The Report Committee for Sean Edward Stephens
Certifies that this is the approved version of the following Report:

The Parla Runtime

APPROVED BY
SUPERVISING COMMITTEE:

Mattan Erez, Supervisor

Christopher J. Rossbach

The Parla Runtime

by

Sean Edward Stephens

Report

Presented to the Faculty of the Graduate School of

The University of Texas at Austin

in Partial Fulfillment

of the Requirements

for the Degree of

Master of Science in Engineering

The University of Texas at Austin

May 2022

Dedication

I dedicate this report to my family. My parents, Bobby and Tangie, have been incredibly loving and supportive throughout my academic career. My siblings, Rob and Blake, have helped me so much in navigating academia and life in general, and without them I cannot imagine taking on the challenges of graduate school.

I also dedicate this report to my friends. In particular, I dedicate it to Julia, who has inspired me to be the best person and engineer I can be on a day-to-day basis throughout graduate school; to Grant and Thomas, who have never failed to reach out for a conversation when times are difficult and for a game when they are not; to Mark, who has always offered words of encouragement when they are most needed; and to Ben, who, throughout this entire experience, has been my anchor.

Acknowledgements

My work would not be possible without the momentous efforts of my fellow students on the Parla team. I owe a great deal of thanks to them, particularly to: Hochan Lee for his collaboration in designing and implementing the runtime system; William Ruys for his insights throughout the design process and for his tireless work on improving so many facets of Parla; Arthur Peters and Ian Henriksen for their guidance in learning about Parla; and Yineng Yan and Bozhi You for their designs and implementation of Parla features closely integrating with the runtime. Without these students, Parla would not exist.

I would like to thank all the professors on the Parla team for their support throughout this research undertaking. In particular, the guidance of Christopher Rossbach in designing the runtime has been incredibly helpful.

Special thanks goes to Mattan Erez, my adviser, who has been a wonderful mentor throughout graduate school. He has offered much-welcomed support in terms of engineering and research insights as well as generous funding to make my research possible, and he has been a pleasure to work with over the past two years.

I would also like to thank The University of Texas at Austin Cockrell School of Engineering, the donors of the Temple Foundation MCD Fellowship, and the Predictive Science Academic Alliance Program, without whose funding support I would not have been able to focus all my efforts on invaluable learning and research experiences at such a great university.

Abstract

The Parla Runtime

Sean Edward Stephens, MSE

The University of Texas at Austin, 2022

Supervisor: Mattan Erez

Writing high-performance programs to target heterogeneous compute nodes poses many challenges associated with properly managing data-level and task-level parallelism across various processing units. Parla is a heterogeneous task-based programming framework which simplifies writing portable multi-device code by enabling programmers to leverage task-level parallelism with simple decorator annotations while fully utilizing Python's rich scientific programming stack.

The underlying runtime system of Parla must support the efficient execution of a variety of task graphs on complex heterogeneous nodes. This runtime is divided into three phases: mapper, scheduler, and launcher. I present the design of each phase and discuss the motivation behind design decisions, with particular attention to the performant treatment of GPU tasks. I show that the current runtime's heuristic-based mapping policies run similarly well to optimal user-specified mappings on a variety of workloads. Lastly, I detail many areas of future work to further improve the runtime performance.

Table of Contents

List of Figures

List of Code Listings

Chapter 0: Foreword

This Master's Report serves as documentation of research on Parla. Parla is a tasking system in Python targeting heterogeneous compute nodes; in other words, it is a Python library which makes it easier to write parallel code for computers with server-class CPUs and multiple GPUs. My research focus has been to improve the performance of Parla's underlying runtime system. This report satisfies my MSE degree research requirements and serves as documentation for the motivation behind and design of the Parla runtime system, particularly the Parla scheduler.

Chapter 1: Introduction

Writing performant software to harness the processing power of parallel compute architectures requires a great deal of experience. From the programmer's perspective, software can be written to take advantage of two types of parallelism: data parallelism, where the same operation is performed on multiple pieces of data concurrently; and task parallelism, where different functions (called tasks) are distributed over the hardware to be run in parallel.

Properly exploiting data parallelism can be complex. For example, when writing parallel code targeting Nvidia GPUs, programmers write kernels—high performance functions—in the CUDA programming language. To implement an optimized CUDA kernel the programmer must be familiar with intricacies of the GPU memory hierarchy, avoid issues stemming from thread divergence, take advantage of the latest architectural features of the GPU generation being programmed, etc. To alleviate this burden, many high-performance scientific libraries in various programming languages have been developed which abstract away these issues from the programmer.

Manually managing task parallelism also requires some expertise. Even in the simple case of a single multi-core processor, the programmer must take care to prevent race conditions between threads, enforce a proper ordering of tasks, exchange correct data between threads, and avoid deadlock. To address these complications, several tasking systems exist which enable the programmer to explicitly define tasks and their dependencies, forming a task graph which the tasking system's runtime executes without the programmer having to worry about manually managing concurrency.

Modern high-performance computers are generally clusters consisting of multiple heterogeneous nodes—that is, multiple CPUs each paired with their own set of

accelerators, typically GPUs. Properly leveraging the processing power of even a single node requires exploitation of both data- and task-level parallelism. Even with existing data-parallel libraries and task-parallel systems, combining management of both forms of parallelism has proven to still be a major challenge. Multi-GPU management, in particular, requires a great deal of attention to detail from the programmer. To maximize performance in such a machine one must efficiently distribute work across multiple GPUs, co-locate work within the context of a given device, launch asynchronous concurrent data copies and compute kernels on various devices, properly synchronize dependent kernels across CUDA streams, etc. Managing these facets while having them cooperate with diverse libraries is a challenge that programmers shy away from, particularly non-experts who are not keen on diving into the intricacies of multi-accelerator management.

Of course, several existing tasking systems have attempted to address these difficulties by incorporating GPU management natively into their task workflow. All modern tasking systems fall short in one or more of these aspects:

- They do not manage GPU computations at all, or they manage them naively.
- They are not truly general purpose, instead only supporting a specific set of parallel operations.
- They are difficult to adopt and use.

I will discuss several such systems in Chapter 6; for now, I turn to a discussion of Python before introducing Parla.

Python has grown to be a powerful and versatile programming language with a rich ecosystem of scientific computing modules for processing, analyzing, graphing, and reporting data. Libraries like NumPy [1], CuPy [2], and others enable programmers to stitch together highly-optimized data-parallel functions targeting various architectures to

rapidly develop powerful scientific applications. Further, custom kernels may be implemented using just-in-time (JIT) compiled functions via Numba [3]. As such, it is a suitable choice of language in which to incorporate a tasking system to address the above three shortcomings, given that a plethora of heterogeneous HPC (high-performance computing) data-parallel libraries already exist, general-purpose kernels are supported, and it is known for being productive to code in.

Parla is a Python-based tasking system which embraces heterogeneity in high-performance computing applications and introduces an *orchestration layer* to control and connect library functions and kernels within a single process and address space. Whereas these functions and kernels are often device- and multi-device-specific to obtain high performance, the orchestration code is device-agnostic and describes the overall structure of the program.

To increase usability, a primary design goal of Parla is gradual adoption. Parla focuses on intra-node architectures and interoperability with existing Python frameworks and libraries. Parla does not require the user to port an entire application to a new framework; rather, converting sequential programs to task-based Parla applications can be done by gradually wrapping individual portions of the program in Parla tasks as the programmer sees fit. Parla provides lightweight wrappers to allow users to take advantage of familiar data structures within tasks without manually managing data movement between devices. These features enable rapid prototyping of parallel and heterogeneous applications using a familiar Python ecosystem.

My research focus has been on increasing the performance of the underlying runtime system of Parla, which is responsible for ensuring that task dependency requirements are met, mapping tasks to compute devices, and launching tasks on worker threads for execution. My contributions are:

- Design of a three-phase runtime divided into a mapper, scheduler, and launcher for efficiently handling task execution on heterogeneous compute nodes.
- Design of a heuristics-based mapper policy accounting for data locality and load balancing when choosing task placements.
- Design of mechanisms for effectively handling GPU tasks in Python.
- Implementation and evaluation of a significant portion of the above three designs in collaboration with other students on the Parla team.

The rest of this report is organized as follows: In Chapter 2, I provide more details on the Parla tasking system so that the reader may understand its basic usage, setting a framework for the underlying runtime discussion. In Chapter 3 I discuss the design challenges and implementation of the Parla runtime, my main research contribution. Chapter 4 contains an evaluation of the runtime, Chapter 5 discusses ideas for future work, Chapter 6 discusses related work with a focus on comparing the Parla runtime to that of other tasking systems, and Chapter 7 concludes the report.

