - **Multi-platform applicability:** GRAMPS should suit a wide range of heterogeneous, many-core, commodity configurations.

   - **Performance:** Applications built in GRAMPS should give up little performance over native implementations, specifically in terms of scale-out parallelism and buffering of intermediate data.

- **Tunability:** Developers should be able to become experts in GRAMPS, i.e., be able to explain the performance of their applications and be able to tune (optimize) them by adapting their use of the programming model.

3. Demonstration that GRAMPS performs better than current general-purpose programming models, with respect to the important metric of memory allocated to queues, without giving up performance (Chapter 6).

Tunability perhaps seems incongruous to have as such a prominent design goal. We believe it is absolutely critical for our audience: an un-tunable system is a system which can have no expert users (by definition) and heterogeneous, parallel programming is too complex to fully automate. Skilled developers will always know more about their workloads than any programming model can capture or infer. They will expect to have to optimize and refine their programs to employ that knowledge and will have little use for any tool that does not embrace that process.

Boiled down, these contributions amount to two things: (i) GRAMPS is a viable programming model for many applications on current and likely future machines; (ii) GRAMPS can be implemented to achieve good parallel scalability while keeping a low queue (memory) footprint. After a background discussion of current parallel programming models (Chapter 2) and a detailed overview of GRAMPS (Chapter 3), these assertions are validated by three case studies: GRAMPS for simulated future graphics architectures (Chapter 4), GRAMPS for current multi-core x86 machines (Chapter 5), and a horizontal comparison to schedulers based on alternative programming models (Chapter 6). We end with some overall discussion (Chapter 7) and provide a sample GRAMPS application as an appendix (Appendix A).

# Chapter 2

# Parallel Programming Models

This thesis proposes GRAMPS as a new programing model for parallel applications, but there are many existing ones. This chapter surveys the alternatives and identifies their defining properties.

## 2.1 Domain Specific Programming Models

Programming models are able to simplify development and accelerate and/or manage applications by imposing and systematizing assumptions about their behavior. One common and effective way to do this is to restrict the supported application domain: this enables data types, synchronization primitives, and scheduling to be tailored to application semantics. Domain specific programming models are widespread, and range from long-lasting formalized models such as SQL for databases [45] to more one-off ad hoc models such as the plugin interfaces for web browsers (e.g., [28]).

### 2.1.1 Real-Time Graphics Pipelines

The most relevant examples to GRAMPS are the rendering-specific programming models of OpenGL and Direct3D [39, 5]. These two, which are very close to each other, provide developers a simple, vendor-agnostic interface for describing real-time graphics computations. More importantly, the pipeline and programmable shading

| Model | Supports Shader | Producer-Consumer | Hierarchical Work | Adaptive Scheduling | Examples |
|---|---|---|---|---|---|
| **Task-Stealing** | No | No | No | Yes | Cilk, TBB, OpenMP |
| **Breadth-First** | Yes | No | Yes | No | CUDA, OpenCL |
| **Static** | Yes | Yes | Yes | No | StreamIt, Imagine |
| **GRAMPS** | **Yes** | **Yes** | **Yes** | **Yes** | – |

Table 2.1: Summary of different general purpose programming models.

abstractions exported by these interfaces are generally backed by highly-tuned GPU-based implementations. By using rendering-specific abstractions (such as vertices, fragments, and pixels) OpenGL/Direct3D maintains high performance without introducing difficult concepts such as parallelism, threads, asynchronous processing, or synchronization. The drawback of these design decisions is limited flexibility. Applications must be restructured to conform to the pipeline that OpenGL/Direct3D present. A fixed pipeline makes it difficult to implement many advanced rendering techniques efficiently. Extending the graphics pipeline with new domain-specific stages or data flows to provide new functionality has been the subject of many proposals, for example, [5, 18, 4], but their availability is gated by their endorsement, adoption, and release by the OpenGL/Direct3D model owners. In the first case study, we will look closely at using GRAMPS to implement a graphics pipeline more flexibly. In fact, one of the early motivations that led to GRAMPS was to enable "non-owners" of OpenGL/Direct3D to make performance-efficient adjustments to the graphics pipeline.

## 2.2   General Purpose Programming Models

Other types of parallel programming models aim to be general purpose. They make simplifying assumptions/gain workload insight with higher-level operations and data-structures than raw kernel threads and locks, but avoid explicit expectations coupled to a particular application domain.

There are a great many general purpose parallel programming models, but we believe they mostly fit into three broad categories—Task-Stealing, Breadth-First, and

Static—which we name in terms of their scheduling policies and for which we can identify canonical traits. While we will discuss particulars of specific implementations below, they have significant biases that complicate direct comparisons (due to unevenly tuned runtimes, different toolchains, varying benchmark implementations, etc.). Our primary objective is *to distill the key scheduling policies*, which we will use to compare them with GRAMPS and each other in Chapter 6.

Table 2.1 summarizes the three canonical models (and GRAMPS) according to four criteria:

- **Support for shaders**: The model provides a built-in construct for data-parallel kernels, which are automatically parallelized.

- **Support for producer-consumer**: The runtime is aware of data produced as intermediate results (i.e., created and consumed during execution) and attempts to exploit this awareness during scheduling.

- **Hierarchical work**: The model allows work to be expressed at different granularities rather than all being expressed at the finest granularity.

- **Adaptive scheduling**: The scheduler makes use of information available at runtime, and can choose from all available work to execute.

## 2.2.1 Task-Stealing

Task-Stealing models are characterized by an underlying pool of threads that execute work which the application explicitly divides into independent *tasks*. Each thread has a queue of ready tasks, to which it enqueues and dequeues work. For load balance, when a thread's local queue is empty, it tries to steal tasks from other threads. Task-Stealing is a common low-level scheduling technique implemented by several runtimes, such as Cilk [15], TBB [21] and OpenMP [13].

Task-Stealing runtimes often focus on low-overhead task creation and execution in order to exploit fine-grain parallelism [3]. As a result, they tend to avoid any features that add per-task overhead, such as priorities, dependencies, or hierarchies. Rather,

they operate on 'task soup', a sea of equivalent-looking tasks. Since Task-Stealing systems offer minimal flow control or synchronization builtins, applications often roll their own locks and other primitives, which are then opaque to the programming model.

Task-Stealing models do not aggressively try to minimize memory footprint, but some provide guarantees that memory footprint grows at most linearly with the number of threads (e.g., by restricting which tasks to steal and using LIFO task queues [15]).

Thus, in terms of the criteria of Table 2.1, Task-Stealing only exhibits adaptive scheduling. Models can, and are designed to, schedule whatever tasks they believe enhance load-balance. The tasks, however, are essentially unstructured: all are equal in the eyes of the scheduler without any notion of grouping for hierarchy or recognizing producer-consumer behavior and without any support for automatic data-parallelism.

### 2.2.2   Breadth-First

Breadth-First models are fundamentally about simple data-parallelism. They express programs as a sequence (DAG or pipeline) of implicitly data-parallel kernels and run one kernel at a time, with the runtime automatically instancing and managing enough independent copies of the kernel to utilize the available hardware resources. Data-parallel algorithms initially arose in the context of massively parallel supercomputers, such as the Connection Machine [19], and were exemplified by programming models/languages such as C* [38]. The archetypal current Breadth-First models are those arising from GPGPU programming (using GPUs for non-graphics general purposes), such as OpenCL and CUDA [23, 31].

Executing one kernel at a time with barriers between stages/passes is straightforward to implement (and understand), and highly effective for regular, divide-and-conquer, algorithms. Breadth-First's two weaknesses, though, are footprint and load-balance in the presence of irregularity. Scheduling breadth-first is the opposite of producer-consumer: the entire output of one kernel must be accumulated and buffered before any can be consumed by the next and no pipeline parallelism can be used to

overlap stages that cannot by themselves fill a machine.

Referring back to Table 2.1, Breadth-First models excel at shader support and include some hierarchical notion of execution: processing is expressed in terms of kernels launched on large input data sets with the fine-grained divisions onto hardware resources handled internally. As mentioned above, though, producer-consumer communication between kernels is completely impossible. And scheduling, while potentially somewhat dynamic in the dispatching of intra-kernel threads/instances, is not adaptive: no matter how plentiful the work available for other kernels is, only one kernel runs at a time.

### 2.2.3 Static

Static models are very similar to GRAMPS, only scheduled according to offline/upfront transformations and decisions rather than dynamically. Applications are expressed graphs of stages that explicitly interact via data streams. The programming model automatically instances them and attempts a layout that exploits producer-consumer locality. They trade the adaptability of dynamic scheduling for up-front transformations to maximize run-time computational density. The (offline) scheduler needs full *prior* knowledge of the execution requirements of each stage and their cross-stage communication patterns, which are obtained from the compiler, a profiler, and/or user annotations. These models are typified by streaming systems, such as StreamIt [43] and architectures like Imagine [22], Merrimac [11], and RAW [44]. Unsurprisingly, their biggest weakness is handling irregularity/input-dependent workloads where execution behavior, communication patterns, or both, are impossible to approximate or know in advance.

Static models fulfill three of the four criteria used in Table 2.1: all but adaptive scheduling. From an application development perspective, they extend Breadth-First to add a full execution graph and allow overlapping of stages (i.e., kernels), and thus capture producer-consumer interactions. From an implementation perspective, however, Static models are entirely distinct: as its name implies, the entire execution order is predetermined before execution begins, the opposite of adaptivity.

## 2.3  Conclusion

The primary advantages programming models offer come from the unifying/simplifying assumptions they make in the constructs they expose and the resulting leverage that gives them for implementation and scheduling. Making highly domain-specific assumptions has allowed very efficient, high performance implementations of real-time graphics pipelines on GPUs. At the same time, encoding application-domain traits naturally interferes with extending or modifying the range of applications well-suited to a model. In the next chapter, the first case study, we will look at using GRAMPS as an intermediate layer upon which the real-time graphics pipeline can be an application and use this indirection to both extend it and replace it completely.

There is also a diverse array of general purpose programming models. Among those relevant to our focus, we feel they group roughly into three canonical representatives: Task-Stealing, Breadth-First, and Static. Task-Stealing focuses on lightweight tasks and adaptive scheduling, Breadth-First focuses on data-parallelism, and Static focuses on regular/predictable execution graphs. Each fulfills a different subset of the key criteria we see for GRAMPS: support for shaders; support for producer-consumer; constructs for hierarchical work; and adaptive scheduling (Table 2.1). In Chapter 6, the third case study, we will compare GRAMPS and all three alternatives to show the impact of each criterion.

# Chapter 3

# The GRAMPS Programming Model

This chapter describes the GRAMPS programming model and its abstractions. It does not include execution details, such as scheduling algorithms, which are traits of particular implementations of the programming model. Those are described in the case studies in the subsequent chapters. Most of the programming model description has been published previously in [41] and/or [42].

## 3.1   GRAMPS Design

GRAMPS is a General Runtime/Architecture for Many-core Parallel Systems. It defines a programming model for expressing pipeline and computation-graph style parallel applications. It exposes a small, high-level set of primitives designed to be simple to use, to exhibit properties necessary for high-throughput processing, and to permit efficient implementations. We intend for GRAMPS implementations to involve various combinations of software and underlying hardware support, similar for example, to how OpenGL permits flexibility in an implementation's division of driver and GPU hardware responsibilities. However, unlike OpenGL, we envision GRAMPS as being without ties to a specific application domain. Rather, it provides a substrate upon which domain-specific models can be built.
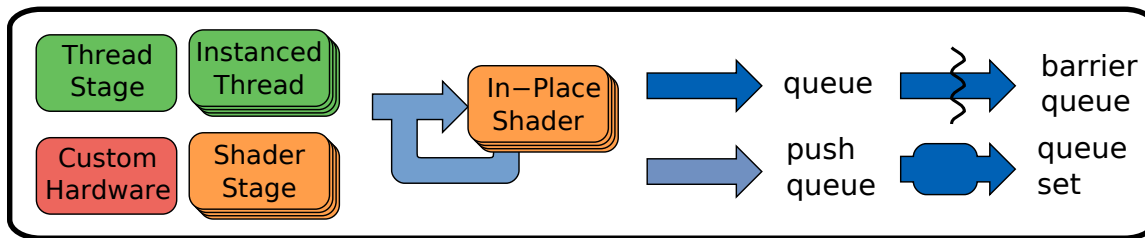
Figure 3.1: Legend: the design elements of a GRAMPS program.

GRAMPS is organized around the basic concept of application-defined independent computation stages executing in parallel and communicating asynchronously via queues. This relatively simple producer-consumer idiom is fundamental across a broad range of throughput applications. In keeping with our goal of domain independence, GRAMPS is designed to be decoupled from application-specific semantics such as data types and layouts and internal stage execution. It also extends these basic constructs to enable additional features such as limited and full automatic intra-stage parallelization and mutual exclusion. With these, we can build applications from many domains: rendering pipelines, MapReduce, sorting, signal processing, and others. Figure 3.1 enumerates the building blocks of a GRAMPS application and will serve as a legend for all of the GRAMPS application graphs in this thesis. The rest of this chapter will first go through a non-technical example to help build an intuition and then describe GRAMPS's components in technical detail. In addition, Appendix A enumerates the precise interface our GRAMPS implementations export and provides the full contents of one of our regression tests as an example.

## 3.2   A GRAMPS Example

Figure 3.2 shows a (hypothetical) GRAMPS graph for making chocolate chip cookies. Rather than diving into a highly technical workload, this more approachable example is useful for explaining the highlights of the programming model. As shown, dough preparation is broken into individual stages corresponding to logical steps in
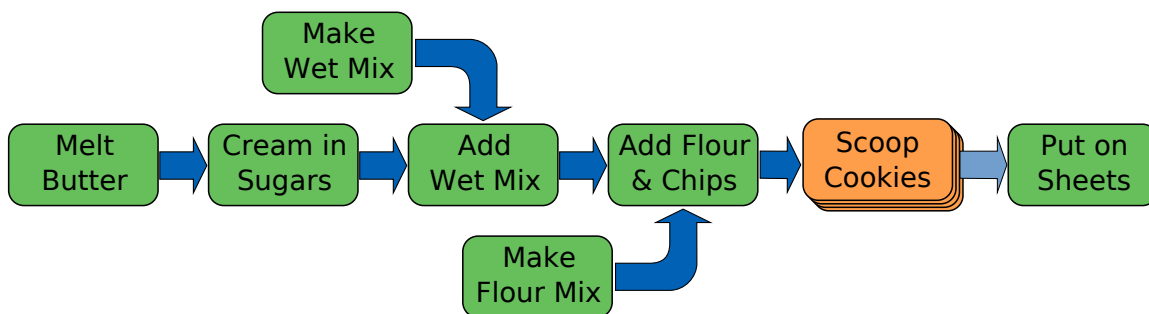
Figure 3.2: A hypothetical GRAMPS graph for preparing chocolate chip cookie dough.

the recipe [7]. Each stage sends its output downstream and takes as input the ingredients and/or batter from its prior stage(s). Most of the stages are Thread stages (serial) because they combine (mix/blend/cream) their inputs, but cookie scooping is a Shader (data-parallel): as many chefs as are available can all independently take a blob of dough and form it into cookies simultaneously. Since each can make cookies of different sizes and shapes, the stage uses `push` for output, which enables conditional and variable output. Note that, in this formulation, the formed cookies have to serialize to be placed on the cookie sheets for baking: this spares the data-parallel scoopers from synchronizing or colliding when accessing the shared cookie sheets, but also constrains the overall parallelism. Section 3.4.2 describes queue sets, which improve this situation, and shows an enhanced cookie dough graph 3.3.

Cookie dough preparation also presents a straightforward physical analog for considering queue footprint: counter space. Any staged ingredients and partially finished batches of batter need to be set aside until it is time to use them. And, in every kitchen where we have cooked, counter space is *always* at a premium. This provides a strong incentive to prepare the batter (schedule the graph stages) in an order that leaves minimal pending dishes and bowls to fit on the counters (buffer in the queues).

Similarly, the granularity with which the recipe (and all recipes) is designed illustrates another significant aspect of GRAMPS: the concept of an efficient, but natural, work size that motivates the packet-based organization of queues and dispatching of stages. The author of the recipe (designer of the GRAMPS graph) chooses quantities

for one batch that roughly fill the bowls, measuring utensils, etc. of the target bakers (caches, SIMD widths, etc. of the available cores). And, while a baker would likely scale up or down a little, for example, to double or halve a recipe, he or she is unlikely to expend the effort to prepare a batch scaled down to a single cookie and would likely split preparation into multiple batches rather than attempt a single tenfold-recipe.

## 3.3   Execution Graphs

GRAMPS application are expressed as execution graphs (also called computation graphs). The graph defines and organizes the stages, queues, and buffers to describe their data flow and connectivity. In addition to the most basic information required for GRAMPS to run an application, the graph provides valuable information about a computation that is essential to scheduling: insights into the program's structure. These include application-specified limits to the maximum depth for each queue, which stages are sinks and which sources, and whether there are limits on automatically instancing each stage. We have built a few different ways for developers to create execution graphs, all of them are wrappers around the same core API: an OpenGL/Streaming-like group of primitives to define stages, define queues, define buffers, bind queues and buffers to stages, and launch a computation.

Note that GRAMPS supports general computation graphs to provide flexibility for a rich set of algorithms. Graph cycles inherently make it possible to write applications that loop endlessly through stages and amplify queued data beyond the ability of any system to manage. Thus, GRAMPS, unlike pipeline-only APIs, does not guarantee that all legal programs robustly make forward progress and execute to completion. We have intentionally designed GRAMPS to encompass a larger set of applications that run well, at the cost of allowing some that do not.

Forbidding cycles would allow GRAMPS to guarantee forward progress—at any time it could stall a stage that was over-actively producing data until downstream stages could drain outstanding work from the system—at the expense of excluding some irregular workloads. Many of the GRAMPS versions of the applications in Chapters 4 and 5 contain cycles in their graph structure. Sometimes cycles can be

eliminated by "unrolling" a graph to reflect a maximum number of iterations, bounces, etc. However, not only is unrolling cumbersome for developers, it is awkward in irregular cases, such as when different light rays bounce different numbers of times depending on what objects and materials they hit. While handling cycles increases the scheduling burden for GRAMPS, it remains possible to effectively execute many graphs that contain them. We believe that the flexibility that graphs provide over pipelines and DAGs outweighs the cost of making applications take responsibility for ensuring they are well-behaved. The right strategy for notifying applications and allowing them to recover when their amplification swamps the system is an interesting avenue for future investigation.

## 3.4 Queues

GRAMPS queues provide the means by which the (independent) GRAMPS stages communicate and exchange data. Each queue has two key application-specified traits: its *packet* information—the type and granularity at which it handles data—and its *capacity*—the maximum number of packets it can contain at any one time. Capacity is an important knob the application can use to throttle a point in the execution graph, for example because it has application specific knowledge about where work is represented most compactly or where a producing stage will often outpace its consumers. Note that it is an optional knob, however: the schedulers in all of our GRAMPS implementations have various domain-oblivious heuristics for managing queue depths that handle most situations well. When developing applications, we generally start each queue with a large capacity and only need to tune one or two strategic places in the graph.

