Scaling Parallelism in Interactive Programming

Tom-Olav Bøyum

Thesis submitted for the degree of
Master in Programming and System Architecture
60 credits

Department of Informatics
Faculty of mathematics and natural sciences

UNIVERSITY OF OSLO

Autumn 2020

Scaling Parallelism in Interactive Programming

Tom-Olav Bøyum

© 2020 Tom-Olav Bøyum

Scaling Parallelism in Interactive Programming

http://www.duo.uio.no/

Printed: Reprosentralen, University of Oslo

Abstract

IPython parallel is a python package that's used for distributed and parallel computing. It enables users of IPython and Jupyter Notebooks to interactively run computations in parallel on a local machine or remotely on traditional supercomputers or in the cloud. The way this is done is that the user sets up a cluster of IPython Parallel engines. These are wrappers around the IPython kernel, a process that executes arbitrary python code and reports the results. The engines communicates with the clients through a set of schedulers and executes python coded that the user has requested. The package has already seen wide usage. However, there are some limits to how many engines the IPyton Parallel package can use efficiently. As the need to do more expensive computation that demand ever higher number of processes running in parallel keeps increasing, it was suggested that the package could be improved upon with new schedulers that makes the performance scale better with the number of engines so that more processing power becomes available. This thesis explores the current state of the package and describes the architecture of the system. Some analysis of the performance characteristics that show how the performance scales with the number of engines are presented. Then the details of the new scheduler implementations are explained. Some different benchmarks simulating typical usage are explained and results from these benchmarks are shown. The benchmarks show that for some cases the new scheduler implementations are a significant improvement on the existing ones when it comes to scaling with a higher number of engines. The results from the benchmarks also show some cases where the existing scheduler implementations perform better. Because IPython Parallel is an open source project the work done for this thesis has been submitted to be merged into the main branch of the package. One of the early optimizations has already been submitted and merged into the Jupyter Client. A is a package that is used among others, by IPython Parallel and Jupyter Notebook.

Contents

I Introduction **1**
0.1 What this Thesis is About 3

1 Background **5**
1.1 Project Jupyter 5
1.1.1 IPython 6
1.2 IPython Parallel (IPP) 7
1.2.1 Architecture Overview 7

2 Motivation for the Project **11**
2.1 Scaling Parallelism 12
2.1.1 The Problem With Duplication 12
2.1.2 The Problem With Sequential Code Execution 13
2.1.3 Applying Amdahl's Law to Gain Perfomance Improvements by Parallalelization 14

II The project **15**

3 Planning the project **17**
3.1 Project Goals 17
3.2 Software Tools 17
3.2.1 GitHub 17
3.2.2 py-spy 18
3.2.3 airspeed velocity (asv) 18
3.2.4 Jupyter 18
3.3 Identifying Bottlenecks for Performance 18
3.4 Developing Strategies for Scaling Parallelism 19
3.4.1 BroadcastView 19
3.4.2 Spanning Tree Scheduler 19
3.4.3 BroadcastView + Spanning Tree Scheduler 20

4 Analyzing the Current Performance **21**
4.1 Reproducible Testing Environment Using Google Cloud . . 21
4.2 Benchmarking IPython Parallel 22
4.2.1 Timing DirectView.apply() 22
4.3 Profiling IPython Parallel 24
4.3.1 Optimization of the Message Id Creation 25
4.3.2 Profiling the Task Scheduler 26

5 Investigating New Scheduler and View Implementations **29**
5.1 Implementing a New View 30
5.1.1 BroadcastView 31
5.1.2 New Schedulers 32
5.1.3 Non-Coalescing Scheduler 33
5.1.4 Coalescing Scheduler 34
5.1.5 Spanning Tree Scheduler 35

III Conclusion **37**

6 Results **39**
6.1 Comparing the DirectView vs the BroadcastView 39
6.1.1 Pushing Large Messages 40
6.1.2 Sending and Receiving Large Messages 44
6.1.3 Sending Multiple Empty Messages 48
6.2 Figuring Out the Depth for the Spanning Tree 53
6.3 When to Choose the BroadcastView Instead of the DirectView 55

7 Future Work **57**
7.1 Smart Defaults 57
7.2 Fine Grained Performance Measurements 57
7.3 Benchmarking Real World Examples 58
7.4 Compare Benchmarks With Other Tools 58
7.5 The Partially Coalescing Broadcast Scheduler 58
7.6 Getting the Code Into the Main Repository 59

List of Figures

1.1 Connection diagram showing multiple Jupyter frontends connecting to an IPython kernel 6

1.2 Connection diagram showing multiple Jupyter frontends connecting to multiple IPython kernels with IPython Parallel 6

1.3 ipyparallel architecture overview [3] 7

2.1 Shows a example of what a the message flow between a `Client` and the engines looks like with `DirectView` 12

2.2 Results from a benchmark showing the run time of DirectView.apply() for different number of connected engines for messages per engine per second. 14

4.1 Benchmarked run time of DirectView.apply() for different number of targeted engines for messages per engine per second. Same as Figure: 2.2 22

4.2 Showing measured time for sending N messages per engine for M engines. Here it can be seen that sending 20 messages per engine to two engines takes about ~66µs. For 1024 targeted engines it takes 1300µs to send 20 messsages to each. Missing data points for the two upper lines is because the benchmark timed out. If `DirectlView.apply` perfectly scaled with the number of engines these lines would be overlapping. 23

4.3 Profiling execution of `LoadBalanced.apply` on the client . . 24

4.4 Profiling execution of `LoadBalanced.apply` on the task scheduler 26

5.1 Message flow for the `DirectView`. The `Client` sends 100 messages to the scheduler which is then relayed to 100 engines. The replies from the engines are relayed directly back to the `Client`. 30