Chapter 2: The Parla Tasking System

This chapter describes the basic features and implementation of Parla. An in-depth system description is elided from this report for brevity, as the focus is on the underlying runtime system covered in Chapter 3. This chapter serves as an overview to set the framework for the runtime system discussion. I discuss Parla tasks, Parla's managed array class, and the interfaces to control task behavior, then tie these concepts together by walking through a simple example using various features.

PARLA TASKS

In Parla, *tasks* are arbitrary blocks of Python code that run asynchronously with respect to the enclosing block. Provided that sufficient resources are available, tasks run in parallel with each other. Parla uses Python decorators for task creation. To create a task, the programmer encapsulates a code block with an in-place function and annotates the function with Parla's `@spawn()` decorator, as follows:

```
@spawn()
def task_name():
     # Code block goes here
```

Task creation can be data dependent or otherwise conditional, allowing Parla to handle arbitrary irregular and dynamic parallelism that is decided at runtime.

Parla tasks execute on separate threads within a single process. This execution model has the advantage of keeping communication between tasks lightweight and easy, as a program's variables are all accessible by any task since they all execute in the same address space. On the other hand, this does mean that tasks contend for the Global Interpreter Lock (GIL). The GIL is a limitation of CPython (the most common implementation of the Python language) which permits only one thread to execute pure

Python code at any given time, limiting concurrent execution by serializing threads. A thread's forward execution may continue without the GIL, though, if it enters a section of compiled code. These compiled sections allow the thread to release the GIL, enabling other threads to acquire it to perform Python computations. In practice, all high-performance computation in modern Python is done through JIT-compiled kernels (such as through Numba [3]), a mix of static and dynamic kernel compilation (like in PyKokkos [4]), user wrapped kernels from a lower-level language (for example via a Cython [5] interface to C++), or in Python modules such as NumPy [1] and CuPy [2] that hide away these pre-compiled routines. During these calls, the GIL can be released allowing tasks to schedule and execute in parallel. Thus, any Parla user is encouraged to make use of these compiled code options to enable their tasks to properly run in parallel. For applications targeted by Parla, calling into such code is the common case and the GIL does not pose a major impediment to harnessing task parallelism.

THE PARLA ARRAY (PARRAY)

Efficient data management is a key challenge for performance on heterogeneous systems, both in terms of programmability and performance. Existing data structures like NumPy or CuPy n-dimensional arrays (ndarrays) are designed for a single device and provide limited information to the runtime. Parla introduces a wrapper for ndarrays called the Parla Array, or PArray. A PArray is a single logical ndarray with copies on one or more devices. The copies follow a coherence protocol. If the ndarray is copied from one device to another without being modified, both buffers are valid and can be read by tasks. Once the array is modified on any device, that particular device's copy becomes the only up-to-date copy, and other copies are invalidated. The intricacies of this protocol are beyond the scope of this report.

In the spirit of gradual adoption, PArrays are an optional Parla feature and are not required for Parla tasks. Programmers may use their own objects within tasks if they so choose. PArrays do, however, provide advantages for the programmer both in terms of performance and programmability. From a programmability standpoint, using a PArray relieves the user of explicitly needing to track which device(s) and architecture(s) an array exists on. Parla will always provide a valid copy of the array in any given Parla task. PArrays provide performance advantages because the runtime is directly able to manage the movement of PArrays, enabling it to automatically prefetch data to devices as well as to make more informed task mapping decisions based on data locality. These features will be discussed in more detail in Chapter 3. An expert Parla user may choose to forego PArrays and can achieve the same benefits through proper implementation of prefetching tasks and manual mapping policies, but this practice is not expected to be the common case.

CONTROLLING TASK BEHAVIOR

Task behavior can be controlled using parameters of the `@spawn()` decorator. Tasks may be assigned a unique Task ID. Task IDs are useful for identifying tasks when supplying arguments to other parameters. Tasks may also be grouped into a Task Space, an n-dimensional indexable collection of tasks that can be used to organize and refer to the tasks later. A Task ID may be a string or an element from a Task Space and is assigned using the `taskid` parameter.

Dependencies between tasks may be expressed using the `dependency` parameter. Task IDs and Task Spaces may be assigned to this parameter to enforce ordering between tasks. Parla tasks may begin executing as soon as they are spawned and their dependencies have finished executing.

There is a `placement` argument to specify the mapping of a task. Tasks may be mapped to a specific device, a specific architecture, the location of a particular piece of data, or the location of another task. The tasking runtime determines which device is used from the set of valid placements. Currently, tasks support placement on two architectures: CPUs and CUDA-capable GPUs. Placing a task on a GPU notifies the runtime that the task should be treated as a GPU task, allocating resources on the device and receiving special context from the runtime such as a dedicated CUDA stream and different mapping considerations (detailed in Chapter 3). GPU tasks should be thought of as code that launches one or several GPU kernel(s) to process device-side data. Given the nature of calling GPU code in Python, some amount of work does occur on the CPU during a GPU task, e.g., to call into the CuPy runtime and asynchronously launch kernels. However, these CPU-side routines are trivial, only briefly acquiring the GIL on a single core during kernel launches, so CPU-side resources are not provisioned to handle GPU tasks.

When using PArrays, a task's input and output data may be specified using the `input`, `output`, and `inout` parameters. Assigning a PArray to these parameters notifies the runtime that the ndarray is to be prefetched to the device and enables the user to reference the ndarray within the task without worrying about which device the task runs on. Additionally, the mapper is notified of the memory usage of these ndarrays so that it can ensure that no device runs out of memory when allocating resources on the device. There is an additional `memory` parameter which informs the mapper of any out-of-place memory used by a task, that is, memory associated with kernel calls and with data not contained in PArrays.

INNER PRODUCT EXAMPLE

Listing 1 contains a very simple example inner product application using several of the aforementioned features. First, in lines 1 and 2, two PArrays, `a` and `b`, are initialized on the CPU from NumPy ndarrays. Lines 4-5 initialize a list of empty PArrays to be populated by tasks as their output. An empty Task Space is created and named in line 9. The first set of tasks is spawned on line 14 inside the loop. The `@spawn()` decorator is used to create a task on each iteration. Every task in this loop may run in parallel. The Task ID is assigned a unique index into the Task Space. No dependencies are expressed. Line 15 specifies that tasks should be placed on GPU devices in a cyclic manner. Lines 16 and 17 list the PArrays to be used as inputs and outputs, respectively.

Within the task, products of the input PArrays are taken on line 19. Note that for basic arithmetic operations, PArrays can be operated upon without extracting the underlying array, but for more NumPy/CuPy function calls (as in line 28), the underlying NumPy or CuPy ndarray must be retrieved. Line 20 updates the output PArrays with the product result.

A single new task is created on line 22. No Task ID is assigned, but dependencies are expressed. The task is to wait upon the Task Space `T`, meaning that it will wait for every task in the previous loop to complete before it launches. This task is placed on the CPU in line 23 and its inputs are expressed in line 24. Within the task, the PArrays' underlying NumPy ndarrays are extracted in lines 26-27 so they may be used via a NumPy call. Note that the programmer did not need to explicitly retrieve the PArrays from the GPU to the CPU. The final result is calculated on line 28.

Thus, I have demonstrated a short application using tasks with simple dependencies, data movement managed by PArrays, and placements on various architectures using Parla semantics.

```
a = parla.asarray(numpy.random(n))
b = parla.asarray(numpy.random(n))

partial_sums = \
  [PArray() for i in range(num_gpu)]

block_size = n // num_gpu

T = TaskSpace('T')

for i in range(num_gpu):
  s = slice(i*block_size,(i+1)*block_size)

  @spawn(taskid=T[i],
         placement=gpu(i),
         input=[a[s],b[s]],
         output=partial_sums)
  def inner_local():
    prod = a[s] @ b[s]
    partial_sums[i].update(prod)

@spawn(dependencies=T,
       placement=cpu,
       input=partial_sums)
def reduce_task():
  partial_sums = \
    [parray.array for parray in partial_sums]
  result = numpy.sum(partial_sums)
```

Listing 1. Inner product in Parla.