In order to support applications with ordering requirements (such as OpenGL or Direct3D), GRAMPS queues can also be specified as strictly FIFO. With multiple or instanced consumers, this incurs two kinds of overheads: reduced parallelism as out-of-order packets stall until the in-order stragglers arrive, and extra storage to buffer and reorder the delayed packets.

### 3.4.1   Packets

As mentioned above, the basic unit with which stages enqueue and dequeue work is called a packet. Each queue has its own packet layout, whose composition is application-specified. Packets are intended to allow application logic to collect and expose bundles of related work for efficient processing, for example, to amortize enqueue/dequeue operations, to fill hardware SIMD lanes, or to increase memory coherence/locality benefits. At the same time, in regions of the graph that are not performance critical, the application is free to use packets as small as it wishes. There are also often natural granularities to an algorithm itself, as with the cookie dough example.

GRAMPS imposes as few constraints on packet composition and layout as it can, so as to preserve application flexibility and keep its own interfaces simple. Packets have one of two high-level formats, defined as part of queue creation:

- **Opaque:** Opaque packets contain work that GRAMPS has no need to interpret. The application graph specifies only the size of Opaque packets (so GRAMPS can enqueue and dequeue them). Their contents are entirely defined and interpreted by the logic of stages that produce and consume them.

- **Collection:** Collection packets are for queues with at least one end bound to a data-parallel GRAMPS Shader stage. They represent a shared header together with a set of independent data elements which GRAMPS will dispatch as a unit. Collection packets specify three sizes: the packet size, the header size, and the per-element size. Additionally, GRAMPS reserves the first word of the header as a count of valid elements in the packet. Beyond that, the internal layout of elements and any addition header fields are opaque to GRAMPS.

### 3.4.2   Queue Sets

The cookie dough graph in Figure 3.2 exhibits a common serialization problem: independent parallel processing followed by synchronization for distribution using a shared resource. In this case, packets of dough can be scooped into individual cookies
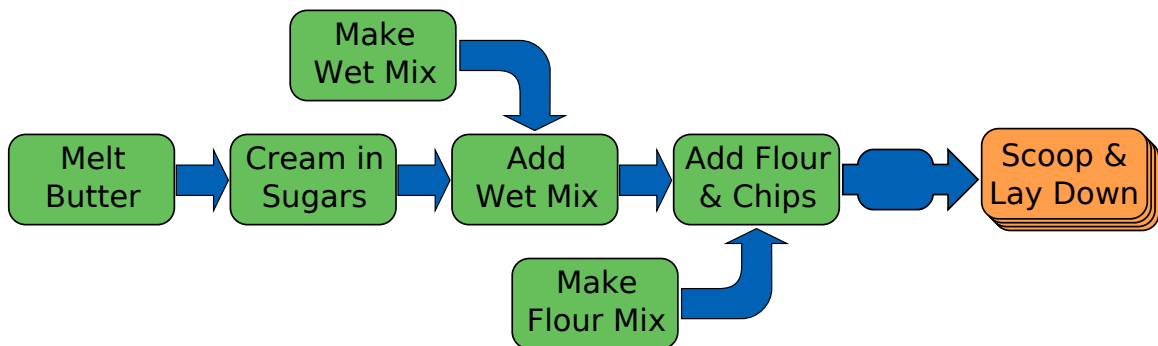
Figure 3.3: Replacing the Put on Sheets input queue from Figure 3.2 with a queue set replaces the serialized Put On Sheets with the combined parallelizable Scoop & Lay Down.

in parallel, but all cookies must route through a single baker to be put onto cookie sheets without conflicts or collisions. The same phenomenon happens in many parallel applications. For example, renderers often shade pixels independently before updating a shared frame buffer. On the one hand, this serialization often becomes a bottleneck, but on the other, introducing fine grained locking of cookie sheet or frame buffer locations has many unpleasant effects: managing explicit locking is a burden on application developers, suffers from contention, and can be subtle to keep correct. Instead, performant parallel applications often subdivide the shared resource into disjoint regions that can each be updated independently.

Figure 3.3 displays queue sets, GRAMPS's construct for applying this technique. A queue set functions like $N$ 'subqueues' bundled together as a single logical queue. Different subqueues can be processed in parallel, but GRAMPS ensures that for each subqueue at most one packet is in flight at a time. In the updated example, each subqueue corresponds to a distinct cookie sheet (or portion of a cookie sheet).

An application can use a queue set statically or dynamically. When used statically, the graph specifies the number of subqueues and the stages reference them with dense indices. Dynamically, producing output to a non-existent subqueue creates it and subqueues are identifies with arbitrary keys that GRAMPS transparently internally maps onto dense indices (or some other suitable data structure).

## 3.5   Stages

GRAMPS stages correspond to nodes in the execution graph. The fundamental reason to partition computation into stages is to increase performance. Stages operate asynchronously and therefore expose inter-stage parallelism. More importantly, stages encapsulate phases of computation and indicate computations that exhibit similar execution or data access characteristics (typically SIMD processing or memory locality). Grouping these computations together yields opportunities for efficient processing. Additionally, some computations are data-parallel and separating them as distinct stages exposes intra-stage parallelism that GRAMPS can capture with automatic instancing. GRAMPS stages are useful when the benefits of coherent execution and/or greater parallelism outweigh the costs of enqueuing and dequeuing the data.

GRAMPS supports three types of stages that correspond to distinct sets of computational characteristics. A stage's type serves as a hint facilitating work assignment, resource allocation, and computation scheduling. We strove for a minimal number of simple abstractions that still captured the key execution models for mapping well to many-core platforms. In addition to a general purpose Thread stage, we included two others: stages implemented as fixed-function/dedicated hardware processing and data-parallel Streaming/GPGPU-style Shader stages.

A GRAMPS stage definition consists of:

- Type: Either Thread, Fixed-Function, or Shader.

- Program: Either program code for a Thread/Shader or configuration parameters for a fixed-function unit.

- Buffers: Random-access, fixed-size data bindings.

- Queues: Input, Output, In-Place, and "Push"/"Coalescing" queue bindings.

We expect GRAMPS programs to run on platforms with significantly larger numbers of processing resources than stages. Thus, when possible, GRAMPS will execute multiple copies of a stage's program in parallel (each operating on distinct input

packets). We refer to each executing copy of a stage as an *instance* while stages that require serial processing (e.g. initialization) execute as *singletons*. Different graphs allow different amounts of instancing. Recall that in Figure 3.2, Scoop Cookies can be freely instanced (but stuck serializing afterwards) whereas in Figure 3.3 the combined Scoop & Lay Down can be instanced only as much as separate subqueues have input (i.e., separate cookie sheets are available).

### 3.5.1 Thread Stages

Thread stages are best described as traditional CPU-style threads. They are designed for task-parallel, serial, and other regions of an application best suited to large per-element working sets or operations dependent on multiple packets at once (e.g., repacking or re-sorting of data). They are expected typically to fill one of two roles: repacking data between Shader stages, or processing bulk chunks of data where sharing/reuse or cross-communication make data-parallel Shaders inefficient. Importantly, since Thread stages may be stateful—i.e., may retain state internally as they process packets—they usually must be manually parallelized/instanced by the application rather than automatically by GRAMPS. There is one situation in which GRAMPS can instance a Thread stage: if its sole input is a queue set and all of the stage's processing is per-subqueue, then the application can flag the stage and GRAMPS will instantiate one copy for each input subqueue.

Thread stage instances manipulate queues via two GRAMPS intrinsics—`reserve`, and `commit`—which operate in-place. Each takes four arguments: the queue, the number of packets, optional flags, and an optional subqueue identifier (either a dense index or a sparse key). `reserve` returns the caller a "window" that is a reference to one or more contiguous packets. GRAMPS guarantees the caller exclusive access to this region of the queue until it receives a corresponding `commit` notification. An input queue `commit` indicates that packet(s) have been consumed and can be reclaimed. Output queue `commit` operations indicate the packet(s) are now available for downstream stages.

`reserve` is a potentially blocking operation: an instance will be suspended on an

input reservation if there is no data ready or an output reservation if the queue is currently at its specified capacity. Both `reserve` and `commit` are potential preemption points where a scheduler can choose to suspend an instance if a higher priority instance is runnable.

The queue `reserve-commit` protocol allows stages to perform in-place operations on queue data and allows GRAMPS to manage queue access and underlying storage. Queue windows permit various strategies for queue implementation and add a degree of indirection that enables customized implementations for systems with distributed address spaces, explicit prefetch, or local store capabilities.

### 3.5.2   Fixed-Function Stages

GRAMPS allows stages to be implemented by fixed-function or specialized hardware units. GRAMPS effectively treats them as Thread stages with peculiar internals: they inter-operate with the rest of GRAMPS via queue reservations and commitments and cannot be instanced. Applications configure these units via GRAMPS by providing hardware-specific configuration information at the time of stage specification.

### 3.5.3   Shader Stages

Shader stages define short-lived, run-to-completion computations applied independently to every input packet, akin to traditional GPGPU shaders/kernels. They are designed as an efficient mechanism for running data-parallel regions of an application. GRAMPS leverages these properties in two ways: *automatic instancing* and *automatic queue management*. Since Shaders are stateless across packets, GRAMPS can freely create up to as many concurrent instances as there are available packets to ensure large amounts of parallelism.

Additionally, GRAMPS manages queue inputs and outputs for Shader instances automatically, which simplifies Shader programs and allows the scheduler to guarantee they can run to completion without blocking. As input packets arrive in a queue, GRAMPS internally makes corresponding output packet reservations. Once the paired reservations are obtained, GRAMPS runs the packet's worth of Shader

instances. Each instance receives a reference to the shared packet header and to one element in each of the pre-reserved input and output packets (recall that Shader stages use Collection packets, as described in Section 3.4.1). When all of the instances have completed, GRAMPS internally commits the inputs and outputs. With ordered queues, pre-reserving Shader packets also lets GRAMPS ensure that the commit order of output packets corresponds exactly to the order of inputs, and preserve order across stages. Input and output pre-reservation reflects GRAMPS Shaders' GPGPU antecedents, but, like allowing cycles in execution graphs, GRAMPS Shaders have an additional optional operation, `push`, that significantly extends their flexibility. `push` dynamically inserts elements into *unordered* output queues, thus allowing conditional output. Note that unlike `reserve` and `commit`, and in sync with the data-parallel nature of Shader stages, `push` sends individual elements which GRAMPS accumulates into Collection packets. It coalesces as full a packet as possible, sets the element count in the header, and enqueues the packet for downstream consumption.

Push-style coalescing applied to an input queue allows a Shader stage to perform reductions. In this mode, GRAMPS merges elements from partially full input packets to provide the consuming instance with as full a packet as possible. In turn, the consumer compacts/combines all of the elements in the packet to a single result. An application graph can route this once-filtered data downstream or it can bind the input queue "in-place". With an "in-place" binding, the Shader performs a full parallel reduction: compacted output packets are themselves recoalesced into full packets and recirculated down to a single, final result, which GRAMPS propagates downstream as if `push`'ed.

## 3.6 Common Design Idioms

In our experience building GRAMPS applications, a few application graph constructs emerged as common building blocks and important cases to consider when tuning a GRAMPS implementation:

- Task generator → Shader consumer: A singleton stage subdivides an input range or read-only buffer into (offset, length) pairs to be processed by a Shader.

Important traits are distributing consumer work to idle processors before preempting the singleton producer and a very simple producer that outputs packets rapidly.

- Shader producer → Shader consumer: The canonical GRAMPS pipeline example: a work-rich instanced producer feeding a freely instanced consumer. The important trait is balancing the resources assigned to each stage. Aggressively preempting producer instances with consumers keeps the queue footprint small, but frequently runs out work and must switch back to producers. Too many context switches, especially rapid ping-ponging, can be expensive.

- Queues as barriers: While GRAMPS applications tend to exhibit producer-consumer parallelism between stages, sometimes an application needs to block one stage until its upstream is entirely done. It does this by issuing an unsatisfiable `reserve`, generally for -1 packets. This leaves the stage unrunnable until all producers on that queue have completed, at which point GRAMPS returns a short reservation. The important trait is obvious: the blocked stage will not run until all of its input is available. This can influence, for example, decisions on when to wait to coalesce a more full packet versus flush it or when to even check if a stage can be scheduled.

# Chapter 4

# Future Rendering Architectures

This chapter presents a case study of GRAMPS as a programming model for future graphics architectures. We examine a rasterizing and a ray-tracing renderer as well as a hybrid of the two on two simulated platforms: one more heterogeneous and GPU-like and the other more CPU-like. Most of the content of this chapter was published separately in Transactions on Graphics, January 2009 [41].

## 4.1    Introduction

Rendering is the first application domain where we examine GRAMPS. Not only was GRAMPS highly influenced by the graphic pipeline, but rendering is a critical 'table-stakes' area for parallelism, especially heterogeneous and producer-consumer parallelism: the past decade-plus has seen the rise of commodity GPUs with enormous computational power and the ability to render complex, high-resolution scenes in real time using Z-buffer rasterization. At the same time, the real-time photorealistic rendering problem is not solved, and there remains interest in exploring advanced rendering algorithms such as ray tracing, REYES, and combinations of these with the traditional graphics pipeline. Implementing them in the confines of current GPU programming models is awkward and nonperformant (e.g., [20, 46]), however, but GRAMPS potentially fits well.

While the earliest GPUs were simple, application-configurable engines, the history

of high-performance graphics over the past three decades has been the co-evolution of a pipeline abstraction (the traditional graphics pipeline) and the corresponding driver/hardware devices (GPUs). In the recent past, the shading stages of the pipeline became software programmable. Prior to the transition, developers controlled shading by toggling and configuring an assortment of fixed options and parameters, but the widespread innovation in shading techniques led to an increasingly complex matrix of choices. In order to accommodate the trend towards more general shading, researchers and then graphics vendors added programmable shading to the graphics pipeline.

We see an analogy between the evolution from fixed to programmable shading and the current changes for enabling and configuring pipeline stages. After remaining static for a long time, there are a variety of new pipeline topologies available and under exploration. Direct3D 10 added new geometry and stream-output stages [5]. The Xbox 360 added a new stage for tessellation (also included in the latest iterations of Direct3D and in OpenGL via extensions). We believe that future rendering techniques and increasing non-graphical usage will motivate more new pipeline stages and configuration options. As was true with pre-programmable shading, these new abilities are currently all delivered as predefined stage and pipeline options to be toggled and combined. Looking forward, we propose to instead expose the hardware capabilities via GRAMPS and programmatically construct graphics pipelines as GRAMPS applications.

This is not an (unthinkably) radical change. In fact, it formalizes and exposes the key elements of current GPU hardware: FIFOs and work-buffers for flow control and accumulating work, shader cores for data-parallel processing, and custom fixed-function units for other stages, such as rasterization. Also, as suggested above, it mirrors the multi-configuration to programmable transition in shading now applied to the pipeline topology itself. Finally, the notion of 'GPU' itself is changing. At the extreme is Intel's Larrabee [40], an entirely alternate architecture, but even recent generations of conventional GPUs are being described and designed as more general throughput-oriented architectures.

In this chapter, we examine how GRAMPS holds up as a programming model for rendering on future graphics hardware. Specifically, we survey current graphics

architectures and programming models and then assess GRAMPS in terms of the design goals from Chapter 1:

- **Broad application scope**: An OpenGL/Direct3D-like rasterization pipeline, a ray tracing graph, and an extension for the rasterization pipeline to incorporate ray traced effects.

- **Multi-platform applicability**: GRAMPS runtimes for two simulated architectures: a more general CPU-like system and a more custom GPU-like one.

- **Performance**: Scheduling logic for the two runtimes that demonstrate high scale-out utilization with good queue footprint management.

- **Tunability**: Grampsviz, an application for visualizing the execution of our renderers and descriptions of some application graph improvements that had major effects.

## 4.2 Background and Related Work

### 4.2.1 GPUs

As mentioned above, modern GPUs have become very sophisticated high-performance platforms for real-time graphics. Two widely available examples are NVIDIA's 400-Series [32] and AMD's HD 5000-Series [2], both built around a pool of highly multi-threaded programmable cores tuned to sustain multiple teraflops of performance when performing shading computations. They deliver additional computing capabilities via fixed-function units that perform tasks such as rasterization, texture filtering, and frame-buffer blending. While the programmable cores are exposed for general purpose workloads, much of this non-programmable hardware is central to how GPUs schedule *graphics* and implement the graphics pipeline so efficiently. As a result, GPUs constitute a heavily-tuned platform for rasterization-based z-buffer rendering but offer only limited benefits for alternative graphics algorithms.
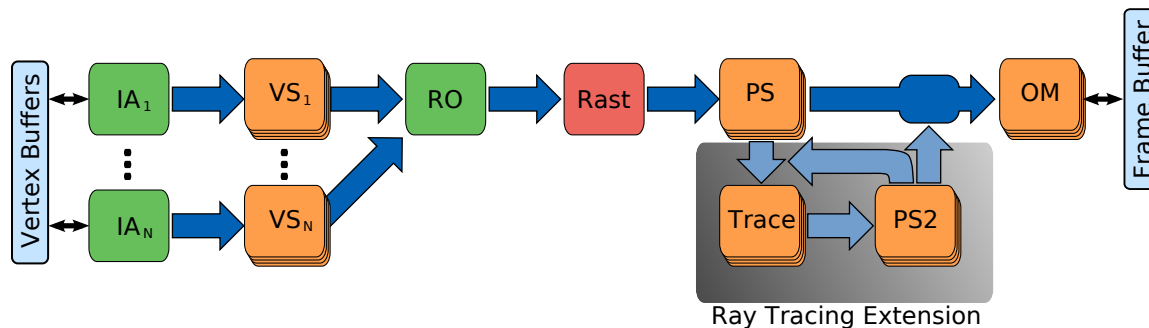
Figure 4.1: Our Direct3D-based pipeline including optional extension.

## 4.2.2   Graphics on Stream Processors

Two different prior efforts explored rendering with stream processors/streaming languages: one on Imagine and one on Raw [34, 10]. Both formulated OpenGL-like pipelines as stream applications much like a GRAMPS application graph. Unlike GRAMPS, however, both suffered from needing to constrain (fake) the dynamic irregularity of rendering in predictable terms for their static scheduling.

Both systems redefined and recompiled their pipelines per scene and per frame they rendered. Not only that, as part of tuning and scheduling each scene-specific pipeline they needed to pre-render each frame once to gather statistics first. In the six-plus years since the streaming pipelines were first published, this irregularity and scene-dependent variation has become even more prominent: branching in shaders is routine, as is composing final frames from large numbers of off-screen rendering passes.