5.2 Message flow for the Non Coalescing `BroadcastView`. The `Client` sends 1 messages to the scheduler which is then relayed to 100 engines. The replies from the engines are relayed directly back to the Client. 33

5.3 Message flow for the Coalescing `BroadcastView`. The `Client` sends 1 messages to the scheduler which is then relayed to 100 engines. The replies from the engines are accumulated in the scheduler and relayed as 1 message back to the Client. 34
5.4 Message flow for the `BroadcastView` with the Spanning Tree Scheduler with depth = 2 and 1000 targeted engines. The `Client` sends 1 message to the root scheduler (RS), the root relays the message to two sub schedulers (SS). The sub schedulers then relays the the message to the leaf schedulers (LS). Finally the leaf schedulers relays the message to the engines, sending to 250 engines each. 35

6.1 Benchmark results showing how long it takes to send a ~2MB message to different numbers of engines. The yellow line shows how the `DirectView` scales with the number of engines compared to the `BroadcastView`. 39
6.2 This chart shows the runtime for the push benchmark for different message sizes for 1 targeted engine. The blue bar is the Coalescing `BroadcastView`, the yellow is the `DirectView` and the red is the Non Coalescing `BroadcastView`. 41
6.3 This chart shows the runtime for the push benchmark for different message sizes for 256 targeted engines. 42
6.4 This chart shows the runtime for the push benchmark for different message sizes for 1024 targeted engines. 43
6.5 This chart shows the runtime for the benchmark for different message sizes for 1 targeted engine. The blue bar is the Coalescing `BroadcastView`, the orange bar is the `DirectView` & the red bar is the Non Coalescing `BroadcastView` 45
6.6 This chart shows the runtime for the benchmark for different message sizes for 512 targeted engines. 46
6.7 This chart shows the runtime for the benchmark for different message sizes for 1024 targeted engines. 47
6.8 Benchmark for sending multiple smaller messages to the engines shows that the Non Coalescing `BroadcastView` performs only slightly better than the `DirectView`. However the Coalescing `BroadcastView` clearly performs better when number of targeted engines is ≥ 16 49
6.9 This charts shows the runtime of the benchmark when connected to 256 engines. The x-axis is the number of messages being sent to each engine. The orange bar is the `DirectView`, the red bar is the Non Coalescing `BroadcastView` and the blue bar is the Coalescing `BroadcastView`. Notice how much better the Coalescing `BroadcastView` scales with the number of messages being sent. 51
6.10 This charts shows the messages per engine per second for different numbers of messages sent with the `DirectView`. The lines represent different numbers of targeted engines. Blue line is 1 engine, orange line is 2 engines. 52

6.11 This charts shows the messages per engine per second for different numbers of messages sent with the Coalescing `BroadcastView`. 52
6.12 This charts shows the messages per engine per second for different numbers of messages sent with the Non Coalescing `BroadcastView`. 53
6.13 Benchmarked runtime of Broadcast Scheduler set to Coalescing and with different depth configurations. It shows that if the number of engines is less than 16 the performance gained by parallelism doesn't outweigh the added overhead of sending messages between the schedulers. 53
6.14 Benchmarked runtime of Broadcast Scheduler set to Non Coalescing and with different depth configurations. 54

List of Tables

4.1 Comparing the run time of the original and the new `msg_id()`. Numbers are mean ± std. dev. of 7 runs, 10000 loops each 25
4.2 Comparing the run time of the original and the new `Session.deserialize()`. Numbers are mean ± std. dev. of 7 runs, 10000 loops each 27

6.1 Duration in µs for the push benchmark when targeting 1 engine. 42
6.2 Duration in µs for the push benchmark when targeting 256 engines. 42
6.3 Duration in µs for the push benchmark when targeting 1024 engines. 44
6.4 Duration in µs for the benchmark when targeting one engine. 46
6.5 Duration in µs for the benchmark when targeting 512 engines 47
6.6 Duration in µs for the benchmark when targeting 1024 engines 48
6.7 Number of messages sent per second per engine for the different schedulers. 49
6.8 Measured runtime in µs for the Broadcast Scheduler set to Coalescing. 54

Preface

When I started my masters program I already knew that the topics I wanted to learn more about were parallel computing and distributed software systems. When I saw the project description for this master project it immediately caught my attention. Then when I read through the details and realized it would give me chance to work on IPython, a tool I already knew well and was fond of, and use the code from my master project as a contribution to the open source community, I knew that this project was perfect for me. In the beginning I spent a lot of time setting up the workflow for benchmarking on the Google Compute Engine, tweaking how that worked and analyzing the results. Then I spent a lot of time coding and trying out different ways of optimizing the code. After a while I realized I had spent maybe a little bit too much time coding, which was easy for me to do because its' an activity I much enjoy. I started to really write the thesis at the same time as the pandemic situation happened. This complicated a lot of things for me and became something that made it much harder to focus on the writing. In the end I managed to work around it and put together a thesis that I'm very satisfied with.

I would like to thank my supervisor Benjamin Ragan-Kelley for always being very patient with my questions and for being great at helping me whenever I got stuck on some problem. I would also like to thank the Simula Research Laboratory for providing me with a great space to work, the equipment I needed to be productive and the funds to pay for a good lunch in their great canteens.

Part I

Introduction

0.1 What this Thesis is About

This thesis explores the IPython Parallel package and the problems it has relating to how the performance scales with the number of engines. The main problems are that a lot of duplicate work is being done in the `Client` and that all the work in the scheduler is being done sequentially. When the number of engines increases past a certain limit, the mentioned problems cause the package to be inefficient and hinders the users from getting better performance by using more processors.