Chapter 3: The Parla Runtime

The Parla runtime ensures that task dependency requirements are met, maps tasks to compute devices, and launches tasks on worker threads for execution. A *worker thread*, in this context, simply means the primary thread assigned to run the Python function associated with the task; tasks may acquire more threads on their own if the code they execute requests them (e.g. via a multi-threaded NumPy call). The primary goal of the Parla runtime is to maximize overall system utilization and efficiency for a variety of workloads on any heterogeneous compute node, making Parla programs both performant and portable. The Parla runtime also provides a mapping API enabling users to exploit machine- or application-specific knowledge to enhance performance. In the following sections, I explain the design principles of the Parla runtime and describe its implementation.

DESIGN PRINCIPLES

To properly leverage the compute capabilities of all devices in a heterogeneous node, the runtime is designed around two basic principles. The first is data locality: tasks ought to be scheduled near their data to minimize unnecessary data movement. The second is load balancing: work ought to be evenly distributed over devices when possible. These principles pose a tradeoff, as in complex task graphs it often proves difficult to maintain a balance of work while limiting data movement between devices.

The runtime is also designed to keep GPUs saturated with useful work. It makes use of CUDA events to map and launch GPU tasks early, keeping GPU hardware command queues full when possible. Latency is hidden by overlapping GPU computation and communication, masking the cost of data movement when it must occur. When users take advantage of PArrays, a task's data movement is decoupled from its computation,

separately scheduling copy operations to prefetch data blocks as soon as their dependencies resolve.

IMPLEMENTATION

The runtime consists of three main phases: the mapper, which chooses where to run tasks, the scheduler, which chooses when to run ready tasks whose dependencies have been completed, and the launcher, which assigns tasks to worker threads to run. These three phases run repeatedly in a tight loop in a dedicated runtime thread. At the end of each loop iteration, the scheduler sleeps, releasing the GIL for tasks to run their own Python code. I demonstrate the implementation by walking through the lifetime of a task, from spawn to completion. The basic structure of the scheduler is shown in Figure 1.

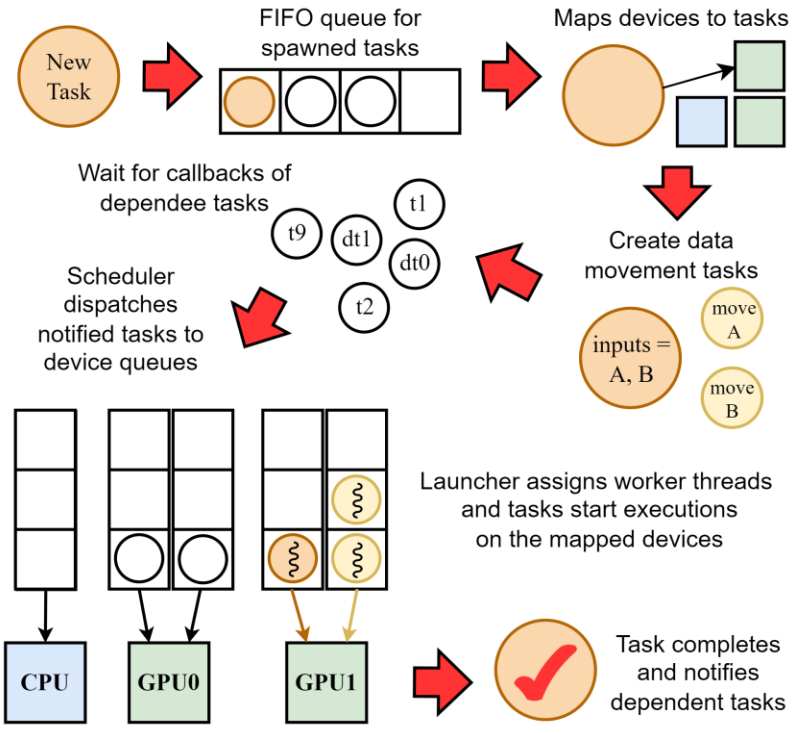

Figure 1. The Parla runtime.

When a task is spawned, it is enqueued into a FIFO queue for mapping. The runtime mapper runs periodically and uses a greedy policy to attempt to map tasks to suitable devices. To account for data locality and load balancing, the mapper calculates a *suitability* for each valid device based on various task heuristics. A *valid* device is a device which satisfies the task's placement requirements (e.g., GPUs for a GPU task, or a specific set of GPUs where a given piece of data is present)—other devices, including those with too little memory to accommodate the task, are ruled out. Simplified pseudocode for this suitability calculation is contained in Listing 2. To calculate the suitability, first, the list of PArrays required for the task is traversed to determine the amount of data the task has local to a given device as well as to estimate the cost of moving nonlocal data to that device. The device is checked for its current load based on the number of tasks already mapped to it. These factors are then added to calculate the suitability, with local data increasing suitability and nonlocal data and device load decreasing it. As such, the device receives higher priority for more data locality and lower priority if it already has a heavy load. The task is mapped to the device with the highest priority.

Occasionally, no suitable device is found (e.g., because no device has enough free memory to satisfy the task's memory requirements); in this case, the task is re-enqueued and processed the next time the mapper runs. It's worth noting that the current mapping function is somewhat rudimentary; more research needs to be done on appropriate heuristics for the suitability calculation, proper weights for these heuristics, and whether they should be combined via simple addition or via some other operation.

```
for device in task.valid_devices:
    for parray in task.input:
        if device.has_copy(parray):
            local_data += parray.nbytes
        else:
            nonlocal_data += parray.nbytes

    load = device.task_count

    device.suitability = w0 * local_data    \
                       - w1 * nonlocal_data \
                       - w2 * load_weight

    if device.owns_dependency(task):
        device.suitability += w3

best_device = max(task.valid_devices,       \
  key=attrgetter('suitability')
```

Listing 2. Mapping policy pseudocode.

Once a task is mapped, the runtime creates a child data-movement task for each of the task's PArrays which needs to be copied to the device. A data-movement task is an independent task scheduled and executed prior to its parent task for the sole purpose of gathering data to the parent task's mapped device. While the parent task must wait for all of its dependencies to complete before executing, each data movement task only needs to wait until the particular dependency producing its PArray has completed. Thus, data movement tasks can be scheduled prior to their parent computation task, effectively prefetching data before the computation task is scheduled to run.

Associated with each device is a set of FIFO queues for storing tasks ready to be launched. Once mapped, a task ordinarily waits for its dependencies to resolve, at which point the runtime scheduler dispatches it to a device queue based on its mapping. The scheduler phase dispatches tasks to device queues in a simple first-come first-serve basis. Here, the task waits until it reaches the front of the queue, at which point it is ready to be

launched. More complex scheduler designs would likely improve performance; I discuss potential improvements in Chapter 5.

The runtime maintains a pool of controllable worker threads for executing tasks. Whenever a device is free, the runtime launcher assigns a dedicated worker thread to the task at the head of the queue and begins the task's execution. Worker threads are responsible for executing user code within a task as well as for notifying dependent tasks and releasing resources back to the runtime when a task completes.

Python code executed by a worker thread does acquire the GIL, so while many worker threads may have work to do, only one runs at a given time. As previously discussed, task parallelism is achieved when tasks call into compiled code through one of a number of options provided by Python, releasing the GIL. In practice, user code in a task should strive to acquire the GIL as little as possible to maintain task-level parallelism. The final action a task takes upon completion is to return its worker thread to the runtime resource pool.

GPU TASKS

As previously stated, GPU tasks contain asynchronous CUDA kernel launches. These kernel launches are enqueued into the GPU's device-side hardware command queue for execution; keeping this queue saturated with work minimizes wasted time between kernels. As such, GPU tasks are dispatched by the Parla scheduler before their dependencies complete so they may enter the GPU hardware command queue as early as possible. Dependency ordering is enforced with CUDA events. Each GPU task records a CUDA event upon completion. If a task has a dependency, it simply waits on that dependency's recorded event at the start of its own execution. In this way, the task can be dispatched to a GPU early, waiting on the event and then asynchronously launching the

kernels it contains. The kernels run as soon as the event is observed; that is, as soon as the dependency triggers it. As such, time between kernel execution on the GPU is minimized since the device no longer must wait for tasks to be scheduled.

The runtime launcher also has special provisions for GPU tasks. Every GPU task, whether compute or data-movement, is launched on its own dedicated CUDA stream. A CUDA stream is a sequence of operations that executes in-order on the GPU, but concurrently with respect to the work in other streams. Each GPU has two dedicated queues: one for compute tasks and another for data-movement tasks. Launching tasks separately from each queue on dedicated streams permits overlap of computation with communication, fully utilizing the GPU's compute kernel and data copy engines simultaneously. Additionally, up to three compute tasks are launched at any given time, providing opportunities for co-locating small kernels on a single device when possible.