Around the same time, Purcell demonstrated a somewhat reverse accomplishment: using the programmable elements of a GPU as a stream processor and performing ray tracing [36]. Stuck inside the Breadth-First-like environment of the graphics pipeline's shading stages, this system and its successors always struggled with load-balancing [14, 20].
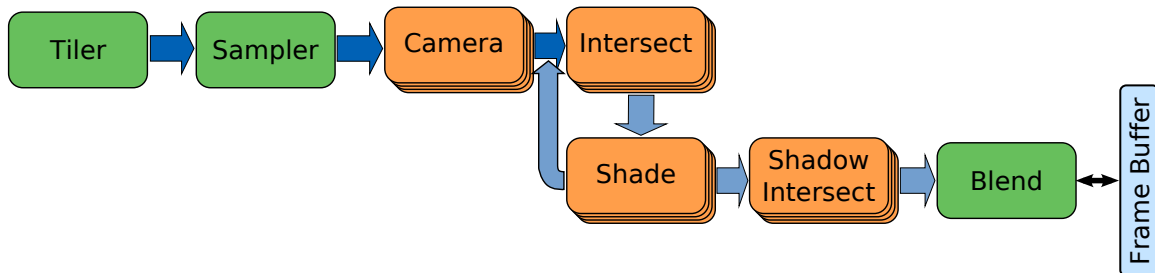
Figure 4.2: Our ray tracing application graph. Note that the queue from Shade back to Intersect (for secondary rays) makes the graph cyclic.

## 4.3 Application Scope

In this section we describe three example rendering systems framed in terms of GRAMPS: a simplified Direct3D pipeline, a packet-based ray tracer, and a hybrid that augments the simplified Direct3D pipeline with additional stages used for ray tracing.

### 4.3.1 Direct3D

The GRAMPS graph corresponding to our sort-last formulation of a simplified Direct3D pipeline is shown in Figure 4.1. A major challenge of a Direct3D implementation is exposing high levels of parallelism while preserving Direct3D fragment ordering semantics.

The pipeline's front-end consists of several groups of Input Assembly (IA) and Vertex Shading (VS) stages that operate in parallel on disjoint regions of the input vertex set. Currently, we manually create these groups—built-in instancing of subgraphs is a potentially useful future addition to GRAMPS. Each IA/VS group produces an ordered stream of post transform primitives. Each input is assigned a sequence number so that these streams can be collected and totally-ordered by a singleton Reorder (RO) stage before being delivered to the fixed-function rasterizer (Rast).

The pipeline back-end starts with a Pixel Shader (PS) stage that processes fragments. After shading, fragment packets are routed to the appropriate subqueue in

the output queue set based on their screen space position, much like described in Section 3.4.2. The queue set lets GRAMPS instance the Output Merger while still guaranteeing that fragments are blended into the frame buffer atomically and in the correct order. Note that Rast facilitates this structure by scanning out packets of fragments that never cross the screen space routing boundaries. Notice that the Direct3D graph contains no stages that correspond to fixed-function texture filtering. While a GRAMPS implementation is free to provide dedicated texturing support (as modern GPUs do through special instructions), special-purpose operations that occur within a stage are considered part of its internal operation, not part of the GRAMPS programming abstraction or any GRAMPS graph.

### 4.3.2   Ray Tracer

Our implementation of a packet-based ray tracer resembles Purcell's streaming implementation [36] and maps natural components of ray tracing to GRAMPS stages (Figure 4.2). With the exception of Tiler, Sampler, and Blend, whose performance needs are satisfied by singleton Thread stages, all graph stages are instanced Shader stages. All queues in the packet tracer graph are unordered.

A ray tracer performs two computationally expensive operations: ray-scene intersection and surface hit point shading. Considered separately, each of these operations is amenable to wide SIMD processing and exhibits favorable memory access characteristics. Because recursive rays are conditionally traced, SIMD utilization can drop severely if shading directly invokes intersection [6].

Our implementation decouples these operations by making Intersect, Shadow Intersect, and Shade separate graph stages. Thus, each of the three operations executes efficiently on batches of inputs from their respective queues. To produce these batches of work, the ray tracer leverages the GRAMPS queue `push` operation. When shading yields too few secondary rays to form a complete packet, execution of Intersect (or Shadow Intersect) is delayed until more work is available. Similarly, if too few rays from Intersect need shading, they won't be shaded until a sufficiently large batch is available. This strategy could be extended further using more complex GRAMPS

graphs. For example, separating Intersect into a full subgraph could allow for rays to be binned at individual BVH nodes during traversal.

Lastly, the ability to cast ray tracing as a graph with loops, rather than a feed-forward pipeline allows for an easy implementation of both max-depth ray termination and also ray tree attenuation termination by tracking depth/attenuation with each ray [17]. While reflections to a fixed maximal depth could also be modeled with a statically unrolled pipeline, this is an awkward implementation strategy and does not permit ray tree attenuation.

### 4.3.3 Extended Direct3D

By constructing execution graphs that are decoupled from hardware, GRAMPS creates an opportunity for specialized pipelines. Our third renderer extends the Direct3D pipeline to form a new graph that adds ray-traced effects (Figure 4.1, including the shaded portion). We insert two additional stages, Trace and PS2, between PS and OM and allow Extended Direct3D Pixel Shaders to `push` rays in addition to performing standard local shading. Trace performs packetized ray-scene intersection and pushes the results to a second shading stage (PS2). Like PS, PS2 is permitted to send its shaded output to OM, or generate additional rays for Trace (the Extended Direct3D graph contains a loop). We introduced PS2 as a distinct stage to retain the ability to specialize PS shading computations for the case of high coherence (fragments in a packet from Rast all originate from the same input triangle) and to separate tracing from shading as explained above.

There are two other important characteristics of our Extended Direct3D renderer. Our implementation uses a pre-initialized early-Z buffer from a prior Z-only pass to avoid unnecessary ray-scene queries. In addition, early-Z testing is required to generate correct images because pixel contributions from the PS2 stage can arrive out of triangle draw order (input to PS2 is an unordered `push` queue).

Note that while this example uses `push` only for the purposes of building ray and shading packets, other natural uses include handling fragment repacking when coherence patterns change, or as a mechanism for efficiently handling a constrained

form of data amplification or compaction.

## 4.4   Multi-platform

### 4.4.1   Hardware Simulator

We developed a machine simulator to conduct our evaluation of GRAMPS as a programming model for rendering.  We chose to use a simulator instead of existing hardware for three reasons: access, flexibility, and instrumentability. Low-level GPU programming access is highly proprietary and minimally documented.  Even with willing vendor collaboration, drivers for modern graphics hardware are enormously complex; customizing them to our ends would be very challenging.  Additionally, many of the features we wish to explore, for example, `push` and Thread stages, are simply unavailable in current hardware.  Finally, one of the most important aspects of validation and bring-up, at least for a completely new programming model, is the ability to report a wide variety of system properties.  With real hardware, it can be invasive and challenging, if not impossible, to measure a consistent snapshot of system-wide state. In simulation, by contrast, it is straightfoward.

Our simulator is single-threaded (though it simulated the execution of many parallel threads). While this makes simulation slower, it enormously simplifies maintaining such things as a shared, global clock and deterministic execution. Essentially, on every simulated clock, the main simulation loop runs the top-level Tier-N scheduler (see below), and steps each core and any fixed function units until the GRAMPS runtime detects application completion or deadlock (in the case of flawed application graphs that contain loops).

The basic execution resource is a simulated, multi-threaded programmmable core, referred to as an XPU.  XPUs use a MIPS64 instruction set architecture [27] extended with a custom 16-wide SIMD vector instruction set that includes basic math and gather/scatter load/store operations.  They can select and execute one hardware thread per clock. We develop XPU programs using cross-compiling versions of GCC and GNU Binutils that we extended to support our vector instructions.  The

core simulation supports many parameters, the most important being the number of execution contexts (hardware threads) and the two latency factors.

Latency in XPU cores is modeled very simply. A core has two values—an 'ALU latency' and a 'memory latency'. The ALU latency is the number of cycles a vector instruction takes to retire. It is applied per-hardware thread, on the presumption of pipelining and/or per-thread resources. That is, a four-threaded XPU with a four cycle ALU latency, for example, can complete one vector instruction per cycle if all of its thread slots are active. Memory latency is also a single value. It functions as the overall (hypothetical) memory hierarchy and is applied to all data accesses (instruction accesses are single cycle). These simple latencies are gross approximations, but we believe them sufficient for assessing GRAMPS's high-level plausibility. One natural area of future research, that is already under initial investigation, is enriching instruction issue and the memory hierarchy so that they can mimic the behaviors of a variety of existing CPU and GPU designs.

In our experiments, we used two different collections of settings for cores. XPU *fat cores* are general-purpose cores optimized for throughput. They are in-order, four-threaded processors with both an ALU and memory latency of four cycles. XPU *micro cores* are intended to resemble GPU shader cores and execute efficiently under the load of many lightweight threads. Each micro core has 24 independent hardware thread execution slots. Thus, with 16-element-at-a-time vector instructions, each micro core is capable of hardware-interleaved execution of 384 Shader instances. However, reflecting their heavy emphasis on longer serial latencies hidden by multi-threading, they are configured with an ALU latency of six cycles and a memory latency of 100 cycles.

To reflect the heterogeneous nature of graphics hardware, and to exercise the Fixed-Function stage support in GRAMPS, we also implemented a simulated hardware rasterizer that can be enabled optionally, as shown by the Direct3D graph (Section 4.3.1). It employs the scan conversion algorithm from [33] and can rasterize an entire triangle in a single simulated clock, to mimic highly optimized custom functional units.
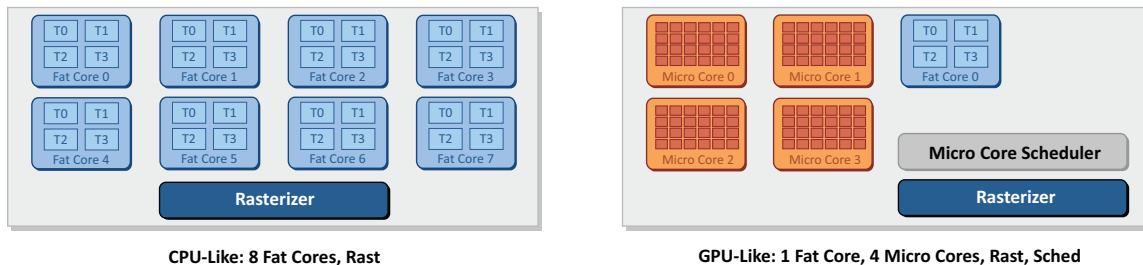
Figure 4.3: The CPU-like and GPU-like simulator configurations: different mixtures of XPU fat (blue) and micro (orange) cores plus a fixed function rasterizer. Boxes within the cores represent hardware thread slots.

We chose two specific configurations of our simulation environment for the hypothetical graphics architectures in our experiments. They are shown in Figure 4.3. The *GPU-like* configuration contains one fat core, four micro cores with dedicated Shader scheduling/dispatch support (see below), and a fixed-function rasterizer. As the name indicates, It is envisioned as an evolution of current GPUs. The *CPU-like* configuration consists of the rasterizer plus eight fat cores, mimicking a more general purpose many-core implementation. This choice of machine configurations allows us to explore two GRAMPS scheduler implementations employing different levels of hardware support.

## 4.4.2 GRAMPS Runtimes

We built implementations of GRAMPS for the two configurations directly into the simulator (as opposed to running them as simulated code). In addition to implementing the programming model, they function as mini-operating systems: application code calls GRAMPS via the MIPS `syscall` instruction and GRAMPS interacts directly with the XPU logic to, for example, launch and context-switch hardware threads. Their primary nontrivial run-time components are their queue implementation, dispatch of Shader instances, and scheduling. The first two are discussed below and scheduling is discussed in detail as part of performance (Section 4.5.1).

Both simulated GRAMPS runtimes share the same queue implementation: one fixed-sized circular buffer per application queue, sized to the capacity specified in

the application graph. Ordered queues use the buffer as a rolling FIFO: there are head and tail pointers, each with separate 'reserve' and 'commit' marks. These are updated on `reserve` and `commit`, respectively, and wrap around when they reach the end of the buffer. The region between the head 'reserve' and 'commit' reflects pending output reservations (packets being produced) and the region between the tails marks reflects pending input reservations (packets being consumed). The entries between the tail 'reserve' and head 'commit' are packets available for consumption and the space between the head 'reserve' and tail 'commit' is empty. Unordered queues are simpler. They have two bitvectors: one indicating buffer entries that contain packets available for input and one tracking entries with packets available for output.

The two runtimes implement Shaders quite differently. The GPU-like version is simple. Recall from Figure 4.3 that its micro cores are dedicated units for running Shader stages. Thus, its version of GRAMPS launches Shader instances 'in hardware': it calls the XPU code to create and reap hardware contexts as needed. The CPU-like version, however, implements Shader instancing 'in software'. It keeps a pool of privileged, internal meta-Thread stages, called 'Dispatch stages'. When the scheduler decides to schedule Shader work, it records the stage information and pre-reservation in a data structure we call 'the scoreboard' and marks an idle Dispatch stage as runnable. When the Dispatch stage runs, it reads the scoreboard entry, confirms its pre-reservation, executes the Shader kernel (just a call through a function pointer), and post-commits. After it finishes, the Dispatch stage queries the runtime to determine whether to process another scoreboard entry or resume idling.

We run the renderers by linking them against a library with the combined simulator-runtime. The GRAMPS entry points that define the application graph run natively (i.e., outside the simulator). Then, the call to launch the execution graph mirrors a device driver: it bootstraps the simulated hardware; passes it the stages, queues, etc.; and starts (simulated) asynchronous execution.
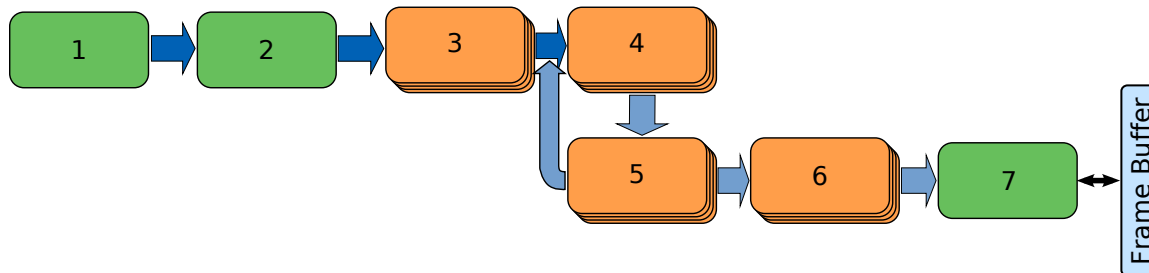
Figure 4.4: The static stage priorities for the ray tracing application graph. In general, each consumer is higher priority than its producer.

## 4.5   Performance

### 4.5.1   Scheduling

Recall that the goal of the scheduler in any GRAMPS implementation is to maximize scale-out parallelism, or machine utilization, while keep working sets, or queue depths, small. Specifically, it seeks to synthesize at run-time what streaming systems arrange during up-front compilation: aggregated batches of parallel work with strong data locality. The queues of the computation graphs are intended to delineate such coherency groupings. The GRAMPS scheduler then balances the need to accumulate enough work to fill all available cores against the storage overheads of piling up undispatched packets and the computational overhead of making frequent scheduling decisions. Note that compared to a native GPU or other single pipeline-specific schedule GRAMPS's generality creates a significant scheduling disadvantage: GRAMPS lacks semantic knowledge of—and any scheduling heuristics based on—stage internals and the data types passed between them. The GRAMPS abstractions are designed to give an implementer two primary hints to partially compensate: the topology of the execution graph, and the capacity of each queue.

Our scheduling algorithm for both simulated configurations assigns each stage a static priority based upon its proximity in the graph to sink nodes (stages with no output queues) and distance from source nodes (stages with no input queues), as shown in Figure 4.4. Specifically, just before execution, the runtime recursively traverses the execution graph depth-first (from each stage that is initially marked

runnable), marking each stage it visits. It increments the priority with each level it descends and unwinds when it encounters a sink or a stage that has already been visited. In a pipeline or DAG, this gives the start(s) lowest weight and the end(s) highest weight which predisposes the scheduler towards draining work out of the system and keeping queue depths small.

Additionally, the scheduler maintains an 'inspect' bitvector containing a field for each graph stage. A stage's inspectable bit is set whenever a new input packet is available or output packet is consumed (in case it was blocked on a full output queue). Its bit is cleared whenever the scheduler next inspects the stage and either dispatches all available input or determines the stage is not truly runnable.

Within the simulator, our scheduler is organized hierarchically in 'tiers'. The top tier (Tier-N) has system-wide responsibilities for notifying idle cores and fixed-function units when they should look for work. The cores themselves then handle the lower levels of scheduling.

**Fat Core Scheduling**

Since we intended fat cores to resemble CPUs and be suitable for arbitrary threads, the GRAMPS fat-core scheduling logic is implemented directly in software. It is organized as a fast, simple Tier-0 that manages a single thread slot and a more sophisticated Tier-1 that is shared per-core.

Tier-1 updates and maintains a prioritized work list of runnable instances based upon the inspect bitvector. It also groups Shader instances for launch and issues the implicit `reserve` and `commit` operations on their behalf. Tier-1 runs asynchronously at a parameterized period (one million cycles in our experiments). However, if a Tier-0 scheduler finds the list of runnable instances empty (there is no work to dispatch), Tier-0 will invoke Tier-1 before idling its thread slot.

Tier-0 carries out the work of loading, unloading, and preempting instances on individual thread slots. It makes no "scheduling" decisions other than comparing the current thread's priority to the front of the work list at potential preemption points. For Thread stages, preemption points include queue manipulations and terminations. For Shader stages, preemption is possible only between instance invocations.

|            | Triangles | Fragments/Tri |
|------------|-----------|---------------|
| Teapot     | 6,230     | 67.1          |
| Courtyard  | 31,375    | 145.8         |
| Fairy      | 174,117   | 36.8          |

Table 4.1: Test scenes. Courtyard uses character models from Unreal Tournament 3. Fairy is a complex scene designed to exercise modern ray tracers.

**Micro Core Scheduling**

In the GPU-like configuration, all Shader work is run on micro cores. Similar to current GPU designs, micro cores rely on a hardware-based scheduling unit to manage their numerous simple thread contexts (see Figure 4.3). This unit is functionally similar to combined fat-core Tier-1 and Tier-0's with two significant differences: a single hardware Tier-1 is shared across all micro cores, and it is invoked on demand at every Shader instance termination rather than asynchronously.

When data is committed to Shader queues, the micro-core scheduler identifies (in order of stage priority) input queues with sufficient work, then pre-reserves space in the corresponding stage's input and output queues. It associates this data with new Shader instances and assigns the instances to the first unused thread slot in the least-loaded micro core. When Shader instances complete, the scheduler commits their input and output data, then attempts to schedule a new Shader instances to fill the available thread slot. The micro-core scheduler also takes care of coalescing elements generated via `push` into packets.