The questions this thesis is trying to answer is how the problems of duplication and sequential code execution can be solved in a way that makes it possible to scale parallelism efficiently in IPython Parallel. One of the new scheduler implementations described in this thesis, the Coalescing `BroadcastView` tries to solve the problems of duplication by putting more of the work on the scheduler instead of the `Client`. The new scheduler tries to solve the problem of sequential code execution by using several sub schedulers running in parallel to distribute the work load in between them. The results show that the performance of the Coalescing `BroadcastView` scales much better for some of the benchmarks, but also that the existing scheduler is better for some scenarios.

Chapter 1 (Background) gives an overview of the architecture of IPyton Parallel, how it works and how it relates to Project Jupyter.

Chapter 2 (Motivation for the Project) explains why it's beneficial to scale parallelism. There is a detailed explanation of why the performance doesn't scale well and the problems that this project wants to solve are explained in detail. At the end of the chapter some theory about why parallelism can be used to improve performance is explained.

Chapter 3 (Planning the Project) states the goals for this project and how the goals will be achieved. The most important software tools used in this project are presented. Then the strategies for identifying bottlenecks for performance are given. At the end of the chapter new strategies for scaling parallelism are explained.

Chapter 4 (Analazying the Current Performance) is all about how the performance characteristics of the existing schedulers were analyzed. Then the results from the analysis are shown and explained.

Chapter 5 (Investigating New Scheduler and View Implementations) explains how the strategies for the new schedulers were implemented and exactly how they work.

Chapter 6 (Results) shows and explains the results from benchmarking the new scheduler implementations, when and why the new schedulers are better and when they are not.

Chapter 7 (Future work) brings up some of the questions that arose during the project that could be interesting to explore in a new project.

Chapter 1

Background

To be able to understand the reason for the IPython Parallel project to exist and the value it provides, its important to understand it's place in the IPython and Jupyter software ecosystem and the niche that it fills.

1.1 Project Jupyter

Project Jupyter is a non-profit, open-source project that was born out of the IPython Project in 2014. "Project Juypter exists to develop open-source software, open-standards, and services for interactive computing across dozens of programming languages."[1]. For this project, the key Jupyter tecnhology is the Jupyter messaging protocol. This protocol is based on request/reply pattern, where on one side you have a kernel that executes code, and on the other side you have frontends that connects to the kernel as illustrated in 1.1 . The protocol defines a communication design and message specification for how to develop kernels that communicate with Jupyter frontends. This different in comparison to traditional REPLs (read-eval-print-loop) in that it's based on a two-process communication model. This allows the evaluation part of the REPL to happen in it's own process. The default Jupyter kernel is the IPython Kernel. The IPython kernel provides an alternative interpreter for Python and a ZeroMQ transport layer used to send and receive messages. One such message is the `execute_request`. This message type is used by a frontend to execute code on the kernel. After the code is executed, the kernel returns an `execute_reply` to the client with the results of running the code. This architecture makes it possible to have a kernel running on one machine and multiple clients connecting to it enabling the creation of applications that execute code remotely. One of the most popular Jupyter frontends is the Jupyter Notebook. The Jupyter Notebook is an open-source web application that among other things, allows users to write code in the browser that is executed on the connected kernel that is running locally on a the users machine or remotely.

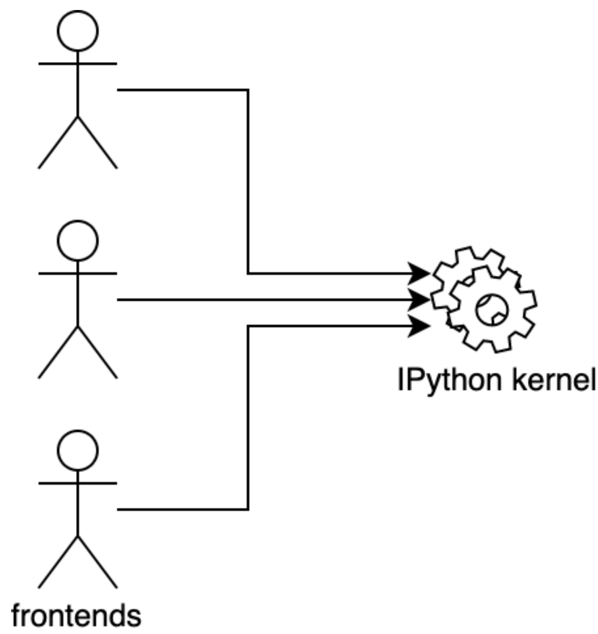

Figure 1.1: Connection diagram showing multiple Jupyter frontends connecting to an IPython kernel

1.1.1 IPython

IPython is a a collection of projects that has a goal of improving interactive and exploratory computing with python. IPython provides an interactive shell for the users to interact with that connects to a kernel that evaluates input from the shell. Using the Jupyter messaging protocol, multiple clients can connect with an IPython shell to the IPython kernel to evaluate code.

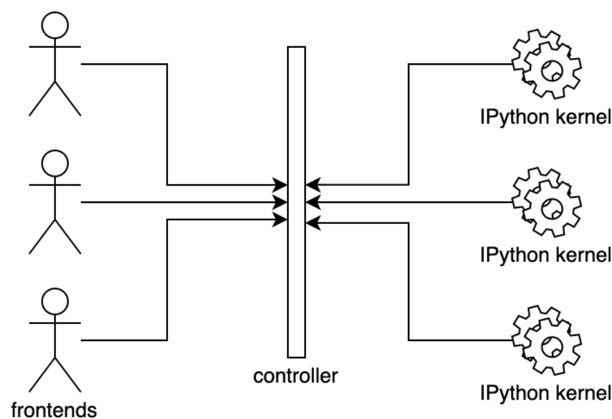

Figure 1.2: Connection diagram showing multiple Jupyter frontends connecting to multiple IPython kernels with IPython Parallel

1.2 IPython Parallel (IPP)

IPython Parallel [2] is a standalone package developed and maintained by the Jupyter team that is used for parallel and distributed computing. IPython Parallel builds on the Jupyter messaging protocol, but reverses the connection direction. Instead of having mutliple clients connecting to one kernel, IPP has multiple kernels connecting to one controller process which clients also connect to. "The architecture abstracts out parallelism in a general way to support many different styles of parallelism"[3]. What this quote means is that IPython Parallel has abstracted out the parallelism in a modular way as classes and services that are part of the architecture. For the users that find the provided implementations of parallel programming patterns insufficient, the architectural choices and the fact that the project is open-source enables the users to extend and change parts of the package to support different styles of parallelism. This can be done by adding new `views` and `schedulers` to the package. IPython Parallel can be used in any python code, but it's especially powerful when used together with the IPython shell because from there users can use the package in an interactive way.