TUNING

As with all components of Parla, the runtime is designed with gradual adoption in mind. The baseline mapping policy is suitable for a variety of use cases. However, different applications and topologies will naturally result in different computation and memory access patterns, and finding one policy to fit all scenarios is difficult, if not impossible. As such, the Parla API provides means for users to leverage their own knowledge in making mapping decisions.

A user need only specify minimal information for tasks to run correctly. For example, to run a task on the GPU, a user need only specify the architecture. But if users wish to leverage application- or machine-specific to improve their mapping schemes, the Parla `@spawn()` API enables them to specify necessary memory size, particular devices, or ranges of devices. As an example, line 15 from Listing 1 demonstrates a user's ability

to map tasks within a for loop in a round-robin ordering based on the iteration count of the loop, ensuring that work is evenly distributed across devices.

Chapter 4: Evaluation

Parla has been evaluated on a range of both real and synthetic benchmarks. All data was collected on a system with 4 NVIDIA Quadro RTX 5000 GPUs and a dual-socket Intel Xeon E5-2620 (total of 16 cores, hyperthreading disabled). The pairs of RTX 5000s on the same socket are connected with Peer-to-Peer communication links. All other communication between GPUs must pass through the host machine.

In the following subsections, reported results represent the median over five runs for each given benchmark. I report runtime for strong scaling results on one, two, and four GPUs. The primary focus of these results is to evaluate the effectiveness of the runtime's underlying mapping policy, the main focus of my research. For each benchmark I report one experiment with a user-specified fixed placement policy from an experienced user and compare it to another experiment where the Parla runtime decides mappings based on heuristics as discussed in Chapter 3. All benchmarks use PArrays and take advantage of the prefetching mechanism they provide. When available, I also compare benchmarks to another non-Parla implementation as a reference.

REAL BENCHMARKS

Five real benchmarks were evaluated; I will briefly describe each benchmark's individual experimental details and discuss its results. All real benchmarks use double precision data.

The first benchmark is a blocked matrix multiplication computing $C = AB^T$ for two 32k by 32k arrays. Tasks are assigned to each blockwise multiplication. The third-party implementation is the 'cublasMg' multi-GPU matrix multiplication sample code. Uniquely for this experiment I also ran a version using a random mapping policy (*Parla*

Random) in addition the user-based (*Parla User*) and the heuristic-based (*Parla Automatic*) mappings. The results are contained in Figure 2. For this simple workload, the three Parla versions all perform relatively similarly, with the Automatic policy falling between the optimized User and the Random policies. The cublasMg implementation outperforms Parla on one GPU, likely due to its lack of Python overheads, but notably the Parla apps actually scale better when moving to four GPUs.

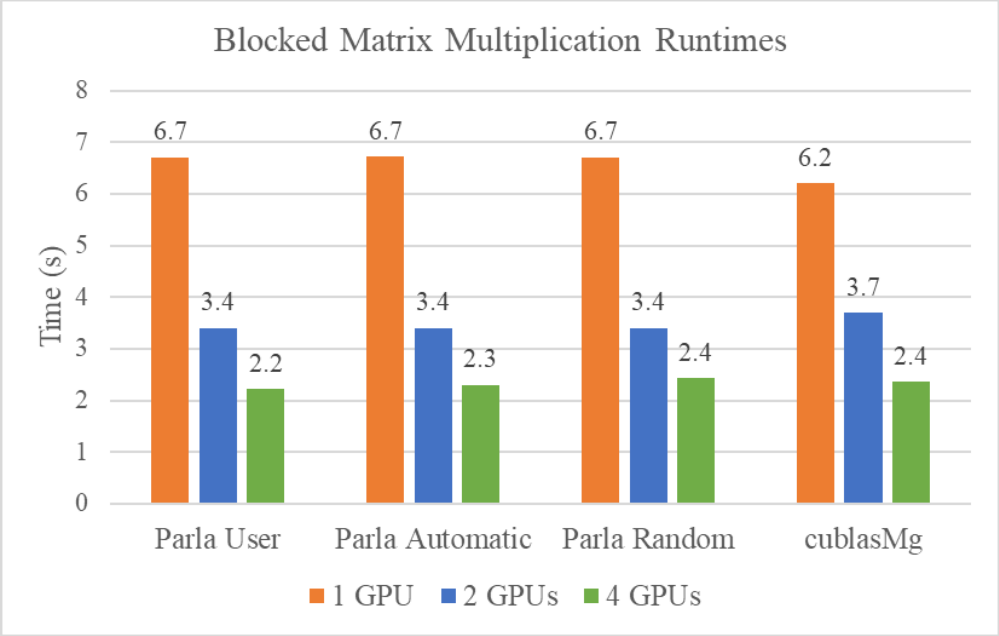

Figure 2. Blocked matrix multiplication runtimes.

The second benchmark is a 2D 4-point Jacobi stencil distributed across GPUs using a 1D block-row partitioning with tasks calling JIT-compiled Numba kernels. The benchmark is run for 500 iterations on a 30k by 30k matrix. I have no third-party comparison available for this benchmark. The results are contained in Figure 3. Similarly to matrix multiplication, we see that the User and Automatic policies perform equally well.

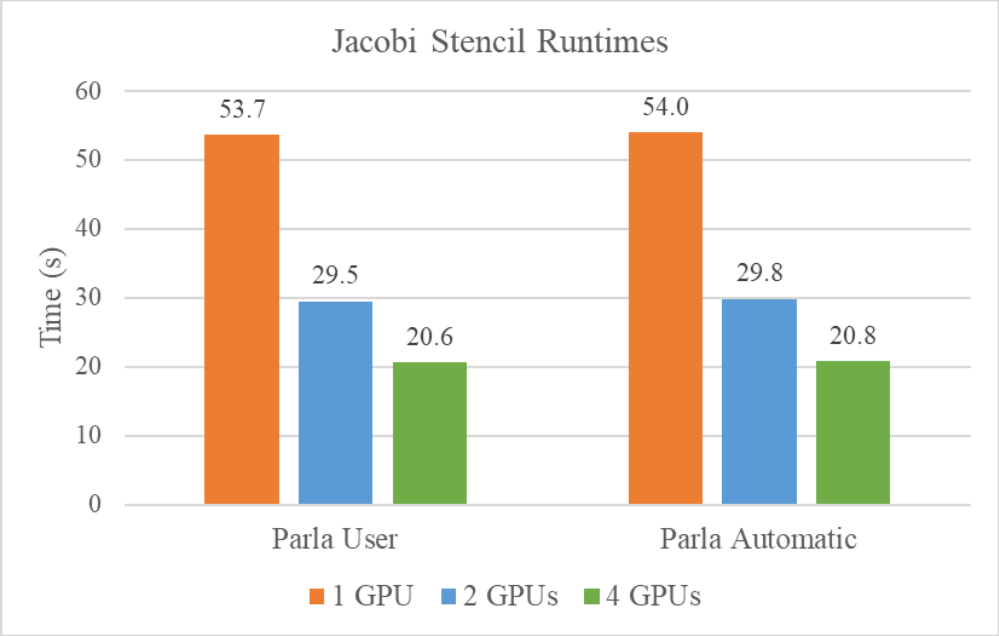

Figure 3. Jacobi stencil runtimes.

The third benchmark is a blocked Cholesky factorization computing $A = LL^T$ on a 28k by 28k matrix with block sizes of 2k. This is a common tasking benchmark as it is a simple-to-understand application with a surprisingly complicated task graph. Parla versions are compared to the performance of a left-looking multi-GPU blocked Cholesky implementation from MAGMA [6], a high-performance linear algebra library. The results are contained in Figure 4. Again, the two Parla versions perform similarly. The MAGMA version outperforms Parla both in terms of speed and scaling, running 3.57x faster from one two four GPUs while the Parla Automatic version only improves by a factor of 3.05x.