## 4.5.2   Evaluation

We chose three test scenes (Table 4.1) and rendered them at 1024 with each of our renderers on our two (simulated) hardware configurations. The ray tracer casts shadow

rays and one bounce of reflection rays off all surfaces. Extended Direct3D casts only shadow rays. The scenes vary in both overall complexity and distribution of triangle size, requiring GRAMPS to dynamically balance load across graph stages. Overall, we were surprised at how effective our unsophisticated scheduling proved.

As explained in the discussion of scheduling, our primary focus is our implementations' abilities to find parallelism and to manage queues (especially in the context of loops and the use of `push`). To evaluate this, we measure the extent to which GRAMPS keeps core thread execution slots occupied with active threads, and the depth of queues during graph execution.

The simulation introduces two simplifying assumptions: First, although our implementations seek to minimize the frequency at which scheduling operations occur, we assign no cost to the execution of the GRAMPS scheduler or for possible contention in access to shared queues. Second, as described above, we incorporate only a simple memory system model—a fixed access time of four cycles for fat cores and 100 cycles for micro cores. Given these assumptions, we use thread execution-slot occupancy as our performance metric rather than ALU utilization (ALUs may be underutilized due to memory stalls or low SIMD efficiency even if a slot is filled). Slot occupancy is convenient because it directly reflects the scheduler's ability to recognize opportunities for parallelism. At the same time, it is less dependent on the degree of optimization of Thread/Shader programs—which is not a need unique to GRAMPS nor a highly optimized aspect of our application implementations.

Table 4.2 summarizes the overall simulation statistics. Note that on GPU-like configurations, we focus on the occupancy of the micro cores that run Shader work (the fat core in the GPU-like configuration is rarely used as our graphs perform a majority of computation in Shaders).

Both GRAMPS implementations maintained high thread-slot occupancy with all of the renderers. With the exception of rendering the Fairy using Direct3D, the GRAMPS scheduler produced occupancy above 87% (small triangles in the Fairy scene bottle-neck the pipeline in RO limiting available parallelism—see GPU-like fat core occupancy in Table 4.2).

| | | CPU-like Configuration | | GPU-like Configuration | | |
|---|---|---|---|---|---|---|
| | | Fat Core Occup (%) | Peak Queue Size (KB) | Fat Core Occup (%) | Micro Core Occup (%) | Peak Queue Size (KB) |
| **Teapot** | D3D | 87.8 | 510 | 13.0 | 95.9 | 1,329 |
| | Ext. D3D | 90.2 | 582 | 0.5 | 98.8 | 1,264 |
| | Ray Tracer | 99.8 | 156 | 3.2 | 99.9 | 392 |
| **Courtyard** | D3D | 88.5 | 544 | 9.2 | 95.0 | 1,301 |
| | Ext. D3D | 94.2 | 586 | 0.2 | 99.8 | 1,272 |
| | Ray Tracer | 99.9 | 176 | 1.2 | 99.9 | 456 |
| **Fairy** | D3D | 77.2 | 561 | 20.5 | 81.5 | 1,423 |
| | Ext. D3D | 92.0 | 605 | 0.8 | 99.8 | 1,195 |
| | Ray Tracer | 100.0 | 205 | 0.8 | 99.9 | 537 |

Table 4.2: Simulation results: Core thread-slot occupancy and peak queue footprint of all graph queues.

Our emulations maintained high occupancy while keeping worst-case queue footprint low. In all experiments queue sizes remained small enough to be contained within the on-chip memories of modern processors. The ray tracer, despite a graph loop for reflection rays and heavy use of `push`, had by far the smallest queue footprint. This was the direct result of not needing ordered queues. With ordering enabled, when instances complete out of order, as happens from time to time, GRAMPS cannot make their output available downstream or reclaim it until the missing stragglers arrive.

## 4.6   Tunability

While the GRAMPS renderers we present performed well, our experiences indicated that understanding how a given formulation executes and navigating among alternatives can make a big difference.

### 4.6.1   Diagnosis

The fact that our implementations ran inside a simulation allowed us to measure a great deal of raw statistics. While they were highly useful, they were also unwieldy for
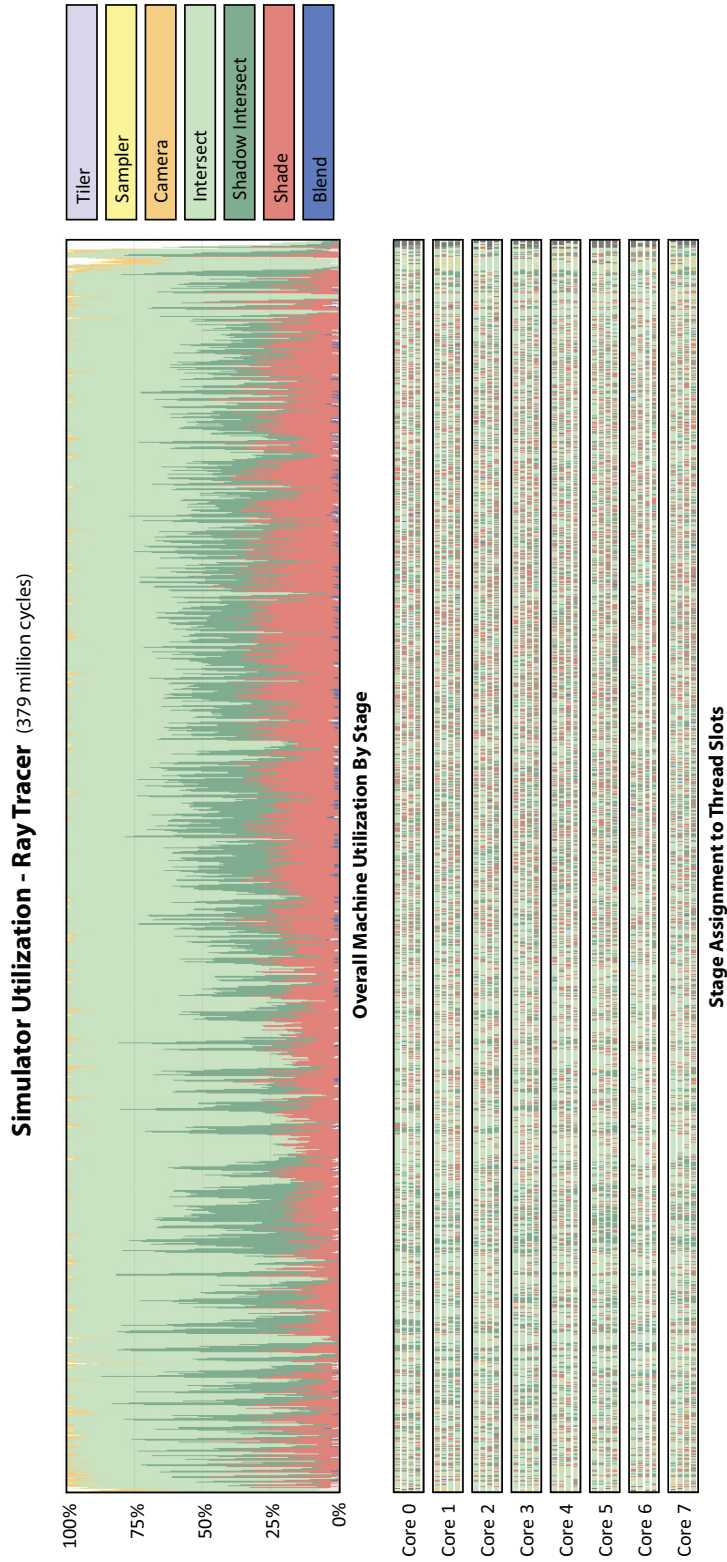
Figure 4.5: A grampsviz visualization of ray tracing the Teapot scene on the CPU-like configuration. The bottom shows the dynamic mapping of stage instances onto hardware threads.
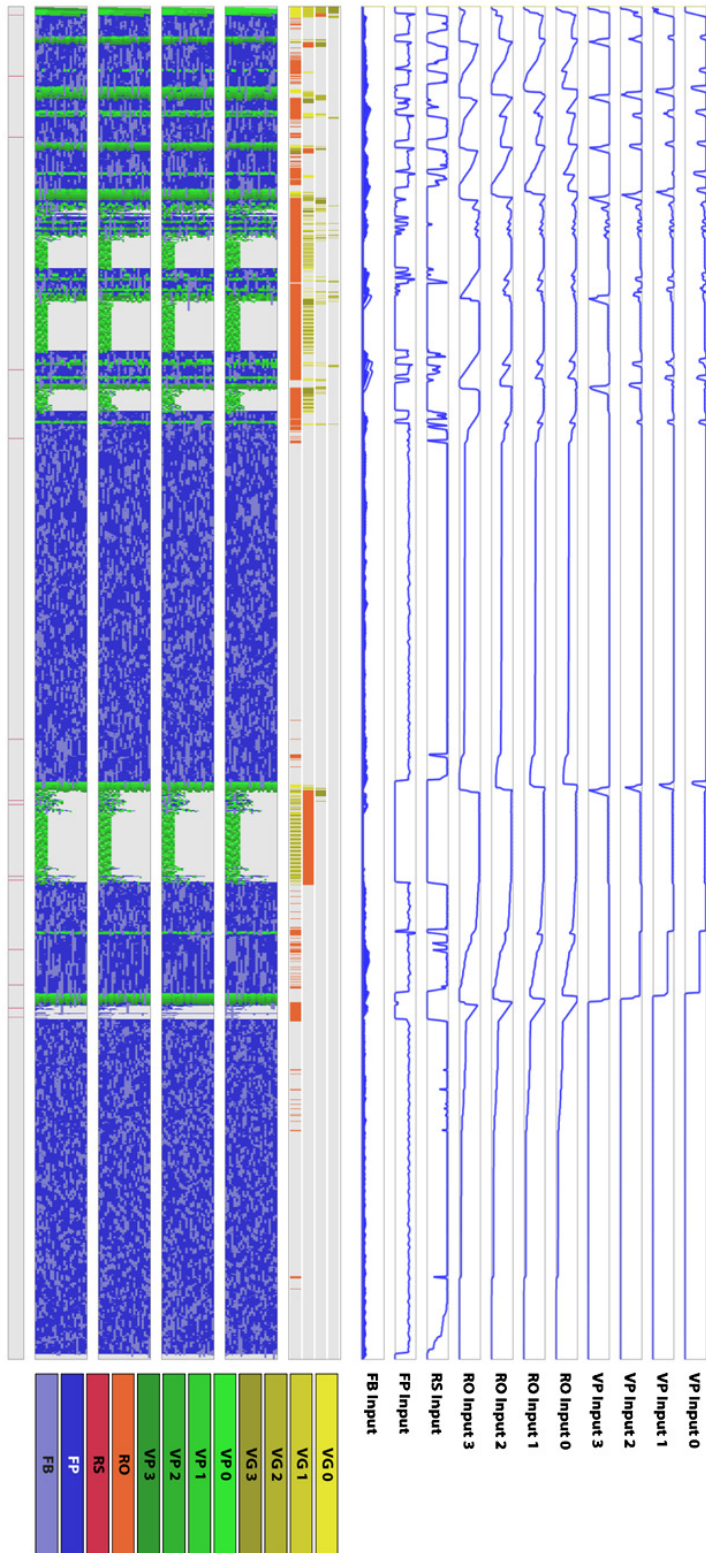
Figure 4.6: A grampsviz visualization of rasterizing the Courtyard scene on the GPU-like configuration. The top shows the queue depths and the bottom the mapping of instances onto Fat and Micro core threads.
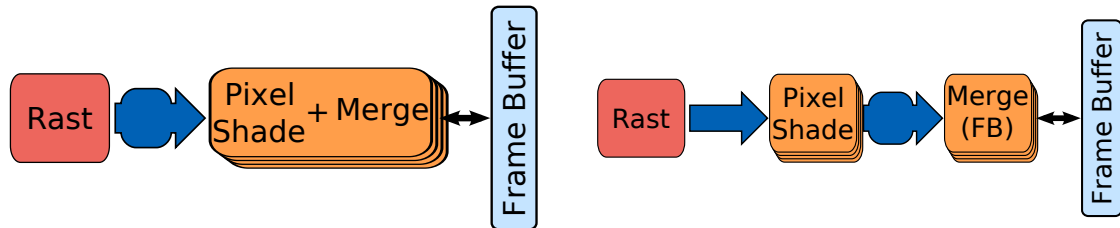
Figure 4.7: Initial and revised versions for the bottom of our Direct3D pipeline.

understanding high-order behavior and would not be as applicable in a real system.

To that end, we instrumented the GRAMPS runtimes to record their scheduling decisions and queue operations and built a tool called *grampsviz* that let us visually navigate the output. It can show three different timeline views: a per-core display of which stage is resident, a stacked display of how much of the whole system is dedicated to each stage, and a per-queue display of how many packets are available for `reserve` and `commit`. Cross-referencing these views show us, for example, which queues accumulate data or which singleton stages run at times when no other work is available, both of which indicate likely places to improve the application graph design, the scheduler, or both. Figures 4.5 and 4.6 show two examples of using grampsviz views that summarize an entire workload execution. It can also be used interactively to zoom the time window as far down as individual queue or scheduling operations.

## 4.6.2 Optimization

The GRAMPS concepts/interfaces permit designers to create graphs that do not run well, and even good graphs can profit from tuning. For example, our initial Direct3D graph—which used a single Shader stage to handle both PS and OM—exhibited a large queue memory footprint (see Figure 4.7).

Although our first Direct3D graph used a queue set to respect OM ordering requirements while still enabling parallel processing of distinct screen space image tiles, this formulation caused all PS work for a single tile to be serialized. Thus, the graph suffered from load imbalance (and corresponding queue backup) when one tile—and thus one subqueue— had a disproportionate number of fragments. Separating PS

and OM into unique graph stages and connecting these stages using a queue set allowed shading of all fragments—independent of screen location—to be performed in parallel. This modification reduced queue footprints by over two orders of magnitude.

Similarly, in the ray tracer, limiting the maximum depth of the queue between the sampler and the camera while leaving the others effectively unbounded reduced the overall queue footprint by more than an order of magnitude.

In the same vein, although the general graph structure is the same across our two simulation configurations, we made slight tuning customizations as a function of how many machine thread slots were available. In the GPU-like configuration of Direct3D/Extended Direct3D, we increased the number of OM input subqueues to enable additional parallelism. We also set the capacities on several critical Direct3D queues and, as mentioned above, the ray tracer's sample queue based on the maximum number of machine threads.

## 4.7   Conclusion

This case study has introduced the first validation of GRAMPS: successful formulation of a conventional rasterization pipeline as an application while also enabling extensions and, in the case of the ray tracer, outright replacement. Despite enabling this flexibility, our prototypes performed well in simulation. The application graphs were able to expose/our GRAMPS runtimes were able to capture high parallel utilization. At the same time, with static priorities and basic implementations, GRAMPS kept the queue footprints, i.e., necessary inter-stage buffering, to very practical levels. Even the order sensitive, and thus footprint hungry, Direct3D graph was contained to 70KB per core (less than 18KB per hardware thread) on the CPU-like architecture and 285KB per core (less than 15KB per hardware thread) on the more parallel GPU-like architecture. These are perfectly plausible/reasonable resource needs. Finally, we described some tools for understanding how GRAMPS runs an application graph and a few examples of major application improvements to which those led us. In sum, we find these results encouraging, but confined to one domain—rendering—and to simulated—and thus simplified—hardware. The next case study will target both

of these limitations.

# Chapter 5

# Current General-Purpose Multi-cores

This chapter presents a second case study of GRAMPS. After studying GRAMPS on simulated rendering-oriented platforms, we investigated how it applied to current general purpose multi-core CPUs and accordingly incorporated many new applications. This chapter focuses on our experiences specifically with GRAMPS and the next describes extending our implementation to compare with schedulers from other CPU parallel programming models. Much of this material is currently under submission for separate publication [42].

## 5.1   Introduction

We selected our second case study to complement the first in two ways: to build an implementation for real, non-simulated, systems and to consider general purpose hardware together with a broader set of problem domains. Thus, we shifted our focus to multi-core CPUs. As mentioned earlier, performance increases in current CPU designs stem heavily from exposing increasing numbers of cores, and sometimes multiple threads per core, in a single processor and provide a natural target for GRAMPS.

In this chapter, we describe our multi-core GRAMPS implementation and the

results from the diverse array of applications we built for it running on an 8-core, 16-thread machine, cast in terms of our design goals for GRAMPS:

- **Broad application scope**: Nine applications (13 distinct configurations) drawn from rendering, physical simulation, sorting, MapReduce, CUDA samples, and StreamIt.

- **Multi-platform applicability**: A pthreads plus work-stealing GRAMPS runtime for multi-core CPUs.

- **Performance**: Improvements over scheduling for simulated hardware that handle real-world constraints and still produce good scale-out parallelism with queue footprint management.

- **Tunability**: A journaling mechanism for supporting grampsviz and profiling GRAMPS runtime operations.

## 5.2   Application Scope

In keeping with the more general hardware than the previous study, we extended our application set well beyond rendering. Additionally, we intentionally included some programs and algorithmic formulations from examples for existing parallel programming models. Not only does that help ensure our problem domains are valuable, but it will allow us to examine GRAMPS's suitability and overlap compared to other models in the next chapter.

Table 5.1 summarizes the origin and important characteristics of the applications we built. It is important to emphasize that these are not at all the same underlying application structures that happen to manifest in a range of domains: these applications vary significantly. This is clearly visible in Figure 5.1, which shows the application graphs for a diverse sampling of our workloads. Particular points of interest in these graphs are highlighted in the detailed descriptions below.

Specific applications and workloads (configurations) are as follows:

| Workload | Origin | Graph Complexity | % Work in Shaders | Inter-Stage Parallelism | Regularity |
|---|---|---|---|---|---|
| **raytracer** | GRAMPS | Medium | 99% | Yes | Low |
| **spheres** | GRAMPS | Medium | 33% | Yes | Low |
| **histogram-red / com** | MapReduce | Small | 97% / 99% | No / Yes | Low |
| **lr-red / com** | MapReduce | Small | 99% | No / Yes | High |
| **pca** | MapReduce | Small | 99% | No | High |
| **mergesort** | Task | Medium | 99% | Yes | High / Low* |
| **srad** | CUDA | Small | 99% | No | High |
| **gaussian** | CUDA | Small | 99% | No | High |
| **fm** | StreamIt | Large | 43% | Yes | High |
| **tde** | StreamIt | Huge | 8% | Yes | High |

Table 5.1: Application characteristics. Inter-stage parallelism denotes the existence of producer-consumer parallelism at the graph level. *Mergesort is regular in execution time but irregular in queue access.