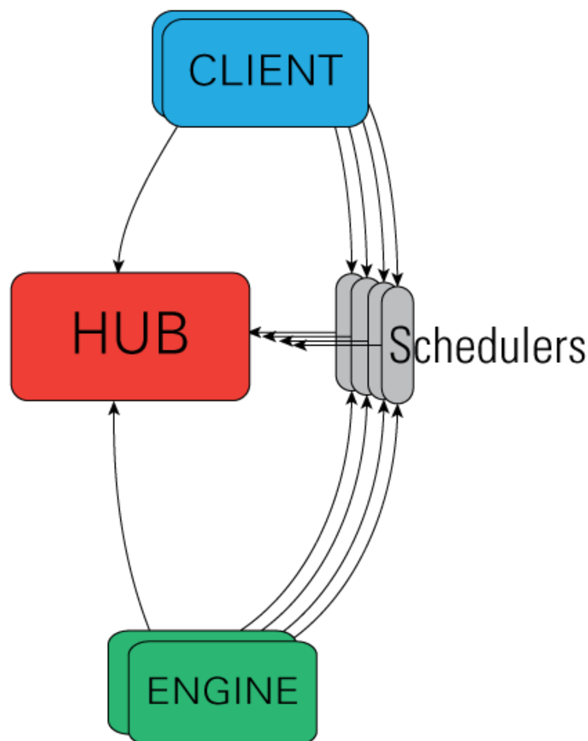

Figure 1.3: ipyparallel architecture overview [3]

1.2.1 Architecture Overview

The IPython Parallel architecture consist of four components:

- The IPython engine

- The IPython hub
- The IPython schedulers
- The IPython client

The IPython Engine

The IPython engine wraps an IPython kernel and adds some functionality that allows it to talk to the controller and schedulers using the ZeroMq protocol defined by the kernel. By modifying the connection direction of the kernel, the engine can connect to the Hub and Schedulers instead of listening for frontend connections. By reversing the connection direction of the IPython kernel implementation, multiple engines can connect to a client instead of having multiple clients connecting to an engine. The job of the engine is to listen to requests over the network, evaluate the code from the requests and returns the results. Each engine has its own namespace where it can store variables independently, this means that code that is run on the engines can return different values if they reference variables that are defined on the engine. Each engine is a separate process, so that when multiple engines are started, parallel and distributed computing becomes possible.

The IPython Controller

"At a general level, the controller is a collection of processes to which the engines and client can connect. The controller is composed of a `hub` and a collection of `schedulers`. The schedulers are typically run in separate processes on the same machine as the hub."[3] The `Client` class is used to connect to the schedulers. The `Client` implements different interfaces that the users can use to access the IPython cluster. At the start of the project, the two primary models for interacting were:

- "A **Direct** interface, where engines are addressed explicitly" [3]
- "A **Load Balanced** interface, where the scheduler is entrusted with assigning work to appropriate engines." [3]

The Direct Interface (DirectView)

The `DirectView` is the simplest interface for interacting with the engine cluster. The `DirectView` communicates with the engines through a multiplexing-scheduler. Together, this scheduler and the view can be understood as an implementation of the SPMD(Single Program, Multiple Data) parallel programming pattern. The `DirectView` is implemented as a class that can be accessed through a method on the `Client`. Using the ids of the connected engines, the users can select all engines or a subset of engines to use as targets to execute code. Its most important method is `apply`. `apply` takes a function as the first argument and optional positional- and keyword arguments to call the provided function with. When calling

the `apply` method, the arguments are serialized and sent in messages, one message per target, to the multiplexing-scheduler which then relays the messages to the targeted engines. When the function is finished, the results are returned from the engines to the scheduler and relayed back to the client. `apply` returns an `AsyncResult` object immediately after the message is sent. By using the non-blocking version of `apply`, the user doesn't have to wait for all the results to come back before doing anything with them, instead the user can iterate through the results from the `AsyncResult` object and act on the partial results as they arrive.

The Task Interface (LoadBalancedView)

The `LoadBalancedView` is the second interface for users to interact with the engine cluster. The `LoadBalancedView` together with the task scheduler implements the task-farming style of parallelism. Contrary to the `DirectView`, the `LoadBalancedView` doesn't let the users select which engines to target. Instead, the `LoadBalancedView` leaves that responsibility to the task scheduler. The view presents the same functions to the users as the `DirectView`, but the implementation of these functions differ slightly. One of the differences is how the `apply` sends messages to the connected scheduler, instead of sending one message per target for each call to `apply`, the `LoadBalancedView` only sends one message to the scheduler for each call, adding a list of available engine ids to the metadata field of the message. In addition, this view allows the users to set functional dependencies and graph dependencies for the tasks. That makes it possible to control the order in which the tasks are executed and/or setting certain conditions that must be satisfied for the task to run.

The Schedulers

All messages that the are sent from the `Client` to the engine go through a scheduler. A scheduler in IPP can be defined as a service to which the engines and clients connect, that implements a messaging pattern between them. "While the engines themselves block when user code is run, the schedulers hide that from the user to provide a fully asynchronous interface to a set of engines"[3]. At the start of this project, there were two different scheduler implementations available.