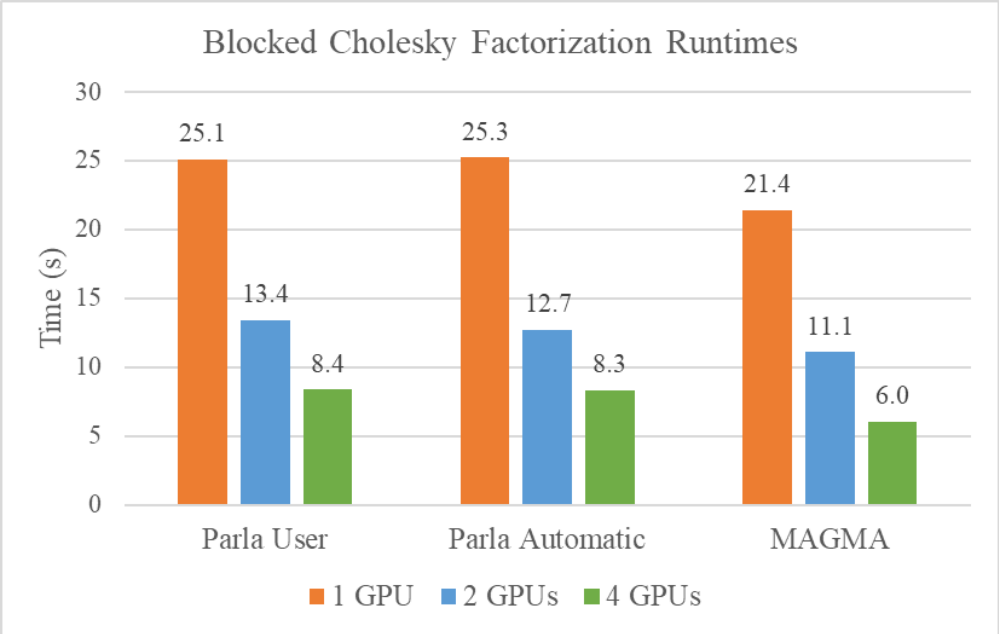

Figure 4. Blocked Cholesky factorization runtimes.

The fourth benchmark is a 2D gravitational n-body solver using a level grid decomposition implemented with Numba kernels and simulating 10M bodies. The non-Parla version for this benchmark is an in-house Python version implemented with no Parla features. Figure 5 contains the results. These results are rather encouraging, as the three versions all perform and scale similarly. For this benchmark, since all versions are written in Python, we see that the overheads incurred by Parla tasking aren't much greater than the overheads associated with writing the application in Python in the first place.

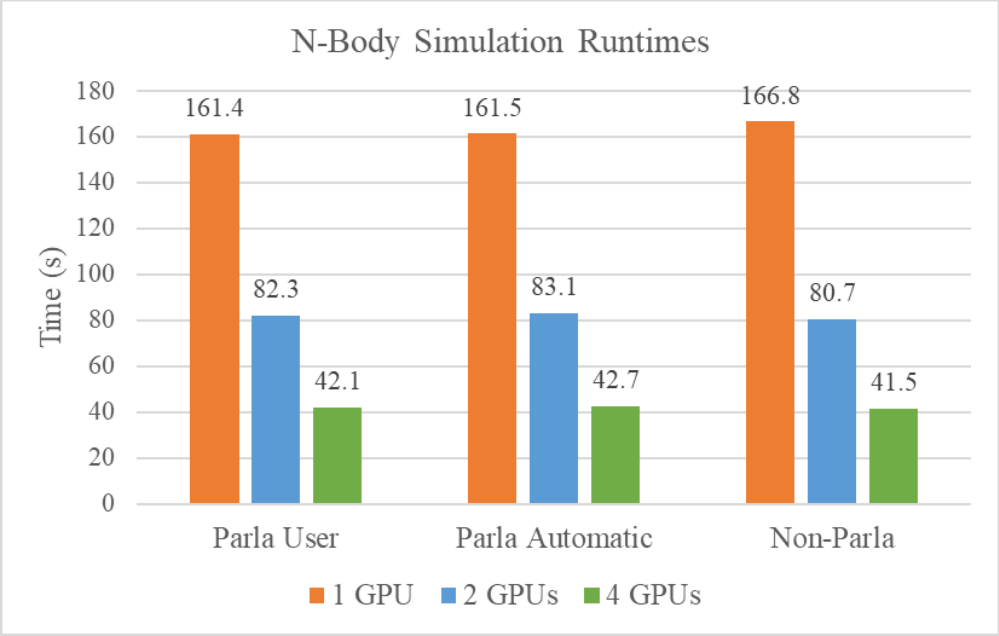

Figure 5. N-body simulation runtimes.

The fifth and final real benchmark is a Block Low-Rank (BLR) Matrix Multiplication wherein 2.5k-element blocks of a 10k-by-10k matrix are streamed across GPUs for compression into an approximate matrix which is then used to perform matrix-vector multiplication. The non-Parla version is again an in-house Python implementation. The results in Figure 6 show that yet again the User and Automatic policies perform similarly. I have not investigated the slight increase in runtime for the non-Parla version on one GPU—I would expect it to be the same as the Parla versions since the implementation is in Python—but as expected it does slightly outscale the Parla versions due to a lack of tasking overhead.

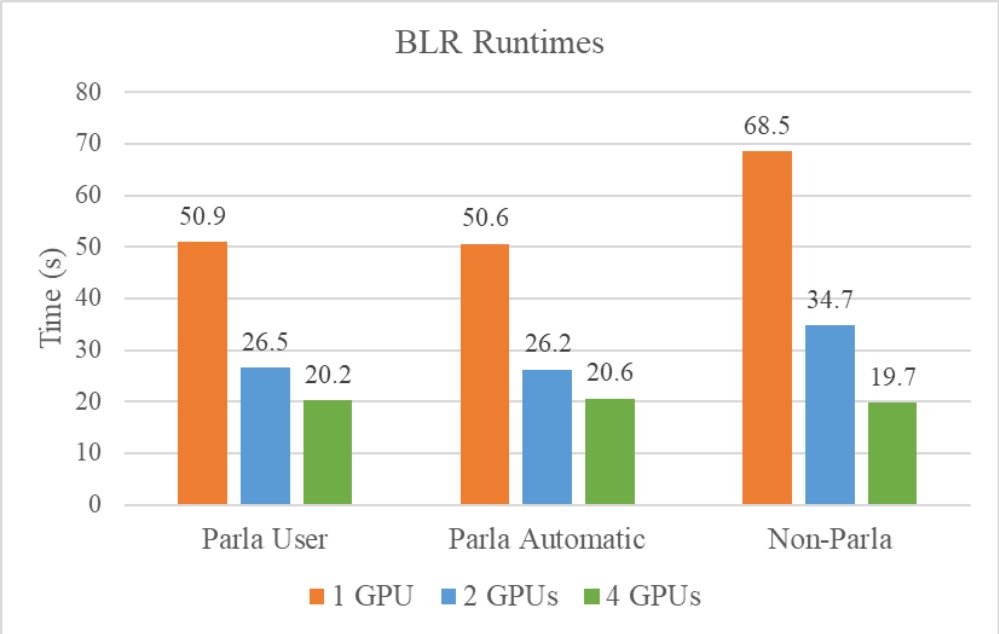

Figure 6. BLR runtimes.

In short, the User and Automatic policies perform similarly well across the board for the real benchmarks. The heuristics-based policy appears to be making intelligent mapping decisions, adequately accounting for both data locality and load balancing. When compared to non-Parla implementations, Parla performs similarly well to other Python versions and is somewhat outperformed by highly-optimized benchmarks from linear algebra libraries. This behavior is expected, as these optimized versions need not deal with Python or tasking overhead. I am optimistic, though, that Parla scales similarly well to these even with its overheads. Of course, these benchmarks represent only a small portion of tasking workflows, and many of them are more compute-intensive than communication-intensive, increasing the importance of simple load balancing over data locality. For further analysis, I now turn to the synthetic benchmarks.

SYNTHETIC BENCHMARKS

Three synthetic benchmarks were created using an in-house synthetic graph generator. The synthetic benchmarks allow a user to specify how much time is spent in each GPU task using a busy wait on the device as well as how much data must be communicated to each device. Benchmarks were run using 16 ms tasks each with 50 MB of data per task, which is a small enough amount of data that its transfer latency can be completely hidden by a single 16 ms task's computation. A good mapping policy will load balance but also avoid unnecessary data copies. In all these tests, the data blocks start evenly distributed across the GPUs in a round-robin manner.

The first synthetic benchmark uses 300 independent tasks. Every 64th launched task shares the same data, so the ideal scheduling policy here is a simple round-robin distributing tasks over GPUs. Figure 7 shows that the User and Automatic policies perform identically, demonstrating that for this simple case the heuristic-based policy does achieve the optimal mapping.