- **raytracer:** The packetized ray tracer from [41] (and the previous chapter). We run it in two configurations: with zero reflection bounces, in which case it is a pipeline, and with one bounce, in which case its graph has a cycle (see Figure 4.2).

- **spheres:** A rigid body physics simulation of spheres from [42]. It uses a dynamic queue set to perform collision detection: each subqueue corresponds to a voxel in a 3D grid, with its (x, y, z) position used as the subqueue key. The Make-Grid stage determines which voxel(s) each sphere overlaps and routes them correctly, effectively turning the queue set into a sparse grid representation (only subqueues with at least one sphere will be created). The rest of the stages compute all of the intra-voxel collisions, resolves them, and uses the resolved velocities to compute updated positions and velocities. The position updates for spheres uninvolved in any collisions happen in Collide-Cell. The application graph for **spheres** can be seen in Figure 5.1. Note that with large spatial volumes and corresponding grids, Make-Grid can severely stress dynamic subqueue creation, which in turn severely stresses Thread-stage instancing for Collide-Cell.

- **histogram, lr, pca:** Three MapReduce applications described in [42] and
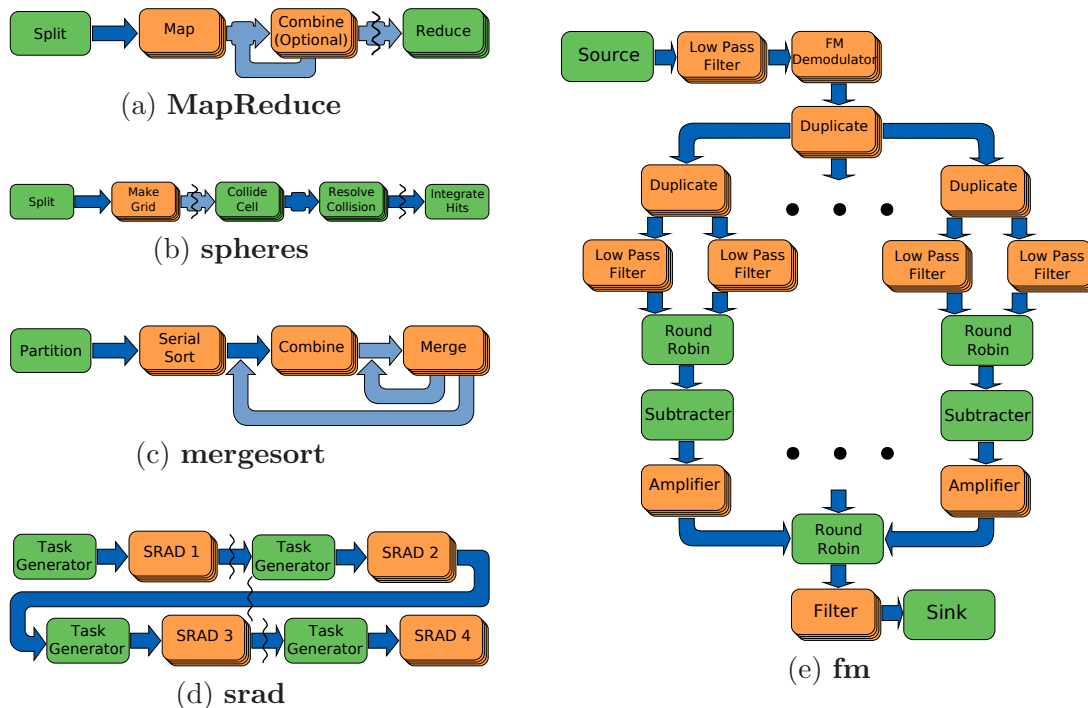
Figure 5.1: GRAMPS graphs for **MapReduce**, **spheres**, **fm**, **mergesort**, and **srad**.

originally from [37]: image histogram, linear regression, and PCA. As with the rendering pipelines, we built a MapReduce 'runtime' as a GRAMPS application (the graph it generates for a single MapReduce pass is shown in Figure 5.1). This was the motivating case for offering In-Place queue bindings and for Shader-stage parallel reductions. Many MapReduce applications, including **histogram** and **lr** use a Reduce kernel that is commutative and associative, and can thus be performed incrementally as values are produced. This option, which MapReduce refers to as Combine, has two big benefits: better load-balancing as values can be freely Combined in parallel and drastically smaller queues as values no longer need to be buffered in their entirety until the entire input collection is available for Reduce. We run **histogram** and **lr** in two forms: with a combine stage and reduce-only. **pca** is interesting in that it is actually two pass: it runs successive Map stages—one to compute per-row means and one to compute covariance between rows—and uses no Reduce (or Combine) stages.

- **mergesort:** A parallel mergesort implementation using Cilk-like spawn-sync parallelism. It runs by partitioning the input data into fixed sized chunks (ala tasks), sorting each chunk in an independent Shader instance, noting when two adjacent chunks are sorted and can be combined, and then merging them. Due to its highly recursive nature, its graph, shown in Figure 5.1, contains two nested loops.

- **srad:** Speckle Reducing Anisotropic Diffusion clears speckled images while preserving edge information. This program, is ported from the Rodinia [9] heterogeneous benchmark suite. Its graph, shown in Figure 5.1 demonstrates the general pattern of casting Breadth-First algorithms in GRAMPS. It is essentially a chain of kernels separated by queues that are only used as barriers—no packets ever flow through them, they just serve to signal the downstream when a kernel is complete. Instead, the kernels process their data through read-write mapped buffers, just like CUDA and OpenCL programs. Each kernel is preceeded by a Task-Generator stage. These stages produce disjoint index ranges into the read-write buffers that are used to instance the Shader stages. One future enhancement to GRAMPS that we have considered is a mechanism to include simple input subdivision information when defining a Shader stage, so that GRAMPS can automatically function as a Task-Generator.

- **gaussian:** An example from the CUDA SDK [30]. Like **srad**, an entirely data-parallel workload, whose graph is laid out very similarly.

- **fm:** The FM Radio benchmark in the StreamIt suite [43]. Like many streaming applications, it is highly producer-consumer and has an elaborate graph, shown in Figure 5.1. After an initial preamble, the input signal is split into frequency bands, each of which passes through its own copy of the core processing graph, and is finally merged and filtered back into a single output signal. Like the Direct3D pipeline in the previous study (Section 4.3.1, Figure 4.1) the application setup code programmatically replicates the repeated chunks of the graph (the per-frequency-range processing).

- **tde:** Also from the StreamIt benchmark suite [16], this is the Time Delay Equalization phase of GMTI (a radar-processing front-end). Like **fm**, its graph is highly complex, highly producer-consumer parallel, and built primarily from programmatically replicated subgraphs (in this case, parallel FFTs). In fact, it is our least Shader-based application graph (see Table 5.1) and instead derives the majority of its parallelism from overlapping its 380(!) Thread stages.

## 5.3   Multi-platform (Implementation)

There are many different potential strategies for designing a GRAMPS runtime to run on general purpose CPUs. We did not attempt to survey the possibilities exhaustively, relying upon our intuitions to guide the high-level design and only experimenting where problems developed. This happened most prominently when implementing dynamic queue sets, with which we tried many variations as described below.

At its heart, our multi-core GRAMPS implementation is a multi-threaded runtime with work-stealing for load distribution/balancing. It creates one pthread per hardware slot (core/Hyper-Thread) and multiplexes application stages onto them with a user-level scheduler. To minimize confusion with GRAMPS Thread stages, the text will use 'pthread' throughout to refer to kernel threads.

### 5.3.1   Data Queues

In this version, GRAMPS data queues are built as standard parallel queues. Each subqueue has a shared head and tail pointer, protected by ticket locks [26], and updated by any pthread that `reserve`'s or `commit`'s to it.

There are two significant details about our implementation of dynamic queue sets: sparse key resolution and memory allocation. With dynamic queue sets, the application indexes subqueues using sparse keys that the runtime converts into dense indices as soon as it enters `reserve` or `push`. Since lookup is much more frequent than new subqueue creation and lookups, read-only operations, can occur in parallel, we expected a straightforward hash table to scale well. We were surprised at the overheads

we saw with both STL [29] and TBB [21] containers. We ended up employing a two-level approach: a concurrent hash map from TBB that all pthreads shared and thread-local (unlocked) STL hash tables for each pthread that cached mappings after their first resolution in the shared table.

Despite the TBB "scalable allocator" and low rate at which new subqueues, and hence new mappings, were created, we still saw significant memory allocation overheads. We side-stepped the problem with slab-based memory preallocation: we added optional calls that a GRAMPS application could use to provide hints to how many dynamic subqueues and Thread stage instances it expected and made the runtime pre-allocate correspondingly sized pools of memory to use until/unless they were exhausted. In the process of development, we experimented with many permutations of single and two-level, STL and TBB hash tables as well as pre-allocation, the TBB scalable allocator, and the default STL allocator. Of all combinations, this design worked best. We believe the best solution ultimately may be a specialized single-level shared hash table with lock-free atomic updates and lookups, and that takes advantage of the fact that mappings are never deleted.

We added one optimization for applications that make extensive use of regular queues (as opposed to queue sets). An application can tag a queue as 'balanced', in which case the runtime implements it as a queue set with one subqueue per pthread and transparently routes queue operations to the subqueue corresponding to the pthread on which they run. This significantly reduces contention on the queue accesses for **raytracer**.

### 5.3.2   Task Queues

Unlike the periodic scheduler and dispatch of the simulated implementations, each pthread in the multi-core runtime selects its stage/instance to execute from task queues. Specifically, it is based upon the non-blocking ABP task-stealing algorithm [3] using Chase-Lev deques [8]. However, unlike most task systems, instead of one task queue per pthread, the runtime has one task queue *per priority* per thread, where

priorities are assigned exactly as in the preceding chapter. The details of the scheduling policies—when/how tasks are generated and preempted—are given below in Section 5.4.1.

### 5.3.3   Termination

One tricky, but crucial, piece of the multi-core GRAMPS runtime is handling stage completion and propagating it down the graph. When a stage finishes, the runtime decrements the count of live producers for each of its output queues and then enqueues a task for each of its Shader and blocked Thread consumers as if an output `commit` had happened. This causes GRAMPS, when those tasks are scheduled, to check the input queues for those stages. Thread stages, as mentioned in Section 3.6, receive a short reservation once all of their producers are done and explicitly yield control back to the runtime once they are done in turn.

Shader stages require more complicated handling, since they are instanced automatically and can have many concurrently live copies. When the multi-core runtime finds a Shader task, but no available input and the upstream stage is done, it marks the input subqueue as done and decrements the number of live input subqueues. The Shader task that decrements the number of live subqueues to zero marks the stage itself as 'finishing'. For each Shader stage, the runtime maintains an atomic counter of the number of currently dispatched instances and the number of pthreads that have partially coalesced `push` outputs for it. When the instance counter reaches zero for a stage that is 'finishing', if there are no outstanding pushes, then it becomes done. If there are outstanding pushes, the last pthread to flush them will mark the stage as done. This will happen promptly because there will be no new tasks generated for this stage: a pthread flushes whenever it dequeues a task for a new stage or goes idle. While this distributed state machine required careful reasoning to construct, it detects Shader stage termination with neither locks nor support for explicit pthread-to-pthread messaging.

## 5.4 Performance

### 5.4.1 Scheduling

Recall, as in the prior chapter, that our key performance mantra is, "Maximize machine utilization while keeping queue footprints small", that our focus for utilization is scale-out without forgetting scale-up, and that this is fundamentally a scheduling consideration. In the multi-core GRAMPS implementation, it amounts to determining when to create tasks and when to preempt. These are similar to the simulated study, but the greater complexities of reality necessitate some improvements.

**Task Creation:** The multi-core GRAMPS scheduler generates tasks in a relatively unsophisticated way. When output is `commit`'ed to a data queue, one task is generated for each consuming Shader stage and blocked Thread stage. These tasks indicate which stage and input subqueue to examine, but *not* a specific assignment of work. Rather, the runtime attempts to satisfy the stage's input reservation (or pre-reservation) when it dequeues the task and discards tasks that prove spurious (which happens rarely).

**Preemption:** Preemption refers to changing which stage a given pthread is executing, not just which task: multiple back-to-back tasks for the same Shader stage are not considered preemptions. As mentioned above, this runtime employs the same static per-stage priorities (favoring 'later' stages in the graph over 'earlier' ones) as both of the simulated runtimes (Section 4.5.1). It also checks for preemption at the same points: on `reserve` and `commit` for Thread stages and between packets for Shader stages.

However, real machines lack the simplifications made in simulation. There is a tension between promptly switching stages when higher priority tasks are generated and amortizing the costs of preemption. We refer to too-aggressive switching among stages as 'ping-ponging'. In addition to its low-level costs (e.g., user level context switching, flushing of partially coalesced `push` packets), ping-ponging can severely impact load-balancing. Consider the singleton producer → Shader consumer idiom
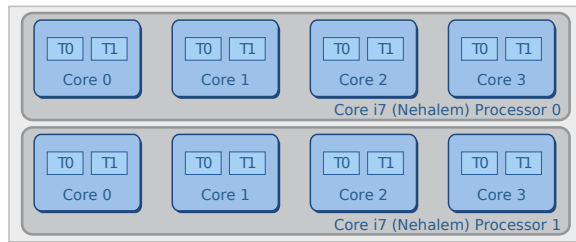
Figure 5.2: Our two quad-core HyperThreaded test system.

described in Section 3.6: if the producer is preempted as soon as it `commit`'s any output, then no other pthread will have work to do until it incurs the expense of stealing the producer and loading its context (after which that pthread will rapidly preempt it, too and perpetuate the problem). There are similar problems switching back and forth between a Shader producer and consumer stages. These realities meant rigid adherence to switching to the highest static priority stage was unreasonable.

Instead, the multi-core GRAMPS scheduler avoids ping-ponging by applying a low watermark below which it does not preempt (as long as the current stage remains runnable). When a Thread stage `commit`'s, the runtime compares the number of total tasks available in that pthread's task queues against the threshold. This gives the pthread time to accumulate some downstream work and amortize preemption. Also, crucially in the singleton producer case, it gives other work-poor pthreads a chance to steal the new work, which in turn allows the producer pthread to run longer before reaching its watermark. Similarly, when a Shader finishes its post-`commit`, the scheduler checks the number of tasks that stage has run in a row. Below the low watermark, the pthread attempts to dequeue more tasks *of the same priority* before checking any others. We implemented configurable per-stage watermarks, but found that in practice a default global value of twice the number of cores/pthreads worked well.

## 5.4.2   Evaluation

We used a 2-socket system with quad-core 2.66 GHz Intel Xeon X5550 (Nehalem) processors to conduct our study (Figure 5.2). With Hyper-Threading, the system
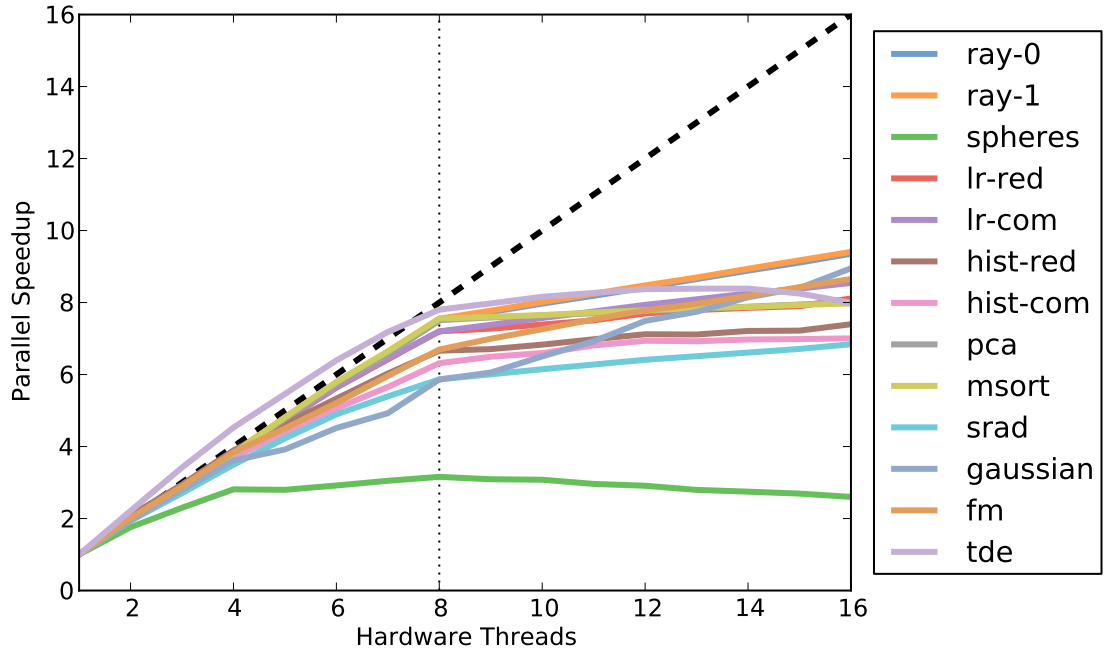
Figure 5.3: Application speedup on an 8-core, 16-thread system.

delivered a total of 8 cores and 16 hardware threads. Its memory hierarchy had 256 KB per-core L2 caches, 8 MB per-processor L3 caches, and 24 GB of DDR3 1333 MHz memory. The processors communicated through a 6.4 GT/s QPI interconnect. As software, we used 64-bit GNU/Linux 2.6.33, with GCC 4.3.3. We ran all of our experiments 10 times and report the averages as our results. This rest of this section focuses on the execution/utilization aspect of the results. The memory/footprint data, which is in fact good, is best understood in the context of alternative scheduling strategies and therefore deferred to the comparison and discussion in the next chapter.

**Scalability**

With a real system, we were able to measure scale-out parallelism directly. Figure 5.3 shows the scaling of each application as we configure multi-core GRAMPS to use from 1 to 16 pthreads. The knee that consistently occurs at 8 pthreads is where all the physical cores were filled and the runtime started provisioning the second
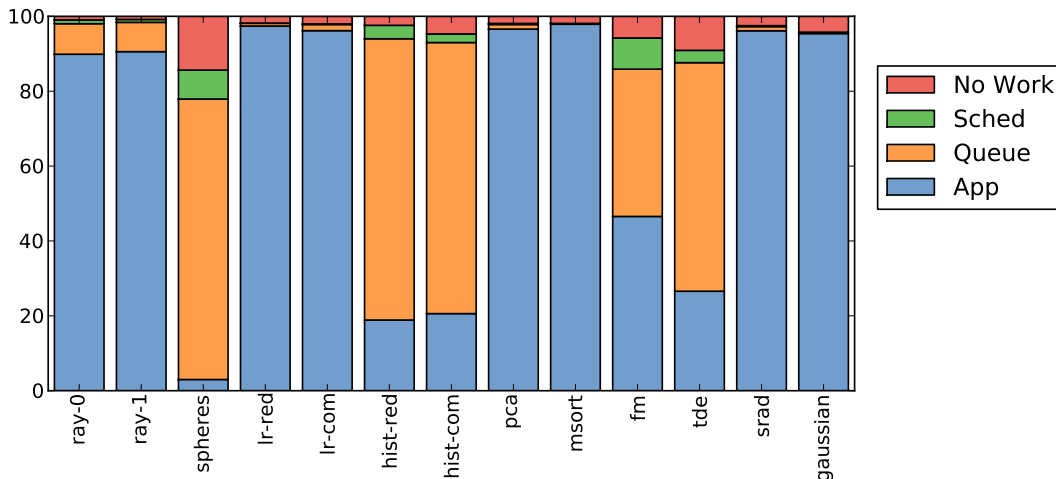
Figure 5.4: Execution time profile (8 cores, 16-threads). 'No Work' represents the time a pthread had no local work and was either idle or trying to steal.