The Multiplexing-Scheduler (mux)

The mux-scheduler is a scheduler used for explicit addressing. The mux-scheduler is implemented purely in C using the libzmq library. The scheduler does direct routing from the client to individual engines. Built on ZeroMQ routing id, it enables sending messages to specific destinations using an identity prefix contained in the headers of the messages. The way the mux-scheduler works is that the `DirectView` creates one message per target engine that the user has selected to execute a task on. The messages are sent from the client to the scheduler and the scheduler relays

the messages to the engines. The engines replies to the scheduler and the scheduler returns the messages to the client. Because the scheduler is implemented in C, it avoids run-time overhead from python code, that makes it very fast for a lot of common use cases. Another advantage of the scheduler is that it's continuously returning replies from the engines as soon as they come in. That means that the user can get partial results of parallel tasks without having to wait for all the engines to finish. The mux-scheduler is a light-weight relay for the `DirectView`, when used with the `DirectView` the bottleneck is the view itself. Because the implementation is so simple, it makes it hard to improve the bottleneck in the `DirectView` without also changing the scheduler for another one.

The Task Scheduler

The task scheduler implementation selects the target engines for the users. This scheduler monitors the work load of the engines and evenly distributes the incoming work so that no engine is overloaded or idling when there is work to do. The scheduler also keeps track of which tasks succeeds, so that if a tasks fails it can try to run the task again. Because the scheduler monitors the status of the tasks and evenly distributes the work load, this implementation is more fault tolerant and load balanced than the multiplexing-scheduler. Like the mux-scheduler, because the scheduler is running in one process only, it suffers from some of the same performance limitations when running on a high number of engines. Additionally the scheduler is implemented in Python, adding some performance overhead compared to the mux-scheduler.

The IPython Hub

"The hub is the process that keeps track of engine connections, schedulers, client as well as all task requests and results. The primary role of the hub is to facilitate queries of the cluster state, and minimize the necessary information required to establish the many connections involved in connecting new clients and engines."[3]

Chapter 2

Motivation for the Project

The amount of digital information generated in the world keeps increasing faster and faster every year. Similarly, the demand for more insight and more knowledge gained from this data keeps increasing, demanding more and more computational power to handle today's challenges. For a while the demand for better computing could be met with the development of faster processors. However, we have now passed the point where the laws of physics makes it unfeasible to significantly increase the performance of single core processors. To overcome this setback, engineers are developing ways of making processors with more and more cores and are creating huge computing clusters that are able to connect very large amounts of processors. To be able to fully take advantage of these computer architectures and keep evolving performance in computation, we have to develop smarter algorithms for highly parallel computing that are easy to use and available for the people who needs it. IPython Parallel aims to make it easy to set up a cluster of IPython kernels, distributed or local, and access them through different interfaces. This makes it possible to do highly parallelized computing in an interactive way. The package has already seen a lot of use, and has shown that it works well for many different use cases. However, it has some scaling limits that could possibly be better by improving the architecture.

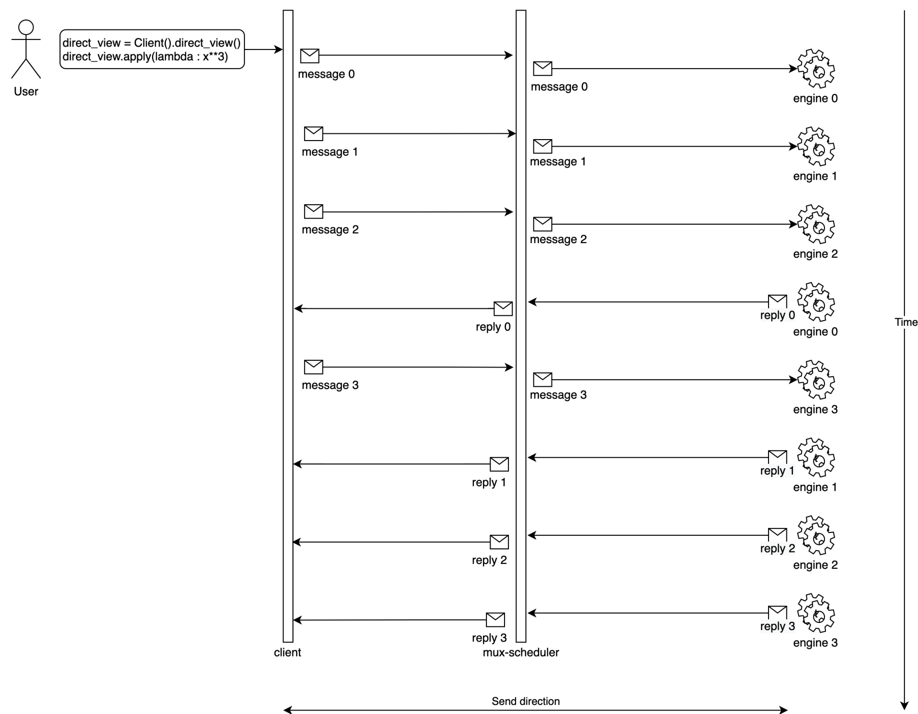

Figure 2.1: Shows a example of what a the message flow between a `Client` and the engines looks like with `DirectView`

2.1 Scaling Parallelism

The existing `DirectView` interface and mux-scheduler is very fast when the number of engines stays low, but due to some limitations with the implementation it doesn't scale well for large numbers of engines. There are two main reasons why it doesn't scale well. The first is that there is a lot of duplicate work being done by the client. The second reason is that the scheduler itself relays messages sequentially.