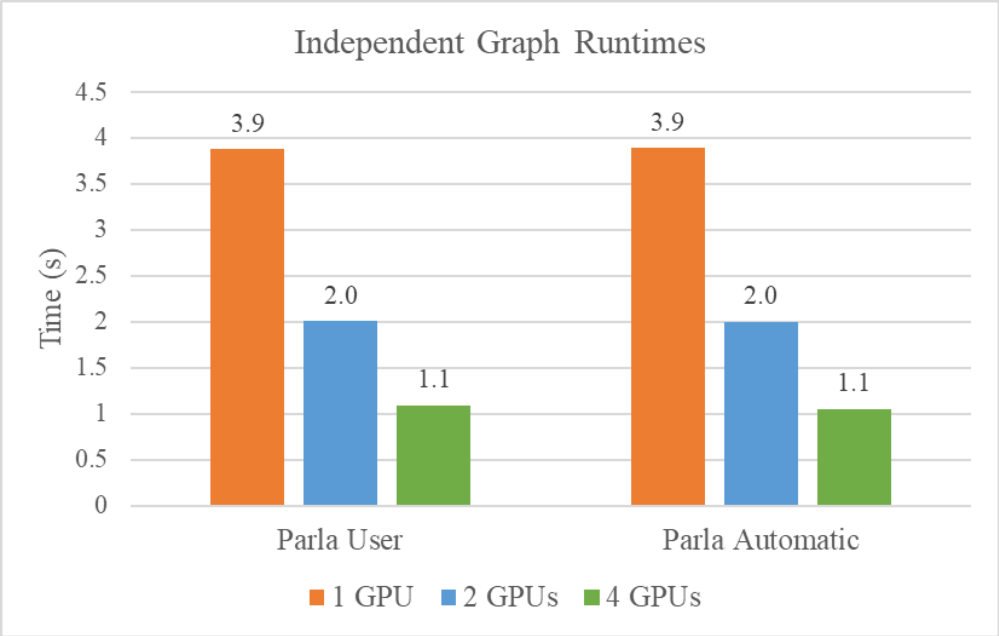

Figure 7. Independent graph runtimes.

The second synthetic benchmark uses 150 tasks in a serial chain where each task simply depends on the previous task launched. All tasks perform read and write operations on the same data. The optimal placement policy here is for all tasks to run on the same device, as parallelism isn't achievable with this graph and using different devices simply causes useless data copies. Figure 8 contains the results for this workload. For the User-based mapping, the results do not change with number of GPUs (as expected given that only one device is actually used). We see *worsening* performance, though, for the Automatic policy. The issue here is that the current mapper is unable to distinguish the fact that nothing can be gained by copying data to other devices. It observes a high load on a single device and tries to map tasks to other devices, causing useless data copies and increasing the overall runtime. Policy improvements are needed to address this issue.

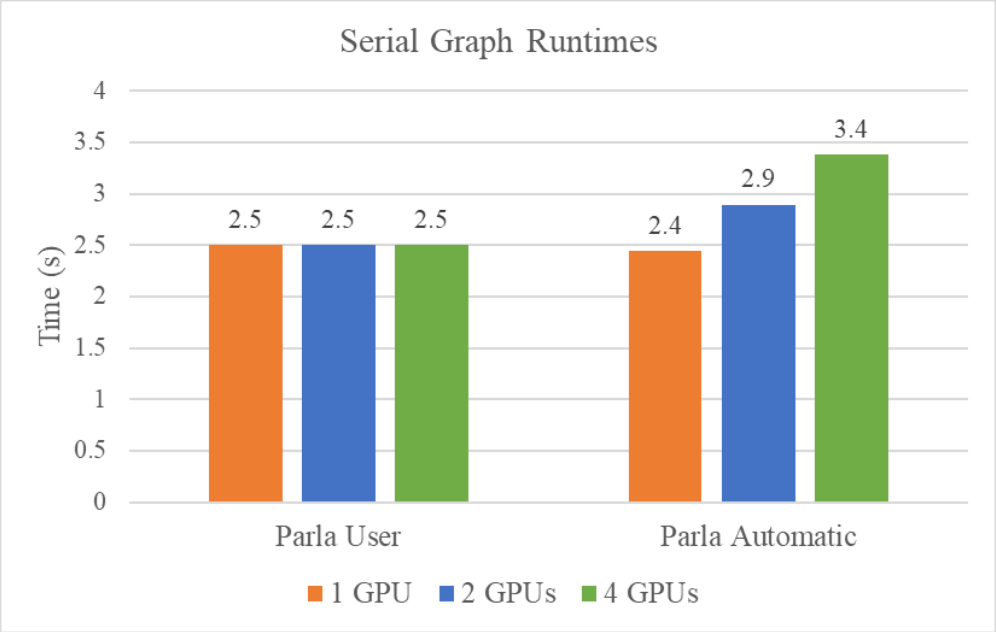

Figure 8. Serial graph runtimes.

The third and final synthetic benchmark is a reduction graph, an inverted binary tree where each task reads data passed to it by its two parents and writes the data of its left parent at output. Optimal placement is to distribute the largest subtrees evenly among the devices, assigning tasks to the same placement as their left parent once the width of the level is less than the number of devices. The reduction tree is an 9-level binary tree. Each level has 2^n tasks where n ranges from 0 to 8. Figure 9 contains the results for this workload. As with the independent workload, for the reduction benchmark the Automatic policy's performance matches that of the User placement.

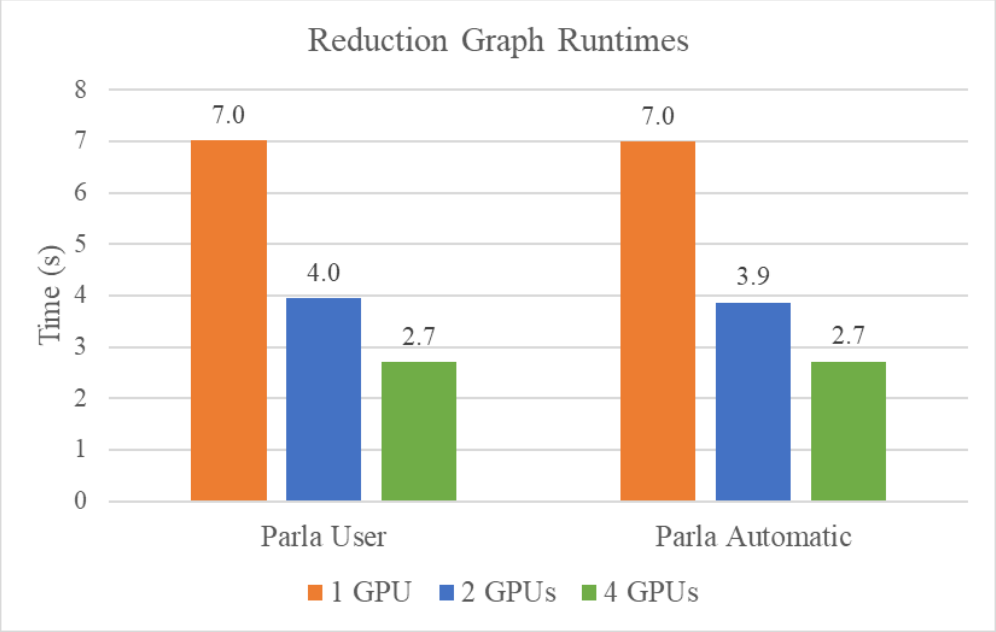

Figure 9. Reduction graph runtimes.

In summary, the synthetic benchmarks give insights into the behavior of the runtime's heuristics-based policy using a few simple graphs with obvious best-case mappings. When simple load-balancing is the only concern, the mapper successfully uses the best schedule; but in the opposite extreme where there are no loads to balance and only data locality should be considered, it fails to choose optimal placements. Improvements to remedy this issue as well as to improve the performance of other areas of the runtime are discussed in the next chapter of this report.

Chapter 5: Future Work

In this section I will outline my ideas for future work. These suggestions are based on the results from Chapter 4, separate profiling results not covered in this report, and my own thoughts on the proper design of the Parla task runtime.