Hyper-Thread on cores.

Overall, all of the workloads besides **spheres** scaled out well. With more cores, **srad** and **gaussian** both started to become bound by cache and memory bandwidth; they are ports of designs for GPUs, which have significantly more bandwidth. **spheres** is a particular challenge. As mentioned earlier, it stresses Thread stage instancing: even when represented sparsely, there are almost 5000 grid cells that overlap with objects in the simulation (and thus instances). We have not focused on optimizing contention in our Thread instancing at that scale.

**Utilization**

Good parallel scalability still does not eliminate the importance of scale-up, i.e., how much time is spent running 'useful' code. To investigate, we profiled our workloads. Figure 5.4 displays the results from running with all 16 hardware threads, grouped into four categories: application code itself, queue manipulation in the runtime (e.g., `reserve`, `commit`, `push`, etc.), scheduling, and without work (e.g., idling or trying to steal tasks from other pthreads).

The vast majority of execution time is spent in application or queue time, both 'useful'. Execution graphs with heavy amounts of communication spend time accessing their data queues, but this is still productive: these applications inherently need to communicate. Using GRAMPS's data queues to synchronize means that none of our application code has locks. If it had to roll and manage its own shared data structures internally, then time would have to be spent there.

As expected from the scalability results, work is plentiful without much imbalance, so 'No Work' time is not much of a factor outside of **spheres**. Additionally, scheduling time is minimal: most workloads spent less than 1% of the time in the scheduler. Even tracking the tens of stages in **fm** (and thousands of instances in **spheres**) took less than 10% of the elapsed time.

In summary, except for **spheres**, the rest of our applications—regular and irregular; data-parallel, task-parallel, and pipeline-parallel; Shader-heavy and Thread-heavy—scaled well with GRAMPS, had good work distribution, and had low scheduling costs. **spheres** stresses a particular feature of the programming model to an extreme far past where we have delved into it or done any tuning.

## 5.5 Tunability

As is evident from the above data and discussion, we included instrumentation and configuration options for our multi-core GRAMPS implementation to aid explaining and improving performance. In addition to configuration knobs for parameters such as the scheduling watermarks, number of pthreads, and amounts of memory to pre-allocate, it has two significant systems for recording run-time data: counters and the journal.

The counters are a lightweight scheme any part of multi-core GRAMPS can use for accumulating integer values (e.g., number of commits, or tasks stolen at a time). After a run, the minimum, maximum, average, and standard deviation are all reported. The counters, in conjunction with the lightweight processor cycle counter, underpin our profiling.

The journal, by contrast, is a mechanism for recording bulkier and freer-form data

for post-processing. Callsites specify a particular journal opcode, but otherwise up to five words of arbitrary data. Each pthread tags journal entries use the cycle counter, but accumulates them in static thread-local buffers until they fill, at which point they are spilled to disk and reset. This reduces the overhead of the journal sufficiently for its data to be useful for tuning and diagnosis. It is enough that we disable the journal for performance runs, however. Nevertheless, the journal is very useful for gathering a detailed picture of execution. We use it, among other things, for recording data to enable a ported version of grampsviz from the simulator environment. In fact, we extended grampsviz (and the multi-core runtime) so that the queue view could also include any unflushed data from `push` that was buffered for coalescing and, with detailed instrumentation enabled, the core view could overlay a visual profile of where in the application or GRAMPS-runtime code each pthread was executing.

## 5.6   Conclusion

This case study has complemented the limitations of the previous one. Its hardware is less futuristic and less heterogeneous, but has the enormous validating property of being real and representative of general purpose many-core architectures available today. Our abstractions and scheduling policies help up well, but unsurprisingly profited from a few enhancements—particularly the scheduling watermarks that extend entirely fixed priorities to amortize out ping-ponging.

In addition, this study greatly broadened the origins and domains of applications formulated for GRAMPS to draw from other programming models and span a range of problems including sorting, image and signal processing, physical simulation, and MapReduce. With the exception of one case that stressed a code path far past its design, they demonstrated both scale-out performance across multiple multi-threaded cores and scale-up performance, spending the large majority of execution time on useful work. We have deferred a detailed investigation of queue footprint to the next, final, case study, where it can benefit from comparison data from alternative policies.

# Chapter 6

# Comparing Schedulers

This chapter presents the third case study. We use the multi-core GRAMPS runtime from the previous study as a testbed for implementing the scheduling policies of Task-Stealing, Breadth-First, and Static parallel programming models alongside our GRAMPS scheduler. Then, we analyze how our applications behave using the different schedulers in terms of both performance and footprint. As with the previous study, much of this material is currently under submission for separate publication [42].

## 6.1   Introduction

Unlike the first two case studies, our third does not introduce a new GRAMPS implementation. Instead, we round out our understanding of GRAMPS by reusing the multi-core implementation from the previous study to compare GRAMPS with the other general purpose programming models described in Chapter 2.

There are many important dimensions in to evaluate a programming model: syntax, toolchain, ease of use, etc. We focused on runtime *resource management and scheduling*. These shape key factors of execution: performance (execution time), scalability, memory footprint, and locality. In order to hold as many factors as constant as possible, we conducted the comparison entirely with internal changes to the multi-core runtime. We reused the applications from Chapter 5 entirely unmodified. Note that this, therefore, is a study of specific application/algorithmic formulations of various

problems rather than the highly subjective (and less apples-to-apples) comparison of a 'best' or 'preferred' formulation per scheduler. Instead, we rely upon the fact that we drew from representatives of each of the models when selecting our applications initially. Thus, each scheduler had at least one case aligned with its strengths.

In this chapter, we describe how each scheduler was cast in terms of our multi-core GRAMPS runtime, go through their results handling our applications, and discuss the strengths and weaknesses they reveal.

## 6.2   Representing other Programming Models with GRAMPS

We implemented three new operating modes for our runtime: Task-Stealing, Breadth-First, and Static. For each, we developed an alternative scheduler and tweaked the data queues as described below. Figure 6.1 shows images from grampsviz depicting how each scheduler mapped stages to pthreads while running **raytracer**. GRAMPS and Task-Stealing look similar, which makes sense as both are task-based internally and load-balance by adaptively moving tasks among pthreads. As we will see, GRAMPS and Task-Stealing generally resemble each other in most tests, diverging the most when tasks are very small or stage priorities and/or queue capacity limits are significant. Breadth-First, however, is strikingly distinct. Its algorithm—a succession of kernels (stages), one-at-a-time—is immediately visible. Finally, while Static is less apparent, its fixed rotation among the stages manifests clearly as periodic patterns that are not present with the two adaptive schedulers.

**Task-Stealing:**   As discussed in Chapter 2, the generic task-stealing model is focused on lightweight task creation and dispatch. It eschews functionality that might add expense, such as priorities. Since our GRAMPS runtime was based on task-queues, our Task-Stealing mode is similar, but has three key differences: unbounded capacity data queues; no task priorities; and preemption based upon child task creation rather than graph priorities and watermarks. It uses the same task-stealing

(a) GRAMPS
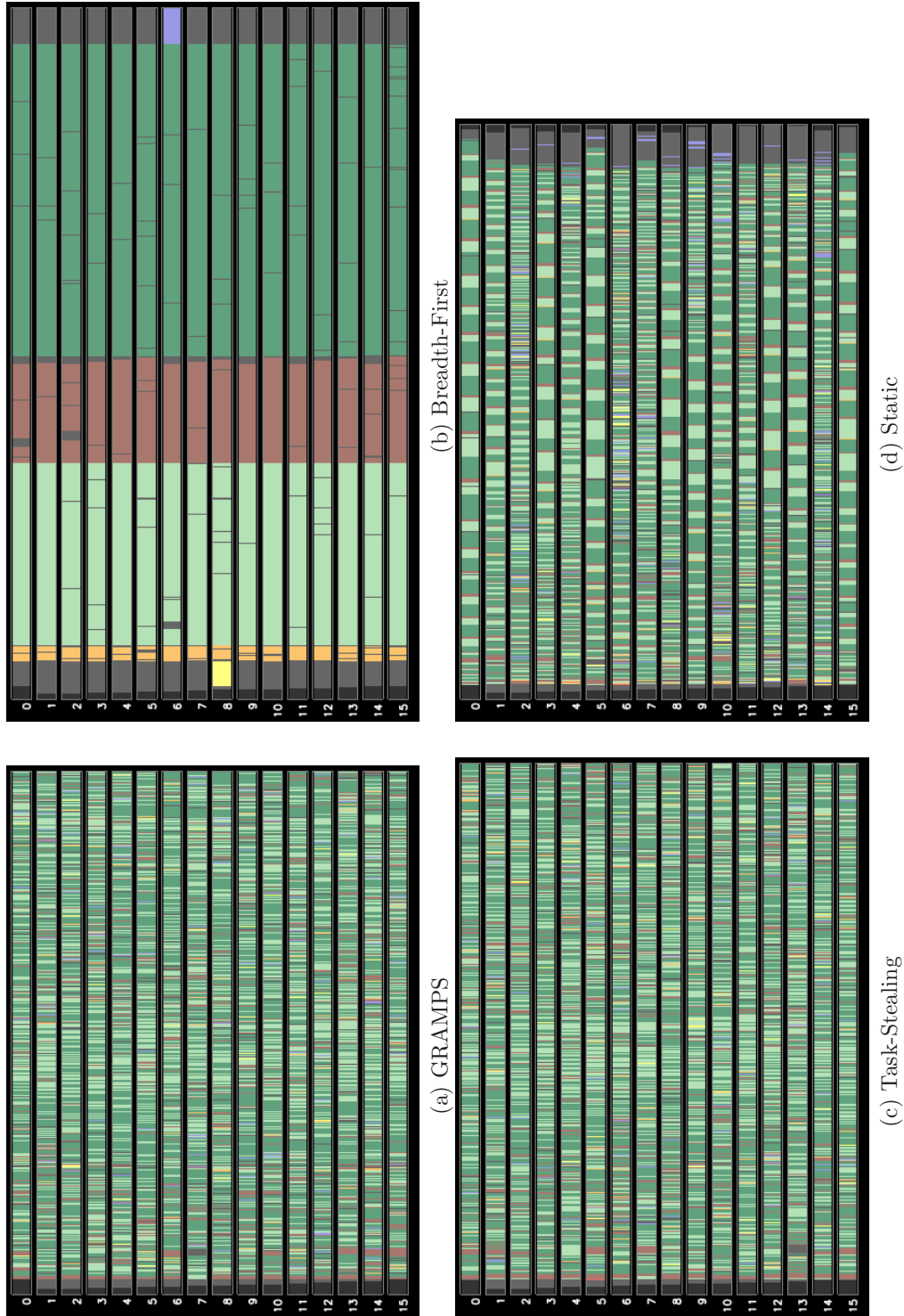
(b) Breadth-First

(c) Task-Stealing

(d) Static

Figure 6.1: Grampsviz output showing how each scheduler behaves running **raytracer** (each horizontal bar corresponds to a pthread and each color to an application stage and image width is proportional to execution time).

algorithm and Chase-Lev deques as GRAMPS mode [3, 8], but with a more conventional single queue per pthread. Local enqueues and dequeues happen in LIFO order from the front of the deque with steals from the back. Again, there can be many variants in implementation choices, but our design captures the gist of performant task-stealing runtime systems.

With no scheduling awareness of stages, preemption is purely task based. To emulate the depth-first policy of Cilk [15], Thread stage producers are preempted after committing a fixed number of output packets (currently 32). A pure depth-first policy would require context-switching the producer every time it commits an output packet, which is expensive. This approach retains most of the benefits of depth-first while amortizing context-switching overheads.

**Breadth-First:**   This mode is a complete departure from GRAMPS. In pseudo-code, it runs an application graph as follows:

```
while (not done and numPasses < max) {
all pthreads:
   while (curStage has input available) try to run curStage
   barrier
pthread 0:
   curStage++
   if (curStage is the last in the graph) {
      if (all stages are done) terminate()
      curStage = firstStage
      numPasses++
   }
}
```

Essentially, starting from the inputs to the graph, all pthreads run the current stage as long as it remains runnable, then they advance in lock-step to the next stage. Some of our applications have cycles in their graphs so the scheduler will reset to the top of a graph finite number of times if necessary. As with Task-Stealing, Breadth-First mode does not enforce any bounds on the depths of the data queues.

**Static:**   At execution time, the Static scheduler is the simplest of all. When the runtime initializes, it reads a specified schedule file which enumerates a sequence of stages per pthread. At run-time, each pthread cycles through its schedule until all

stages are done, skipping stages that are not runnable and going back to the top as necessary until finished.

Generating the static schedules, however, proved not to be simple at all. We experimented with multiple stream scheduling algorithms, but it was not a trivial process. Without hand tweaking, these algorithms frequently failed to generate viable schedules for more irregular applications (e.g., variable outputs/push queues) and more complicated graphs (e.g., loops). As in streaming systems, we also needed to generate configuration-specific schedules for applications that varied in response to workload parameters such as the image to process, output resolution, data range per packet, etc. Finally, this Static scheduler design—running an input schedule in a loop—inherently assumed data is streaming through and cannot handle idioms such as barriers or all-at-once queue consumption. As a result, we used a limited set of workloads for our Static scheduler experiments.

We created the schedules for the workloads we did use according to two state-of-the-art static scheduling algorithms: SGMS and SAS. Stream Graph Modulo Scheduling (SGMS) is a recently proposed scheme that bin-packs stages based on an execution time estimate, where each bin denotes a hardware thread, duplicates stateless kernels as necessary for performance/utilization, and software pipelines the results for a final schedule [24]. It was efficient with our larger graphs, but generated degenerate/useless schedules when there were fewer stages. In those cases, we used the traditional Single Appearance Schedule (SAS) [25], which calculates the relative frequency of each kernel/stage at steady state and replicates that schedule on all pthreads (essentially, the strip mining strategy described in [12]).

## 6.3   Evaluation

We ran our evaluation on the same 8-core, 16-thread system that we used to evaluate GRAMPS (Chapter 5). We compared GRAMPS, Task-Stealing, and Breadth-First first, and then Static separately because of its smaller set of schedules.
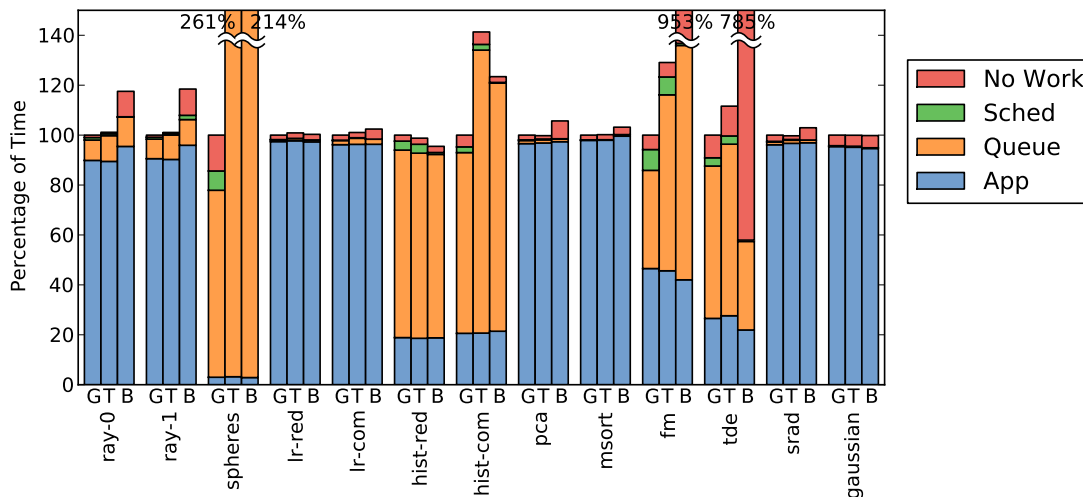
Figure 6.2: Execution time profile (8 cores, 16-threads) of applications running the GRAMPS, Task-Stealing, and Breadth-First schedulers (left to right).

## 6.3.1   Execution Time

Figure 6.2 displays the same sort of profile as Figure 5.4, but includes the breakdowns for GRAMPS, Task-Stealing, and Breadth-First. The data is using all 16 hardware threads and scaled relative to GRAMPS. To a first order, all three exhibited the same profile for many applications. This is unsurprising: all the applications are parallelism-rich and most are easy to keep well-balanced. However, as will be discussed below, the three are not so similar in how much queue space they consumed.

There were a few workloads where the schedulers were visibly different. The Breadth-First schedulers suffered load-imbalance (larges amounts of 'No Work' time) in multiple ways from its inability to overlap stages. It could not exploit the inter-stage/pipeline parallelism in the Thread-stage heavy **spheres**, **fm**, and **tde**: while one hardware thread runs a Thread stage, all the others sit idle. In other cases, intra-stage irregularity was the problem: some **raytracer** stages left most hardware threads idle while a few long running Shader instances completed. The irregular distribution of values to keys in the in-place Shader of **histogram-combine** also exacerbated contention relative to schedulers that could co-mingle instances from both the map
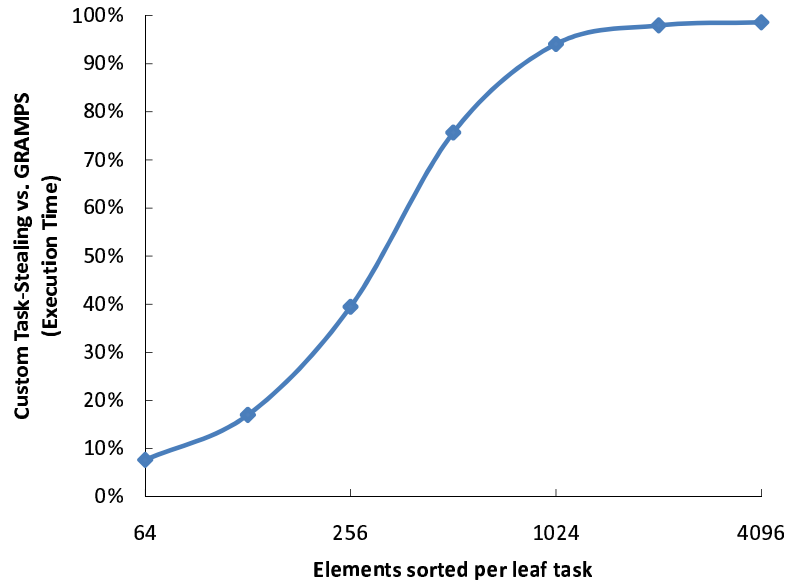
Figure 6.3: Relative execution time running **mergesort** with a custom Task-Stealing runtime, as a function of the leaf task size.

and combine stages.