2.1.1 The Problem With Duplication

The `DirectView` does a lot of duplicated work. For each call to `apply`, it has to create 1 message per targeted engine. When the messages are created, the provided functions and arguments has to be converted to byte strings along with the metadata and header for the message. The only difference between these messages are the targeted engine id in the headers. This means that the `DirectView` is doing the same work over and over again and could be optimized. After the messages are created, they are sent one by one to the multiplexing-scheduler. Again the content of the messages being sent is the same, the only difference is the targeted engine `id` in the header of the message . Sending messages over the network adds a lot of overhead to the run time of the program, especially when the scheduler and the client sending the message are not necessarily on the same machine or

network. When the engines reply, the scheduler has to relay the messages back to the client, where each reply message is then deserialized, adding additional expensive operations. This is not a problem when connecting to small amounts of engines as the number of messages is small, but for the users that want to run tasks on high numbers of engines they will see that it doesn't scale well as a lot of time is being spent in the client just getting the tasks to and from the engines that execute them. The run time of the `DirectView` scales linearly with the number of engines, the fact that most of the work is duplicated suggests that there exists an alternative that scales better. By implementing a new scheduler, it could be possible to change the number of messages sends and serialize/deserialize operations between the client and scheduler from $2n$ sends and serialize for n engines, to just 2 messages and serialize/deserialize operations for each call to `apply`.

2.1.2 The Problem With Sequential Code Execution

Even if the number of messages sent to the multiplexing-scheduler is drastically reduced, the scheduler itself has to send 1 message per engine to the engines. Since sending messages is slow and everything inside the scheduler happens sequentially, this scales linearly with the number of engines. Because the messages are sent one by one from the client through the scheduler to the engines, the last targeted engine in the list of targets can't start before all the other engines have received their messages as well. Furthermore, the scheduler has to listen for messages as well, creating interruptions on the sends and adding additional delay. This is illustrated in Figure 2.1 where you can see the scheduler is receiving and relaying reply 0 from the engine to the client before being finished with relaying the other messages to the engines. In the example on the figure there are only 4 engines so it wouldn't be a problem, but imagine if there were 1000 engines and it's clear that it doesn't scale. IPython Parallel is a project that aims to speedup computation by doing the computation on multiple engines running in parallel instead of just one engine sequentially. Theoretically, 2 engines should be able to run a task twice as fast as 1 engine. 4 engines should be able to run 4 times as fast as 1 engine etc. In practice, the speedup is never that good because usually there will be parts of the tasks that has to run sequentially as described by Amdahl's law [13]. However, we can get a significant speedup by running code in parallel if the code uses suitable algorithms. Using that line of thinking, it should be possible to improve the performance of sending messages by using multiple schedulers instead of just one and distribute the work between them. Using multiple schedulers would make it possible to send messages to multiple engines concurrently, and the work required by each scheduler would significantly decrease and similarly decreasing the delay caused by interruptions.

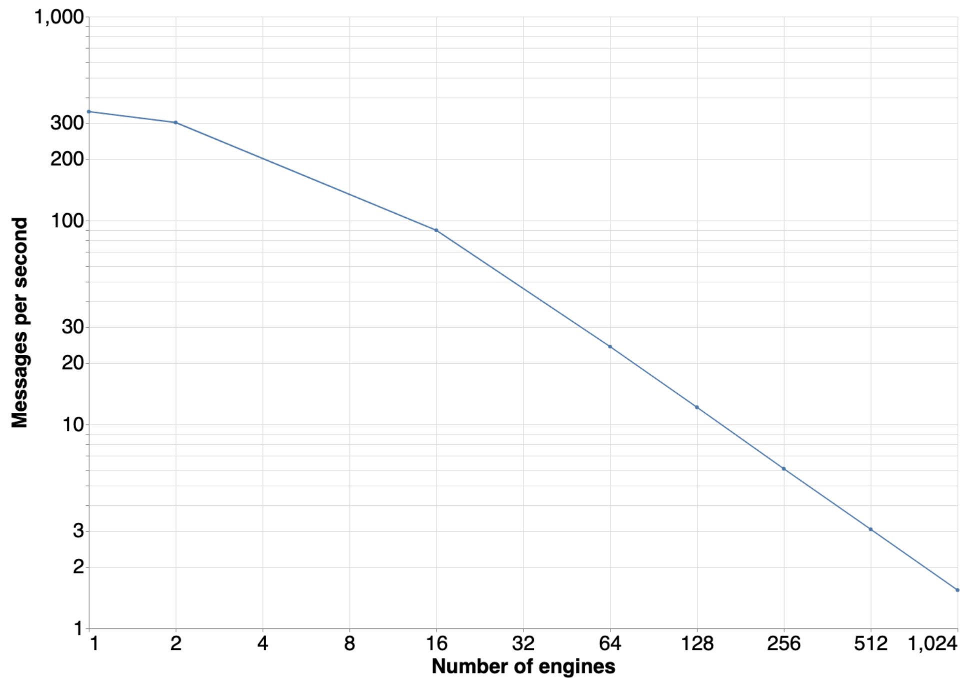

Figure 2.2: Results from a benchmark showing the run time of DirectView.apply() for different number of connected engines for messages per engine per second.

2.1.3 Applying Amdahl's Law to Gain Perfomance Improvements by Parallalelization

Amdahl's law states that the possible performance improvement gained by adding more processors is limited to the run time of the parts of the program that can't be parallelized. Amdahl's equation for speedup by parallelism can be formulated as $Speedup_{max}(N) = \frac{1}{(1-P)+\frac{P}{N}}$ where N is the number of processors and P is the percentage of the total run time of the program that can be parallelized. As N grows the actual speedup becomes $Speedup_{max}(N) = \frac{1}{1-P}$. What can be learned from this equation is that to gain performance improvements by parallelism, P should be as large as possible. Data serialization and sends in the client is not a good candidate for parallelization, however by getting rid of the duplication in the client the scale factor of these operations can be reduced from N for N engines to just 1 for N engines. What remains then is the time it takes for the scheduler to sequentially relay messages between the client and the engines. The scheduler is a good candidate for parallelization, so making a new scheduler that can relay multiple messages simultaneously will increase P significantly.