First, while the real benchmarks show that our current mapper policy is promising, the synthetic workloads demonstrate that it is clearly flawed. In particular, it performs poorly on the serial workload. The heuristics need to be far more thorough than they currently are to create a well-rounded, portable mapping algorithm. The data locality calculations in the current codebase are sufficient, but the load balancing calculations are not. The system simply counts the *number* of tasks running on each device; it does not account for the *length* of those tasks. Ideally it should track the runtime of each type of task and use that statistic in its load-balance calculations instead of the very naïve number of tasks. For tasks which have not been run, the amount of memory consumed by the task can be used as an initial first-order estimate for the task's length. Additionally, a more robust check must be in place for whether moving a task to a different device than where its dependency is located is beneficial. The simple boost to a device's suitability can only account for so much, and this implementation clearly breaks down in even a very simple serial task graph, scattering work among devices unnecessarily.

The runtime uses several locks throughout the system to protect various data structures, particularly for resource tracking. Normally, one would use atomics to implement these tracking counters. Python does not natively support atomics, as multi-threaded parallelism is uncommon in the language. The result is that a somewhat-expensive lock must be acquired by any thread wishing to read or write to resource counters, and our system has a great deal of contention on these locks, particularly

slowing down CPU-based workloads and workloads with fine-grained tasks. (The evaluation section of this report only demonstrated coarse-grained GPU workloads). One way to alleviate this issue is to use a third-party atomics library to eliminate locks altogether.

Lock contention could also be reduced through general code inspection and cleaning. Anecdotally, I believe there are far more locks than needed in the code. In fact, a general issue with the codebase is that there are many simple inefficiencies throughout that just need to be patched, such as unnecessary copies to update the spawned tasks queue. Parla, at this point, is as much a software engineering challenge as it is a research challenge, and a fair amount of time needs to be spent addressing issues stemming from rushed backend design. The specifics of these changes are beyond the scope of this report, but I believe some careful cleaning would both increase the usability of the codebase as well as improve performance.

Another issue is that the Parla runtime runs repeatedly in a tight loop instead of only running as needed, and it is implemented in Python which means it acquires the GIL on every iteration. The only thing limiting its GIL usage is a rather arbitrary `sleep(1.4012985e-20)` call. Moreover, this tight loop greatly exacerbates the lock contention problem, as the mapper frequently checks resource usage, particularly if a task cannot map at any given time. The runtime is broken into three phases—map, schedule, and launch—and it needs to be reworked such that each phase only runs as needed. This behavior could be implemented with a callback mechanism where certain actions kick off only the necessary runtime phase. The mapper only needs to be run in two situations: when a new task with no pending dependencies spawned (as it can be mapped), and when any task ends and there are tasks in the mapper queue (as they may have been waiting for resources to free up). The schedule phase only needs to be run whenever a mapped task's

dependencies complete; the task simply needs to be moved to the appropriate device queue. The launcher only needs to be run when a task finishes, as its device is now free, and when any task is enqueued by the scheduler onto a free device.

Another consideration the scheduler needs to account for is memory pressure management. At this time, there is no mechanism for alleviating memory pressure on a device. If a device fills up with unused PArrays, it essentially becomes unusable. There needs to be a mechanism to free up memory on devices to prevent deadlock caused by tasks being unable to map to devices if they all run out of space. A simple solution is to create a new memory pressure relief phase which simply runs whenever any task can't map due to lack of resources. It would survey all PArrays to see which ones are not in-use (meaning being used by any task currently mapped, scheduled, or launched). Unused PArrays could be flushed to the CPU to free space using special memory deallocation tasks—essentially data-movement tasks which not only copy data back to the CPU but also deallocate it on the GPU to free space. The task which failed to map could be set to depend on the deallocation tasks which free up enough space on a given device. If there still isn't enough space, the memory pressure phase would simply run again when some other PArray is no longer in use. This mechanism could get far more complex to improve performance in memory-constrained workloads—there are several policy questions to be answered around when to run such a phase, whether to implement a waterline to keep memory usage below, and which PArrays to select for eviction—but at minimum, the mechanism as described would prevent system deadlock and as such is a necessary Parla feature.

The entire runtime system was initially designed to support multi-device tasks, that is, either tasks which perform both GPU and CPU computations (via NumPy and CuPy calls in one function, for example) or tasks which use multiple GPUs (in a third-

party multi-GPU library call, for example. Unfortunately, in practice the current implementation is limited to only support single-device tasks. Only simple changes would be required in the runtime backend, though, to support multi-device tasks, as the device queues are designed to handle enqueueing a single task into multiple queues for the launcher to check in a group. I hope that multi-device tasks will be implemented, eventually; however, while the runtime would not require many changes to support them, significant design decisions must be made concerning the implementation of the Parla task API and of PArrays.

Lastly, a major challenge for Parla is that its tasks are essentially black boxes. The runtime knows nothing of the nature of the kernels contained in tasks. This, combined with the fact that tasks are dynamically (possibly conditionally) spawned as the program runs, means that the runtime has very little useful information about the task graph as a whole when it makes its mapping and scheduling decisions. On top of those issues, Parla is meant to be portable and as such must account for a variety of architectures and device configurations. It is for these reasons that I chose to implement a greedy, heuristic-based mapper instead of one which takes into account the entire task graph. In addition to improvements to the mapper, the scheduler can be reworked to use a similar heuristic-based scheme and a work-stealing phase could be added to further improve load imbalance. The scheduler could use priority queues instead of simple FIFO queues and could assign each task a priority based on its estimated runtime, the priority of its dependencies, and priority hints from the user. In this way, if a task is scheduled but is later determined to be on the critical path in the task graph (e.g. because it has many dependencies), its priority could be increased and it could be bumped closer to the head of its device queue for faster launching.

The work stealing phase is necessary because no heuristic mapper or scheduler will make perfect decisions with dynamically spawning black-box tasks on every compute node configuration, and binding the runtime's launch phase to those decisions is bound to waste device resources. This phase would run after the launcher runs if any devices are completely idle and any device queues have waiting tasks. A simple, likely effective work-stealing policy would be as follows: Choose a free device to steal work onto, the *thief*. Then, to find a *victim* device (to be robbed of work), identify the device queue with the most tasks (or, alternatively, the most work in terms of total estimated task length). For the highest priority task, calculate the cost of moving its data to the thief device. Calculate the threshold of the task's estimated runtime with its communication time, and if that ratio exceeds a set threshold, steal the work. If the ratio is not exceeded, move to the next task; and if no task is found on the device, move to the next device. To actually steal the work, the task must be popped from the victim's device queue, have data movement tasks created to move its data to the thief, and be enqueued on the thief for launching once it is ready.

To wrap up this section, I acknowledge that these suggestions all have tradeoffs. A callback-based runtime could still run far too frequently in a fine-grained task graph. Multi-device support will necessarily increase the complexity of the runtime. A priority based scheduler sounds nice, but it will require frequent, possibly expensive priority updates to be effective. Similarly, work stealing requires evaluation on a per-task basis of whether it should be stolen or not. I believe these techniques will all improve the performance of Parla, but they do have upper limits. The smarter the backend algorithms become, the more expensive they become. In the off-chance that a future student reads this report and tries to implement these suggestions, keep these tradeoffs in mind.

Chapter 6: Related Work

Dask [7], PyCOMPSs [8], and Ray [9] are all tasking systems designed to integrate quickly with sequential Python. These systems are general purpose but primarily target inter-node orchestration rather than intra-node orchestration as Parla does. In particular, they do not have the same level of GPU integration as Parla. There has been recent work with Dask-CUDA to enable automatic configuration for heterogeneous systems, and PyCOMPSs does have the notion of GPU tasks and separate data-movement tasks, but neither of them directly manage hardware command queues on GPU devices nor use CUDA event-driven synchronization to efficiently schedule GPU tasks. PyCOMPSs is the most similar to Parla, especially in terms of its syntax, flexibility, and gradual adoption features; but the Parla runtime's treatment of intra-node GPU tasks sets it apart from PyCOMPSs's orchestration features.

Legion [10] and Legion-based systems are all similar to Parla in their management of tasks, but all fail to address either graduation adoption or general purpose use. Legion is a data-driven task-based runtime system much like Parla, with some key differences:

- Legion does not support gradual adoption, as applications must be written entirely in Regent, the Legion-based language extension to Terra which itself is an extension of the Lue programming language.
- Everything is a task in Legion—the programmer may not freely choose when to use and not use tasks.
- All data must be explicitly organized into *logical regions*. A task must be annotated with the read/write *privilege* and access *coherence* with which it accesses its regions. This information makes Legion rather restrictive and

difficult to program in, but it does allow the runtime to make very powerful inferences about locality and parallelism, enabling high-performance mapping policies.

- Legion manages both intra-node and inter-node orchestration.