Task-Stealing, in turn, occasionally demonstrated higher overheads in cases where semantic information from the graph gave GRAMPS an advantage. Its depth-first heuristic was an inadequate alternative to GRAMPS's priorities and stage-based preemption watermarks in more complicated graphs (e.g., the complicated fan-out and fan-in structure of **fm** and **tde**). Similarly, it ping-ponged heavily on **histogram-combine** and **spheres**, which then increased contention accessing the subqueues and creating Thread stage instances.

**Task Size Sensitivity**

As just described, failing to utilize application graph information can be a disadvantage for Task-Stealing. However, it can also be an advantage. Task-Stealing programming models ignore application structure, priorities, etc. by design, not foolishness. Ignoring them streamlines scheduling and recall that Task-Stealing emphasizes very low per-task overheads.

To highlight this trade-off, we built a stripped-down Task-Stealing runtime that
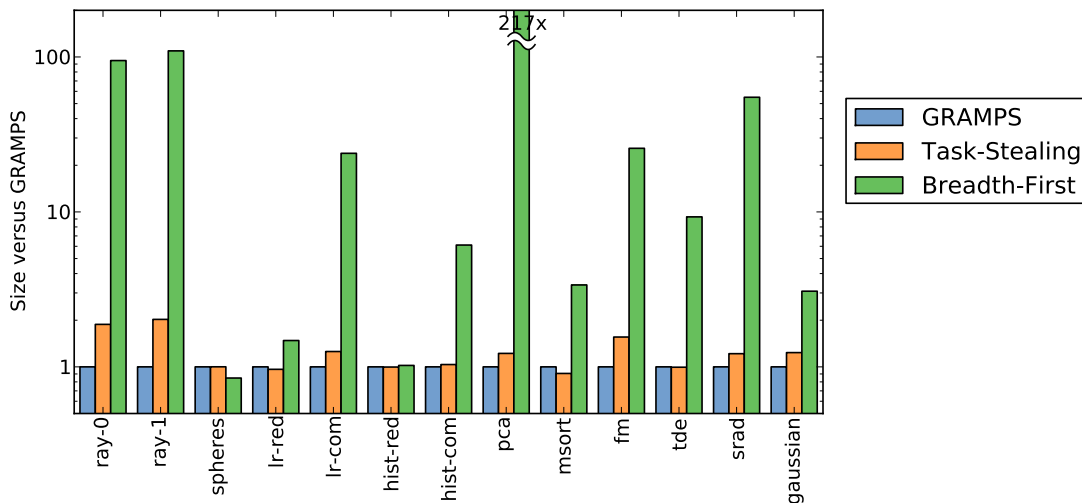
Figure 6.4: Relative data queue depths for GRAMPS, Task-Stealing, and Breadth-First. Breadth-First tends to be huge in comparison (note the log scale).

removes data queues and directly exposes tasks and ported **mergesort** to it. We then varied the size (i.e., number of elements) of the serial sort that constitutes its leaf tasks and compared the performance of GRAMPS and the custom Task-Stealing runtime.

As Figure 6.3 shows, attempting to exploit application knowledge is not free. Pure Task-Stealing significantly outperformed GRAMPS on fine-grained (small) tasks. GRAMPS caught up relatively quickly, however, reaching 75% of Task-Stealing's performance when sorting 512 elements per task and 94% with 1024. This makes sense, since, as shown in Figure 6.2, using the application graph to schedule was neutral or positive for all of our workloads. And, it turned out to be very helpful for reducing queue footprints.

## 6.3.2   Footprint

Recall that containing working set sizes is an important design point of the GRAMPS scheduler, as it affects locality (caching effectiveness), memory bandwidth, and storage requirements.   Figure 6.4 shows the data queue footprint of our workloads in

(a) Packets  (b) Tasks
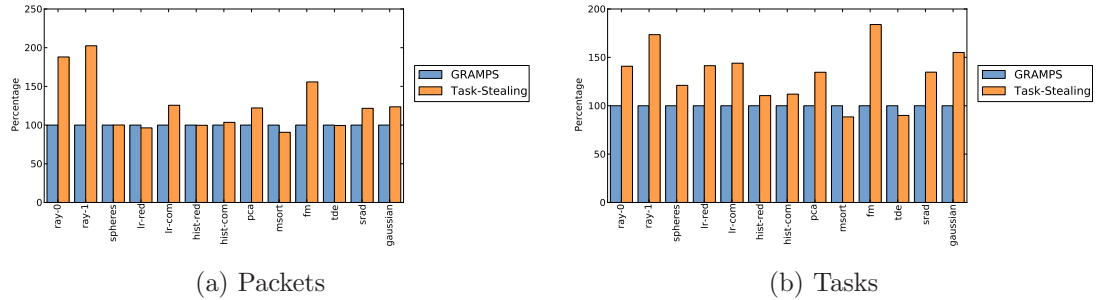
Figure 6.5: Relative footprints of GRAMPS and Task-Stealing

GRAMPS, Task-Stealing, and Breadth-First modes. Specifically, it shows the average number of total packets enqueued (across all queues) during execution.

The plot is drawn to a log scale because the footprint in Breadth-First mode usually dwarfs the others. The only exceptions are the MapReduce workloads, whose barriers make them innately breadth-first independently of the scheduler (**spheres** is very MapReduce-like in this sense). This is unsurprising. Not only does Breadth-First produce extremely deep queues, it produces the deepest possible queue between any two stages: each producer runs until it can run no more before its consumer is ever scheduled.

Figure 6.5 removes Breadth-First and zooms in on GRAMPS and Task-Stealing. Since both schedulers are task-queue-based, those footprints are shown, too.

As before, GRAMPS and Task-Stealing again often look alike. This is also unsurprising: the preemption heuristic in Task-Stealing is meant to approximate the prefer-consumers-over-producers behavior of GRAMPS without actually implementing task priorities. However, Task-Stealing has two disadvantages: its approximation always defers preemption until 32 tasks are generated, regardless of how work-rich the system already is; and unbounded data queues mean Task-Stealing misses any queue sizes set by application to throttle active stages.

In practice, this means that Task-Stealing's heuristics worked for the simple graphs, but deteriorated with the extended Shader producer $\rightarrow$ Shader consumer

(a) Execution Time                                  (b) Packet Footprint
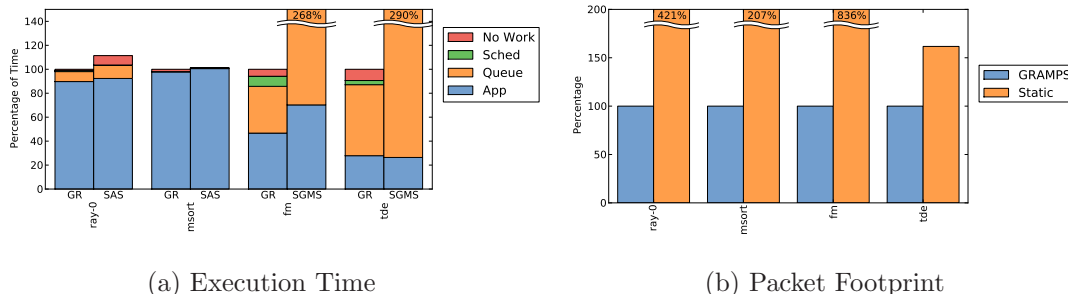
Figure 6.6: Static scheduling results

chain in **raytracer** and the complicated graph of **fm**. Additionally, recall that **raytracer**'s footprint is also highly affected by the maximum queue depths, which Task-Stealing ignores (Section 4.6.2). Task-Stealing also had trouble with the singleton producer → Shader consumer idiom in **pca**, **srad**, and **gaussian**. The inflexibility of its deferred depth-first algorithm could not match the stage-based watermarks.

The most visible indication of Task-Stealing's deviation from GRAMPS is the task queue footprints. Tasks are small in absolute terms, but they are all the same size. This means that the ratio of the task footprints indicates how many more tasks are pending with Task-Stealing on average: often 30% and as much as 84%. Since both schedulers had ample work to fill the machine, these extra tasks reflect work that need not have been generated, or buffered, so soon.

### 6.3.3   Static

Figure 6.6 shows the execution time and data queue footprint comparisons between GRAMPS and the Static scheduler for the four workloads for which SAS or SGMS (and mild hand-tuning) produced viable schedules. Static underperformed GRAMPS in all cases, for two basic reasons: static schedules are a poor match for irregular workloads and unlike stream processors, *all* workloads are irregular on general purpose machines. Even a good static schedule for a workload that is regular at the application level must confront irregularity from the memory hierarchy, occasional OS interrupts,

and other dynamic system events that violate its assumptions.

Since their execution order is static, the schedules are fragile: they have no recourse to recover/adjust when small imbalances form. The queue footprints show evidence of this: the static schedules did not quite align the actual production and consumption rates and the footprint slowly grew over successive passes (until stages finished completely and began terminating).

At the same time, there is also not much performance gain available from eliminating run-time scheduling costs. For all workloads, the profiles for GRAMPS and Task-Stealing showed minimal scheduling overheads. Therefore, dynamic scheduling, at least enough to compensate for system irregularities, seems better suited to general-purpose machines.

## 6.4 Conclusions

This case study has compared the resource management effectiveness of GRAMPS and the Task-Stealing, Breadth-First, and Static canonicalizations from Chapter 2 on a general-purpose multi-core machine. All four, in general, are able to exploit ample parallelism to keep the hardware threads productively occupied, though Breadth-First struggles with the more pipeline-parallel, less data-parallel, StreamIt applications. The inflexibility of both Breadth-First and Static scheduling is a serious load-balancing disadvantage on our machine. Dynamic, *adaptive*, scheduling should clearly play a role in any runtime for general-purpose hardware.

GRAMPS stands out at managing the amount of intermediate buffering consumed by our workloads, that is, the depths of the data queues. As intended by the programming model design, the multi-core GRAMPS runtime leverages its insight into the application structure (graph, queue capacities, etc.) to do an excellent job minimizing footprint. Breadth-First scheduling's stage-at-a-time approach generates extremely deep queues. Static's inability to adjust dynamically leads to a steadily deepening creep. Task-Stealing comes the closest, but its philosophical imperative for lightweight task manipulation comes at a trade-off in control over queue depth.

As the sensitivity comparison demonstrated, while GRAMPS's more elaborate structure does introduce some execution overhead, it rapidly amortizes out at the task granularities our applications use.

# Chapter 7

# Discussion

## 7.1  Contributions and Take-aways

This thesis has introduced the GRAMPS programming model for designing parallel applications to match the trend in commodity computing: heterogeneous, many-core systems. GRAMPS casts applications as execution graphs of stages and queues where developers express the amount of automatic intra-stage parallelism among singleton Thread, instanced Thread, and Shader stages and express hints about flow control, where to buffer work, and synchronization with limited capacity queues and queue sets.

In addition, through three case studies, we described GRAMPS's viability for implementation according to four criteria:

- **Broad application scope**: Three rendering pipelines demonstrated on simulated graphics hardware and nine applications, thirteen distinct configurations/graphs, for general purpose hardware including graphics, MapReduce, image processing, and stream processing.

- **Multi-platform applicability**: Three GRAMPS runtime implementations: two for simulated graphics hardware–CPU-like and GPU-like—and one for existing multi-core CPU machines, tested with eight HyperThreaded cores (16 hardware threads).

- **Performance**: High parallel utilization in simulation and actual scalability on real hardware with good working set (data queue) management—in absolute terms in simulation and in comparison to alternatives on real hardware.

- **Tunability**: Grampsviz for visually summarizing and interactively inspecting scheduling and data queue behavior and examples of high impact execution graph modifications learned from analysis.

There is a fifth high value goal/criteria for a programming model: *informing hardware designs*. That is, GRAMPS should provide sufficient opportunity (and clarity of intent) for hardware implementations to be tuned in support of it. While we did not delve in this direction explicitly, our experiences have given us some intuitions. In each case, we focus on the problem/opportunity rather than try to prescribe (and presume) a particular solution:

- Efficient building blocks for many-producer/many-consumer queues. Inherently, the operations of appending or fetching data using a buffer free from strict ordering requirements has relatively simple synchronization requirements. In fact, though, implementing it with normal read/write memory, especially cache coherent memory, generally requires algorithms that generate a lot of unproductive coherency traffic and sharing overheads.

- Queues/messaging primitives between 'cores'. For heterogeneous systems to be successful, especially across varying configurations, there must be a standard method that runtime or application software can use for at least coarse-grained communication. We found queues to be effective abstractions for impedance matching: providing an asynchronous and incremental channel between operations with different execution granularities, frequencies, etc. Any sort of asynchronous messaging ability can likely be made to work, but we believe it is crucial for software adoption that it be as standard and widely available as possible.

- 'Data-parallel' units (GRAMPS Shader cores). GPUs have established the undeniable computing density possibilities of cores designed for data-parallelism.

We would like to see a few enhancements, foremost among them multi-tenancy—
the ability to have multiple co-resident Shaders—efficient mid-sized launches—
the ability to launch, in GRAMPS parlance, a few appropriately sizes packets
worth of work at a time instead of an entire chip's worth (a middle ground
between current hardware and a single execution context at a time)—and `push`,
or some alternative primitive for automatic/runtime managed compacted con-
ditional data generation.

- CPUs with Shader cores. In the tradition of FPU and SIMD units, we believe
  there could be a place for data-parallel cores on future CPUs. Multi-core, after
  all, makes it easier for non-symmetric core distributions than multi-processing.
  And, it seems likely that a point of diminishing returns exists at which, as with
  FPUs, including some dedicated data-parallel cores is a big enough boost to
  a broad enough set of applications that they constitute a better use of design
  resources than additional conventional cores.

Finally, we compared GRAMPS's scheduling with representatives of the three
broad categories for general-purpose programming models—Task-Stealing, Breadth-
First, and Static—on a multi-core CPU system. The results demonstrated that the
multi-core GRAMPS scheduler consistently out-performed, often greatly, the other
schedulers at keeping queues shallow without sacrificing execution performance.

## 7.2 Final Thoughts

In addition to the major contributions identified, we have observed a few other inter-
esting things in developing GRAMPS. The first is that, when structure is present,
exposing it explicitly helps. That is, GRAMPS's insight into a workload from the
execution graph is central to its ability to manage it well. Additionally, we found that
sketching out at least a first draft of the graph *before writing any code* helped us make
better application designs. The second is that dynamic scheduling that is simple,
albeit rooted in sound principle, was surprisingly effective. There is uncountable po-
tential research, and likely dissertations, in exploring advanced scheduling techniques

for GRAMPS runtimes, but our straightforward algorithms exceeded our expectations. Third, as mentioned above, queues impedance match all sorts of heterogeneity well: different types of cores, different stages, etc. Finally, our two sharp-edged, expert features—allowing cycles in execution graphs and `push`, conditional output for Shaders—paid off. In practice, our implementations handled well-behaved applications fine and we were able to express algorithms that would have been difficult to impossible to build without them.

In closing, we see commodity hardware trending towards increasing numbers of cores and heterogeneity. This is fueling interest in high-level parallel programming models to ease application development and manage the underlying complexity. Additionally, this trend makes scale-out a performance consideration to rival scale-up. That, in turn, makes memory bandwidth and capacity increasingly precious resources. GRAMPS, with its dual emphasis on exposing application parallelism and managing footprint, has the potential to be highly relevant in these environments.

# Appendix A

# Sample GRAMPS Code

This appendix provides the code to **set-reduce-keyed**, one of the regression tests for the multi-core GRAMPS runtime from Chapters 5 and 6. This simple program exercises dynamic queue sets and instanced Thread stages: a generating stage distributes the values from 0 to $N-1$ among $M$ subqueues and an instanced consuming stage reads its entire input, sums it, and validates it against the expected result.

## A.1   Application Graph Setup

The primary API for defining and launching an application graph is a set of C / C++ bindings to GRAMPS, listed in its entirety in Table A.1. Most of the functions are self-explanatory, and also shown in the example below. `GrGetPropInt`/`GrSetPropInt` are for querying and specifying implementation-specific parameters, for example, the parameters for the hardware rasterizer of the simulated runtimes in Chapter 4 and the memory preallocation hints for the multi-core runtime in Chapter 5.

Many of our programs use a simpler config-file-like interface we have built for an interpreter called **grampsh**. It allows us to build graphs and experiment with changes with less boiler-plate and no recompilation. We list both versions of **set-reduce-keyed** below. The two are equivalent, except that the C++ version has had its error checking omitted for brevity while **grampsh** performs it implicitly.

The graph for **set-reduce-keyed** is shown in Figure A.1 and is very simple.

| **Queues** |
|---|
| `GrQueueId GrCreateQueue(queueDesc);` |
| `void GrDestroyQueue(queueId);` |
| **Buffers** |
| `GrBufferId GrCreateBuffer(bufDesc, optionalData[]);` |
| `void GrDestroyBuffer(bufferId);` |
| `void GrMapBuffer(bufferId, window);` |
| `void GrUnmapBuffer(bufferId, window);` |
| **Stages** |
| `GrStageId GrCreateStage(threadDesc);` |
| `GrStageId GrCreateShaderStage(shaderDesc);` |
| `void GrDestroyStage(stageId);` |
| `void GrBindQueues(stage, mode, numQueues, queues);` |
| `void GrBindBuffers(stage, numBuffers, bufs[]);` |
| **Execution/Launch** |
| `GrWaitId GrSetRunnable(numStages, stages[]);` |
| `void GrWait(graphId);` |
| **Miscellaneous Configuration** |
| `int GrGetPropInt(index);` |
| `void GrSetPropInt(index, value);` |

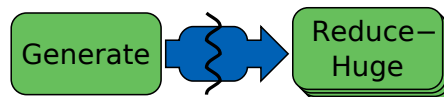Table A.1: The GRAMPS application graph API



Figure A.1: Application graph for **set-reduce-keyed**.

As mentioned, this program is designed to test dynamic queue sets and instanced Thread stages. Thus, it has a singleton Thread stage ("generate") to generate synthetic data, a single dynamic queue set ("data-set"), and an instanced Thread stage consumer ("reduce-huge"). There is also a two element read-only buffer ("params") that specifies the total number of packets to generate and how many subqueues to create, which are set to 25 and 6 respectively in this example. The bodies of the two stages are described in more detail below in Section A.2.

A note about specifying stages in an application graph: in this test, each stage defines a `threadMain`, which the setup code resolves via dynamic linking. In some tests, the stages are linked statically and specified directly via function pointer instead.