Part II

The project

Chapter 3

Planning the project

3.1 Project Goals

The goal of this project is to scale parallelism in IPython Parallel with regards to performance. This will be done by using tools to analyze the run time performance of the program to identify the paint points of the program. Strategies to optimize these paint points will be explored. The focus will be to optimize current bottlenecks, develop and implement new scheduler algorithms in order to explore ways of scaling parallelism to enable performance gains from using higher numbers of processes running in parallel. If successful these new optimizations and sheduler algorithms will be merged into the IPython Parallel code base. This will enable users of IPython and Jupyter to run more computationally expensive tasks because the increase in the number of available engines to use for parallel computing will make larger problems faster to solve.

3.2 Software Tools

To be able to answer the research questions, measure progress and organize the code developed, some smart software tools were needed. One of the big advantages to working with python as a programming language is it's open source nature. This has fostered a large community of developers who create and maintain very high quality software tools that are useful for research projects and software development. That made it easy to find and select the necessary software tools to work with the project.

3.2.1 GitHub

Since this project is focused on developing code, one of the first things that had to be established was a way to access and organize the codebase and the new additions. The main branch of the IPython Parallel project lives on the IPython GitHub repository, so to be able to work on the code independently from the main branch, a fork of IPython Parallel [4] was created. Using GitHub to host the forked repository was the natural choice,

as the graphical interface makes it very easy to get an overview of the commits and to compare it with the original repository.

3.2.2 py-spy

To be able to identify targets for optimization, a tool for profiling the run time of the different parts of the program was needed. After some consideration of different alternatives, py-spy [5] was selected. py-spy is a sampling profiler for python programs. It was selected for its ease of use, it's extremely low overhead and because it has a very nice output in the form of interactive .svg files. py-spy makes it possible to attach a profiler to a running python process, i.e. a scheduler, and measure in detail the run time of the different parts of the program.

3.2.3 airspeed velocity (asv)

While py-spy is very useful for deeply analyzing the performance of python programs, another approach was needed to do a more high-level analysis of performance to make it possible to compare different parts and iterations of the program code. To this end, airspeed velocity [6] was selected as the best candidate. Asv makes it possible to run benchmarks and collect timing measurements of the benchmarks in an easy way. This can be used to measure the performance of new schedulers under different parameters and contexts. The collected measurements makes it possible to compare different versions of the code and different schedulers in IPython Parallel.

3.2.4 Jupyter

To be able to understand the results from benchmark- and profiling results in a useful way and to document the project as it progresses, a good tool for interpreting and visualizing data is needed. Jupyter [1] notebooks were the natural choice for that because it has some great options for managing and visualizing data that are tightly integrated with the notebooks. Pandas [7] used together with altair [8] in Jupyter notebooks is a winning combination for visualizing the type of tabular data produced by our parameterized benchmarks. These libraries makes it very easy to manage the results from benchmark runs and produces some excellent interactive visualizations that are shown directly in the notebook.

3.3 Identifying Bottlenecks for Performance

Once the necessary tools were decided on, the first step was to identify some low hanging fruit that could be easily optimized in order to get familiar with the codebase. Tests were created that simulates typical usage scenarios for the `DirectView` with the `mux-sceduler`. Running those tests while profiling the scheduler with py-spy revealed that a lot of time was unnecessarily being spent parsing date/time metadata in serialize and

deserialize functions and too much time was being spent creating unique message ids by the Jupyter client. Identifying and solving these proved to be a great starting point for working with the code, as it was a good target for optimization and a good opportunity to get familiar with the workflow of contributing to the IPython and Jupyter repositories. The improvements were quickly accepted upstream, and now all the Jupyter users are benefit from them.

3.4 Developing Strategies for Scaling Parallelism

Initially it was hypothesized that one of the reason that the performance of the `DirectView` doesn't scale well with the number of engines is because of the way the `DirectView` does sends sequentially. The main reason being that to apply 1 task, the `DirectView` has to send N messages to the scheduler for N engines, and then the `mux-scheduler` has to relay these messages 1 at the time to all the targeted engines. As explained in the the paragraph 2.1 on page 12, while the scheduler is sending messages to the engines, it is also receiving replies from the engines that it has to return to the client. When N is high, (> 100), this can cause a lot of disruptions for the scheduler. Keeping this in mind, some strategies for new scheduler implementations were devised.

3.4.1 BroadcastView

In order to reduce the number of messages sent between the client and the scheduler from one message per engine per task, to just one message per task, a broadcast scheduler was suggested. The idea was to create a scheduler that receives only one message per task from the client, then forwards the message to each engine defined as a target in the metadata of the message. The hypothesis was that this will decrease the time spent in the client for sending messages to the scheduler, and possibly also reduce the number of times that the scheduler is interrupted while sending messages to the engines by limiting the number of messages received from the client. Two different implementations of this was suggested:

- One that immediately forwards replies from the engines to the client, hereby referred to as the Non Coalescing scheduler
- One that waits all the engines to reply, accumulating the partial results, then returns the merged partial results as one reply to the engine. This approach would also reduce the number of replies returned to the client from one per engine to just one per task. This strategy is hereafter referred to as the Coalescing scheduler.

3.4.2 Spanning Tree Scheduler

To really scale the number of engines a task can run concurrently on, making a parallelised scheduler was suggested as a possible strategy.

Instead of having only one scheduler running in one process, it could be possible to have multiple levels of sub-schedulers running concurrently in separate processes. Splitting the work of the scheduler into several smaller units and distributing them to sub-schedulers create less work to do for each scheduler and allows for speed up of the scheduler work by parallelism.