Parla, in contrast, is more intuitive and Python-based, permits the user to freely use tasks when convenient, and only requires the user to use PArrays when it is convenient for programmability and performance. The runtimes are similar in their GPU management features. Moreover, both runtime's default mapper uses a greedy algorithm to choose the best device for distributing work to at runtime. Legion permits the user to completely rewrite the mapper without affecting a program's correctness, while Parla enables the user to provide placement hints to the mapper.

Pygion [11] is a Python interface for Legion which essentially provides the same advantages as Legion but does not support gradual adoption, requiring overhauls to user code to identify logical regions.

Legate NumPy [12] is a drop-in replacement for NumPy built on the Legion runtime. From a user perspective, Legate is not a tasking system; but it is in fact transparently breaking computation into tasks which it uses to build a task graph and launch kernels asynchronously. Legate requires only a single change to user code—a modification of the NumPy import statement to import Legate rather than vanilla NumPy—but it is not truly general purpose. Legate is a re-implementation of traditionally single-node CPU-only NumPy calls, accelerating and distributing them to scale using the Legion runtime. It does not, however, permit the user to create their own custom task kernels.

Ptask [13] is a Windows-based, OS-managed tasking system supporting a dataflow programming model. A user may build a program graph by expressing a series

of tasks, ports identifying the direction of dataflow to and from tasks, and channels connecting those ports. Ptask is implemented at a higher level of abstraction than Parla, supporting communication between processes; but Parla's runtime, particularly its heuristics-based mapper policy, draw many ideas from the Ptask scheduler.

Several other additional tasking systems exist outside of Python as well. StarPU [14] and PaRSEC [15] provide excellent support for heterogeneous hardware on distributed systems. Both emphasize GPU contexts and data prefetching and have heterogeneous data-aware scheduling policies available. Charm++ [16] is a parallel programming framework in C++ which provides support for asynchronously executing CUDA kernels and for orchestrating data movement between hosts and devices. DARMA [17] and HPX [18] are other C++-based frameworks which provide support for expressing deferred, asynchronous tasks. Cilk [19] is a task-based runtime with language support in C and C++ which enables programmers to create a directed acyclic graph (DAG) of tasks and which balances load across devices using a work-stealing algorithm. Open Community Runtime [20] is a task-based runtime which organizes tasks into a DAG and, like Parla, manages event-based task execution. OmpSs [21] and Hydra [22], like PTask, provide graph-based dataflow programming models for offloading tasks across heterogeneous devices.

Chapter 7: Conclusion

Parla faces a unique challenge in heterogeneous computing given its goals of gradual adoption and portable performance. Designing a runtime which can efficiently execute a dynamically-spawned task graph composed of black-box heterogeneous tasks rules out the option of creating a holistic, optimal schedule for executing the graph. Properly managing heterogenous tasks requires special treatment for various architectures. Implementing a tasking system in Python requires careful consideration of GIL contention and multi-threaded communication. Allowing the programmer to specify mapping hints as frequently or infrequently as they choose means that the runtime often makes decisions with very limited information.

In this report, I present my design for a runtime based on a greedy mapping algorithm, a simple scheduler, and a launcher built on worker threads for tackling these challenges. I demonstrate that, while synthetic workloads do reveal some inefficiencies in the design, based on several real workloads it does perform and scale similarly well with its heuristics-based policy as with optimal user-made placements. I identify areas for improvement and outline several options for addressing the current flaws in the runtime.

Parla is still in relatively early stages of development, but the research presented here sets the stage for developments in its runtime to make it competitive with other state-of-the-art heterogeneous tasking systems, all while maintaining the ease of programmability offered by scientific Python.

References

[1] C. R. Harris, K. J. Millman, S. J. van der Walt, R. Gommers, P. Virtanen, D. Cournapeau, E. Wieser, J. Taylor, S. Berg, N. J. Smith et al., "Array programming with numpy," Nature, pp. 357–362, 2020.

[2] Preferred Networks, inc., "CuPy: A NumPy-compatible matrix library accelerated by CUDA," 2020. [Online]. Available: https://cupy.chainer.org/

[3] Anaconda, "Numba: A high-performance Python compiler," 2018. [Online]. Available: https://numba.pydata.org/

[4] N. Al Awar, S. Zhu, G. Biros, and M. Gligoric, "A performance portability framework for python," in Proceedings of the ACM International Conference on Supercomputing, 2021.

[5] Behnel, S. et al., "Cython: The best of both worlds." Computing in Science & Engineering, 13(2), pp. 31-39, 2011.

[6] A. Haidar, A. YarKhan, C. Cao, P. Luszczek, S. Tomov, and J. Dongarra, "Flexible linear algebra development and scheduling with cholesky factorization," in High Performance Computing and Communications, Cyberspace Safety and Security, and International Conference on Embedded Software and Systems, 2015, pp. 861–864.

[7] Dask Development Team, Dask: Library for dynamic task scheduling, 2016.

[8] E. Tejedor, Y. Becerra, G. Alomar, A. Queralt, R. M. Badia, J. Torres, T. Cortes, and J. Labarta, "Pycompss: Parallel computational workflows in python," The International Journal of High Performance Computing Applications, vol. 31, no. 1, pp. 66–82, 2017. [Online]. Available: https://doi.org/10.1177/1094342015594678.

[9] P. Moritz, R. Nishihara, S. Wang, A. Tumanov, R. Liaw, E. Liang, M. Elibol, Z. Yang, W. Paul, M. I. Jordan et al., "Ray: A distributed framework for emerging {AI} applications," in Operating Systems Design and Implementation, 2018, pp. 561–577.

[10] M. Bauer, S. Treichler, E. Slaughter, and A. Aiken, "Legion: Expressing locality and independence with logical regions," in International Conference on High Performance Computing, Networking, Storage and Analysis, 2012, pp. 1–11.

[11] E. Slaughter and A. Aiken, "Pygion: Flexible, scalable task-based parallelism with python," in Parallel Applications Workshop, Alternatives To MPI, 2019, pp. 58–72.

[12] M. Bauer and M. Garland, “Legate numpy: Accelerated and distributed array computing,” in Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC19), 2019.

[13] C. J. Rossbach, J. Currey, M. Silberstein, B. Ray, and E. Witchel, “Ptask: operating system abstractions to manage gpus as compute devices,” in Symposium on Operating Systems Principles, 2011, pp. 233–248.

[14] C. Augonnet, S. Thibault, R. Namyst, and P.-A. Wacrenier, “Starpu: a unified platform for task scheduling on heterogeneous multicore architectures,” Concurrency and Computation: Practice and Experience, pp. 187–198, 2011.

[15] G. Bosilca, A. Bouteiller, A. Danalis, M. Faverge, T. Herault, and J. J. Dongarra, “PaRSEC: Exploiting Heterogeneity to Enhance Scalability,” Computing in Science Engineering, vol. 15, no. 6, pp. 36–45, Nov. 2013.

[16] L. V. Kale and S. Krishnan, “CHARM++: A Portable Concurrent Object Oriented System Based on C++,” in Proceedings of the Eighth Annual Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA ‘93), 1993.

[17] Sandia National Laboratories, DARMA [Source code]. https://github.com/darma-tasking.

[18] H. Kaiser, T. Heller, B. Adelstein-Lelbach, A. Serio, and D. Fey, “HPX: a task based programming model in a global address space,” in Proceedings of the 8th International Conference on Partitioned Global Address Space Programming Models (PGAS ‘14), 2014.

[19] R. D. Blumofe, C. F. Joerg, B. C. Kuszmaul, C. E. Leiserson, K. H. Randall, and Y. Zhou, “Cilk: An efficient multithreaded runtime system,” in Principles and Practice of Parallel Programming, 1995, pp. 207–216.

[20] T. G. Mattson et al., “The Open Community Runtime: A runtime system for extreme scale computing,” 2016 IEEE High Performance Extreme Computing Conference (HPEC), 2016, pp. 1-7, doi: 10.1109/HPEC.2016.7761580.

[21] A. Duran, E. Ayguadé, R. M. Badia, J. Labarta, L. Martinell, X. Martorell, and J. Planas, “Ompss: a proposal for programming heterogeneous multi-core architectures,” Parallel processing letters, pp. 173–193, 2011.

[22] Y. Weinsberg, D. Dolev, T. Anker, M. Ben-Yehuda, and P. Wyckoff, “Tapping into the fountain of cpus: on operating system support for programmable devices,” in Architectural Support for Programming Languages and Operating Systems, 2008, pp. 179–188.