## A.1.1  C++ Setup

```
/*
 * set-reduce-keyed.cpp --
 *
 * Procedural GRAMPS setup code for set-reduce-keyed
 */

#include <stdio.h>
#include <string.h>
#include <dlfcn.h>

#include "gramps.h"

GrThreadProgram
GetStageMain(const char *fileName) {
    void *progHandle, *progSym;

    if ((progHandle = dlopen(fileName, RTLD_LAZY)) == NULL) {
        printf("Unable to open %s:  %s\n", fileName, dlerror());
        return NULL;
    }

    if ((progSym = dlsym(progHandle, "threadMain")) == NULL) {
        printf("Unable to find threadMain in %s:  %s\n", fileName, dlerror());
        dlclose(progHandle);
        return NULL;
    }

    return (GrThreadProgram) progSym;
}

int
main(int argc, const char *argv[]) {

    /* Create buffer with parameters */
    GrBufferDesc bufDesc;
```

```
    GrBufferId bufId;
    int bufData[] = { 25, 6 }; /* numOutputs, numLanes */

    snprintf(bufDesc.name, GR_MAX_NAME_LEN, "params");
    bufDesc.numBytes = sizeof bufData;
    bufId = GrCreateBuffer(&bufDesc, bufData);

    GrBufferBindDesc bufBind;
    bufBind.grb = bufId;
    bufBind.mode = GR_BUFFER_BIND_RD;

    /* Create queue set */
    GrQueueDesc qDesc;
    GrQueueId qId;

    memset(&qDesc, 0, sizeof qDesc);
    snprintf(qDesc.name, GR_MAX_NAME_LEN, "data-set");
    qDesc.numPackets = 40;
    qDesc.packetByteWidth = 4;
    qDesc.exclusive = true;
    qId = GrCreateQueue(&qDesc);

    /* Create stages */
    GrStageDesc stageDesc[2];
    GrStageId stageId[2];

    memset(stageDesc, 0, sizeof stageDesc);

    snprintf(stageDesc[0].name, GR_MAX_NAME_LEN, "generate");
    stageDesc[0].program = GetStageMain("./generate-keyed.gre.so");
    stageDesc[0].type = GR_STAGE_ASSEMBLE;
    stageId[0] = GrCreateStage(&stageDesc[0]);
    GrBindQueues(stageId[0], GR_QUEUE_BIND_OUTPUT, 1, &qId);
    GrBindBuffers(stageId[0], 1, &bufBind);

    snprintf(stageDesc[1].name, GR_MAX_NAME_LEN, "reduce-huge");
    stageDesc[1].program = GetStageMain("./reduce-huge-reserve.gre.so");
    stageDesc[1].type = GR_STAGE_ASSEMBLE;
    stageDesc[1].instanced = true;
    stageId[1] = GrCreateStage(&stageDesc[1]);
    GrBindQueues(stageId[1], GR_QUEUE_BIND_INPUT, 1, &qId);
    GrBindBuffers(stageId[1], 1, &bufBind);

    /* Run the execution graph */
    GrWaitId graphId;

    graphId = GrSetRunnable(1, &stageId[0]);
    GrWait(graphId);

    /* Cleanup */
    GrDestroyBuffer(bufId);
    GrDestroyQueue(qId);
    GrDestroyStage(stageId[0]);
    GrDestroyStage(stageId[1]);
    return 0;
}
```

## A.1.2   Grampsh Setup

```
#!../grampsh/grampsh
#
# set-reduce-keyed.cfg --
#
# grampsh script for the set-reduce test.  This version uses dynamic
# subqueues and an instanced Thread stage.

###
# Stages
#
numStages = 2

stage0.name = generate
stage0.program = generate-keyed.gre
stage0.type = thread
stage0.runnable = 1
stage0.numBufs = 1
stage0.numOutputs = 1

stage0.buf0.name = params
stage0.buf0.mode = read
stage0.output0.name = data-set

stage1.name = reduce-huge
stage1.program = reduce-huge-reserve.gre
stage1.type = thread
stage1.instanced = 1
stage1.numBufs = 1
stage1.numInputs = 1

stage1.buf0.name = params
stage1.buf0.mode = read
stage1.input0.name = data-set

###
# Queues
#
data-set.numPackets = 40
data-set.packetSize = 4
data-set.numLanes = 0
data-set.exclusive = 1


###
# Buffers
#

# Note:  The first int is the number of outputs and the second is the number
# of queue set lanes to use.
params.numBytes = 8
params.contents = 25 6
```

| Thread Stages |
|---|
| `uint32 GrReserve(window, numPackets, subQueue);` |
| `uint32 GrReserveKey(window, numPackets, key);` |
| `uint32 GrCommit(window, numPackets, flags);` |
| `uint32 GrGetQueueKey(window);` |
| **Shader Stages** |
| `void GrPush(queue, subQueue, data[]);` |
| `void GrPushKey(queue, key, data[]);` |
| `uint32 GrGetInputKey(void);` |
| **All Stages** |
| `uint32 GrPrintf(format, ...);` |
| `void GrAssert(condition);` |

Table A.2: GRAMPS APIs for Thread and Shader stages

## A.2   Stages

Table A.2 lists the GRAMPS intrinsics available to Thread and Shader stages: the appropriate queue operations for each type, the ability to determine which subqueue of a dynamic queue set is bound, and basic debugging functions.

The two Thread stages of **set-reduce-keyed**, listed below, each occupy their own file: generate in generate-keyed.c and reduce-huge in reduce-huge-reserve.c. As described above, the entry-point for each stage is named `threadMain`.

Generate is very simple: it iterates a counter from 0 to $numVals - 1$, makes a packet whose sole contents are the counter value, and uses that value modulo $numLanes$ as the key to select the output subqueue. Reduce-huge is only slightly more complex. Each instance issues a "huge" reservation (-1 packets) to a GRAMPS-chosen subqueue. This reservation blocks until the entire input is available, at which point the stage traverses that input verifying that the values are the ones that correspond to the key. It infers the subqueue key from the data, but it could also use `GrGetQueueKey`.

## A.2.1 Generate-Keyed

```
/*
 * generate-keyed.c --
 *
 * Simple kernel that generates the requested number of ints and
 * distributes them round-robin among the lanes of its output queue
 * set.  This version of the kernel explicitly creates a new subqueue
 * according to its input parameters rather than assuming they were
 * created statically.
 */

#include "grampsthread.h"

void
threadMain(GrEnv* env)
{
    uint32 *params = (uint32 *) env->buffers[0].mem;
    uint32 numVals, numLanes, ii;

    GrAssert(env->buffers[0].numBytes / sizeof(uint32) >= 2);
    numVals = params[0];
    numLanes = params[1];

    GrPrintf("Generate:  Spreading %d values among %d lanes.\n", numVals, numLanes);
    for (ii = 0; ii < numVals; ii++) {
        uint32 lane;

        lane = ii % numLanes;
        GrReserveKey(&env->outputQueues[0], 1, lane);
        *((uint32 *) env->outputQueues[0].mem) = ii;
        GrCommit(&env->outputQueues[0], 1, GR_RESERVE_FLAG_ANYSUBQUEUE);
    }

    GrPrintf("Generate:  Complete\n");
}
```

## A.2.2 Reduce-Huge-Reserve

```
/*
 * reduce-huge-reserve.c --
 *
 * Simple kernel that buffers its input with a gigantic reservation,
 * relying upon it to block and then return GR_RESERVE_SHORT when the
 * input closes.
 */

#include "grampsthread.h"

void
threadMain(GrEnv* env)
{
    uint32 *params = (uint32 *) env->buffers[0].mem;
    GrQueueWin *input = &env->inputQueues[0];
```

```
    uint32 numLanes, curLane;

    GrAssert(env->buffers[0].numBytes / sizeof(uint32) >= 2);
    numLanes = params[1];

    for (curLane = 0; curLane < numLanes; curLane++) {
        uint32 result, laneNo, numPackets, ii;

        GrPrintf("Reduce:  Reducing subqueue %d\n", curLane);

        result = GrReserve(input, (uint32) -1, GR_RESERVE_FLAG_ANYSUBQUEUE);
        if (result == GR_RESERVE_NOMORE) {
            continue;
        }
        numPackets = input->numBytes / sizeof(uint32);

        /* Compute the expected subqueue number based on the first element.  */
        laneNo = ((uint32 *) input->mem)[0];

        GrPrintf("Reduce:  Subqueue %d has %d packets.\n", laneNo, numPackets);
        for (ii = 0; ii < numPackets; ii++) {
            uint32 expected = ii * numLanes + laneNo;
            uint32 actual = ((uint32 *) input->mem)[ii];

            if (expected != actual) {
                GrPrintf("Reduce:  Queue[%d][%d] was %d when expecting %d!\n",
                    laneNo, ii, actual, expected);
                GrAssert(expected == actual);
            }
        }

        GrCommit(&env->inputQueues[0], numPackets, GR_RESERVE_FLAG_ANYSUBQUEUE);
    }
}
```

# Bibliography

[1] AMD. Coming soon: The AMD Fusion Family of APUs. http://sites.amd.com/us/fusion/APU/Pages/fusion.aspx.

[2] AMD. ATI Radeon HD 5000 Series Graphics Cards from AMD, 2010. http://www.amd.com/us/products/desktop/graphics/ati-radeon-hd-5000/Pages/ati-radeon-hd-5000.aspx.

[3] Nimar S. Arora, Robert D. Blumofe, and C. Greg Plaxton. Thread scheduling for multiprogrammed multiprocessors. In *Proc. of the 10th annual ACM Symp. on Parallel Algorithms and Architectures*, 1998.

[4] Louis Bavoil, Steven P. Callahan, Aaron Lefohn, Joao L. D. Comba, and Claudio T. Silva. Multi-fragment effects on the GPU using the k-buffer. In *Proceedings of the 2007 symposium on Interactive 3D graphics and games*, pages 97–104, New York, NY, USA, 2007. ACM.

[5] David Blythe. The Direct3D 10 system. *ACM Transactions on Graphics*, 25(3):724–734, July 2006.

[6] Solomon Boulos, Dave Edwards, J. Dylan Lacewell, Joe Kniss, Jan Kautz, Peter Shirley, and Ingo Wald. Packet-based Whitted and distribution ray tracing. *Proceedings of Graphics Interface 2007*, pages 177–184, 2007.

[7] Alton Brown. The Chewy. Food Network. http://www.foodnetwork.com/recipes/alton-brown/the-chewy-recipe/index.html.

[8] David Chase and Yossi Lev. Dynamic circular work-stealing deque. In *Proc. of the 17th annual ACM Symp. on Parallel Algorithms and Architectures*, 2005.

[9] Shuai Che, Michael Boyer, Jiayuan Meng, David Tarjan, Jeremy W. Sheaffer, Sang-Ha Lee, and Kevin Skadron. Rodinia: A benchmark suite for heterogeneous computing. In *Proc. of the IEEE Intl. Symp. on Workload Characterization*, 2009.

[10] Jiawen Chen, Michael I. Gordon, William Thies, Matthias Zwicker, Kari Pulli, and Frédo Durand. A reconfigurable architecture for load-balanced rendering. In *Workshop on Graphics Hardware*, pages 71–80, New York, NY, USA, 2005. ACM.

[11] William J. Dally, Francois Labonte, Abhishek Das, Patrick Hanrahan, Jung-Ho Ahn, Jayanth Gummaraju, Mattan Erez, Nuwan Jayasena, Ian Buck, Timothy J. Knight, and Ujval J. Kapasi. Merrimac: Supercomputing with streams. In *SC '03: Proceedings of the 2003 ACM/IEEE conference on Supercomputing*, page 35, Washington, DC, USA, 2003. IEEE Computer Society.

[12] Abhishek Das, William J. Dally, and Peter Mattson. Compiling for stream processing. In *Proc. of the 15th Intl. Conf. on Parallel Architectures and Compilation Techniques*, 2006.

[13] Alejandro Duran, Julita Corbalán, and Eduard Ayguadé. Evaluation of OpenMP task scheduling strategies. In *4th Intl. Workshop in OpenMP*, 2008.

[14] Tim Foley and Jeremy Sugerman. KD-tree acceleration structures for a GPU raytracer. In *Workshop on Graphics Hardware*, pages 15–22, New York, NY, USA, 2005. ACM.

[15] Matteo Frigo, Charles E. Leiserson, and Keith H. Randall. The implementation of the Cilk-5 multithreaded language. In *Proc. of the ACM SIGPLAN conf. on Programming Language Design and Implementation*, 1998.

[16] Michael I. Gordon, William Thies, Michal Karczmarek, Jasper Lin, Ali S. Meli, Andrew A. Lamb, Chris Leger, Jeremy Wong, Henry Hoffmann, David Maze, and Saman Amarasinghe. A stream compiler for communication-exposed architectures. In *Proc. of the 10th Intl. Conf. on Architectural Support for Programming Languages and Operating Systems*, 2002.

[17] Roy Hall and Donald Greenberg. A testbed for realistic image synthesis. *IEEE Comput. Graph. Appl.*, 3(8):10–20, 1983.

[18] Jon Hasselgren and Thomas Akenine-Möller. PCU: the programmable culling unit. *ACM Transactions on Graphics*, 26(3):92, 2007.

[19] W. Daniel Hillis and Guy L. Steele, Jr. Data parallel algorithms. *Communications of the ACM*, 29(12):1170–1183, 1986.

[20] Daniel Horn, Jeremy Sugerman, Mike Houston, and Pat Hanrahan. Interactive k-D Tree GPU Raytracing. In *Proceedings of the 2007 symposium on Interactive 3D graphics and games*, New York, NY, USA, 2007. ACM.

[21] Intel. TBB http://www.threadingbuildingblocks.org.

[22] Brucek Khailany, William J. Dally, Ujval J. Kapasi, Peter Mattson, Jinyung Namkoong, John D. Owens, Brian Towles, Andrew Chang, and Scott Rixner. Imagine: Media Processing with Streams. *IEEE Micro*, 21:35–46, 2001.

[23] Khronos Group. OpenCL 1.0 specification, 2009.

[24] Manjunath Kudlur and Scott Mahlke. Orchestrating the execution of stream programs on multicore platforms. In *Proc. of the ACM SIGPLAN conf. on Programming language design and implementation*, 2008.

[25] Edward Ashford Lee and David G. Messerschmitt. Static scheduling of synchronous data flow programs for digital signal processing. *IEEE Trans. Comput.*, 36(1), 1987.

[26] John M. Mellor-Crummey and Michael L. Scott. Algorithms for scalable synchronization on shared-memory multiprocessors. *ACM Trans. Comput. Syst.*, 9(1), 1991.

[27] MIPS Technologies Inc. MIPS64 architecture, 2005. http://mips.com/products/architectures/mips64/.

[28] Mozilla. Gecko plugin API reference (NPAPI). https://developer.mozilla.org/en/Gecko_Plugin_API_Reference.

[29] David R. Musser and Atul Saini. *The STL Tutorial and Reference Guide: C++ Programming with the Standard Template Library.* Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, USA, 1995.

[30] NVIDIA. CUDA SDK Code Samples http://developer.nvidia.com/object/cuda_sdk_samples.html.

[31] NVIDIA. CUDA 3.0 reference manual, 2010.

[32] NVIDIA. NVIDIA GF100 World's Fastest GPU Delivering Great Gaming Performance with True Geometric Realism, 2010. http://www.nvidia.com/object/IO_89569.html.

[33] Marc Olano and Trey Greer. Triangle scan conversion using 2D homogeneous coordinates. In *HWWS '97: Proceedings of the ACM SIGGRAPH/EUROGRAPHICS workshop on Graphics hardware*, pages 89–95. ACM, 1997.

[34] John D. Owens, Brucek Khailany, Brian Towles, and William J. Dally. Comparing Reyes and OpenGL on a stream architecture. In *Workshop on Graphics Hardware*, pages 47–56, September 2002.

[35] D. Pham, S. Asano, M. Bolliger, MN Day, HP Hofstee, C. Johns, J. Kahle, A. Kameyama, J. Keaty, Y. Masubuchi, et al. The design and implementation of a first-generation CELL processor. *ISSCC. 2005 IEEE International*, pages 184–186, 2005.

[36] Timothy J. Purcell. *Ray Tracing on a Stream Processor*. PhD thesis, Stanford University, 2004.

[37] Colby Ranger, Ramanan Raghuraman, Arun Penmetsa, Gary Bradski, and Christos Kozyrakis. Evaluating mapreduce for multi-core and multiprocessor systems. In *Proc. of the 13th Intl. Symp. on High-Performance Computer Architecture*, 2007.

[38] John R. Rose and Guy L. Steele, Jr. C*: An extended C language for data parallel programming. In *Proceedings of the Second International Conference on Supercomputing*, volume II, pages 2–16. International Supercomputing Institute, 1987.

[39] Mark Segal and Kurt Akeley. The OpenGL graphics system: A specification, 2010. [http://www.opengl.org/registry/doc/glspec40.core.20100311.pdf](http://www.opengl.org/registry/doc/glspec40.core.20100311.pdf).

[40] Larry Seiler, Doug Carmean, Eric Sprangle, Tom Forsyth, Michael Abrash, Pradeep Dubey, Stephen Junkins, Adam Lake, Jeremy Sugerman, Robert Cavin, Roger Espasa, Ed Grochowski, Toni Juan, and Pat Hanrahan. Larrabee: A Many-Core x86 Architecture for Visual Computing. *ACM Transactions on Graphics*, 27(3), 2008.

[41] Jeremy Sugerman, Kayvon Fatahalian, Solomon Boulos, Kurt Akeley, and Pat Hanrahan. GRAMPS: A programming model for graphics pipelines. *ACM Trans. Graph.*, 28(1), 2009.

[42] Jeremy Sugerman, David Lo, Richard Yoo, Daniel Sanchez, and Christos Kozyrakis. Comparing parallel programming models using GRAMPS. *Under submission*, Aug 2010.

[43] William Thies, Michal Karczmarek, and Saman Amarasinghe. StreamIt: A language for streaming applications. In *Proc. of the 10th Intl. Conf. on Compiler Construction*, 2001.

[44] Elliot Waingold, Michael Taylor, Devabhaktuni Srikrishna, Vivek Sarkar, Walter Lee, Victor Lee, Jang Kim, Matthew Frank, Peter Finch, Rajeev Barua, Jonathan Babb, Saman Amarasinghe, and Anant Agarwal. Baring it all to software: Raw machines. *IEEE Computer*, 30(9):86 –93, 1997.

[45] Wikipedia. SQL. http://en.wikipedia.org/wiki/SQL.

[46] Kun Zhou, Qiming Hou, Zhong Ren, Minmin Gong, Xin Sun, and Baining Guo. RenderAnts: interactive Reyes rendering on GPUs. In *SIGGRAPH Asia '09: ACM SIGGRAPH Asia 2009 papers*, pages 1–11, New York, NY, USA, 2009. ACM.