3.4.3 BroadcastView + Spanning Tree Scheduler

By using the `BroadCastView` with the `Spanning Tree Scheduler` it could be possible to get a speed up from parallelism but with the added cost of some extra latency. Comparing how long the last engine needs to wait to receive it's message it becomes clear that the`Spanning Tree` has some advantages. Using the `DirectView` the last engine has to wait for N serialize and send at the client and N relays at the scheduler for N engines. With the `BroadcastView` there is only 1 serialize and send at the client. Then it takes $2^{D+1} - 2$ sends to get to the leaf + $\frac{N}{2^D}$ sends at the leaf when N is the number of engines and D is the depth of the `Spanning Tree`.

Chapter 4

Analyzing the Current Performance

To really understand the pain points of IPython Parallel and how it could be optimized, some deep analysis of the performance characteristics of the program were required. In order to simulate typical usage patterns, some benchmarks were created using the airspeed velocity(asv)[6] package. These were ran and measured to get an understanding of the run time perfomance and limitations under different constraints. After getting an overview, py-spy[5] was used to profile the program under similar contexts to figure out which specific parts of the code were creating bottlenecks.

4.1 Reproducible Testing Environment Using Google Cloud

In order to really test the performance of IPython Parallel and to measure performance gains attained from parallelism, an environment with a high number of available processors were needed to be able to run high numbers of engines in parallel. The Abel computer cluster at the University of Oslo was considered, but ultimately Google Cloud Compute Engine(GCE)[9] was chosen as it was more available and simpler to use. Using GCE it was easy to set up an instance template that made it possible to spin up a computing node running linux with a high number of processors available to use for running benchmarks and then shutting it off again when the benchmarks were finished. In addition to being able to run IPython Parallel with a high number of processors available, using GCE had the added benefit of having an environment that made it possible to get accurately reproducible results so that run time statistics from benchmark runs taken on different days could be compared with confidence. As soon as the GCE instance configuration was set up, running the benchmarks and collecting the results became easy as all it took was to run a script that started an instance on GCE, installed the right python packages, pulled the relevant repositories from GitHub, ran the benchmarks and uploaded the results afterwards.

4.2 Benchmarking IPython Parallel

Because the benchmarks were starting a high number of processes and were generating and sending large amounts of data, a new instance on GCE was instantiated for each benchmark to avoid hitting process and memory limits of the OS. When running benchmarks in asv a new python environment will be created each time asv runs. Every benchmark is repeated 10 times, then the mean of the run times of all the repetitions is taken as the result for the benchmark run.

4.2.1 Timing DirectView.apply()

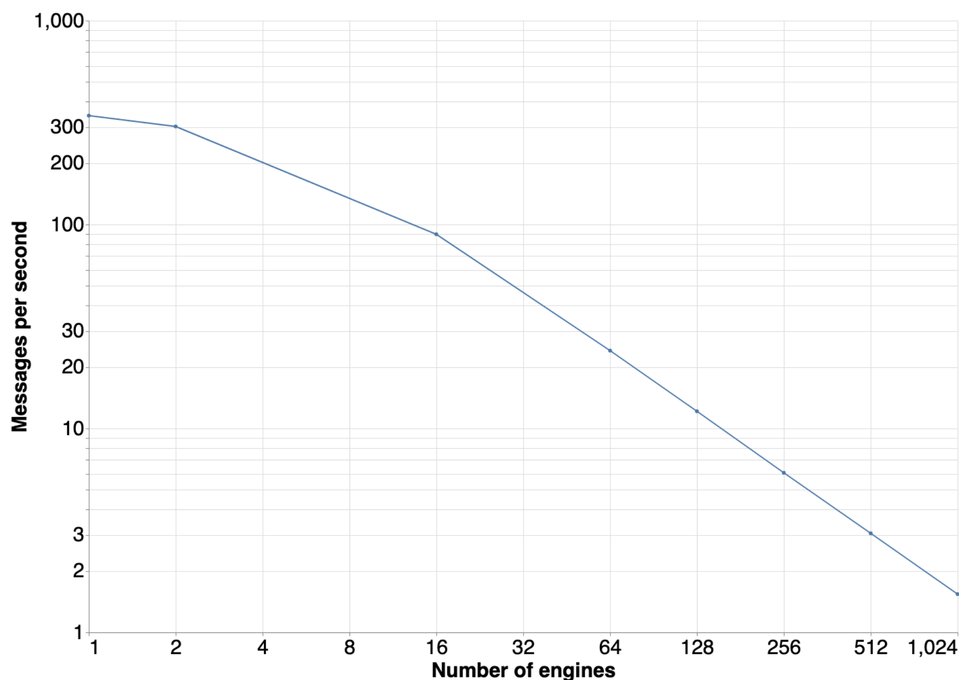

Figure 4.1: Benchmarked run time of DirectView.apply() for different number of targeted engines for messages per engine per second. Same as Figure: 2.2

To get an understanding of how the performance scales with increasing number of engines, a benchmark was created to measure the elapsed time it takes for the `DirectView.apply` method to send a message to each targeted engine. This benchmark runs a loop calling `DirectView.apply` N times for M engines, collecting the asynchronous results and then after finishing all the sends, the benchmark waits until all the targeted engines have replies. The message contains the echo function:

```
def echo(x):
        return x
```

and a 1kB NumPy[10] array as an argument for the echo function. The message is sent to the engines, where each engine calls the echo function

from the message with the array as an argument and returns the result of applying the function on the array in a new message. What can be learned from this benchmark is how long it takes to send 1kB of data to all the targeted engines and back again to the client when using the `DirectView`. The results from this benchmark is shown in Figure: 4.1. The data for this figure is based on how long it took to send 20 messages to each of the targeted engines, and then using that to calculate how many messages are sent per engine per second. Here it becomes apparent just how badly the performance of `DirectView.apply` scales with the number of engines. It starts out good, with just one engine as the target the `DirectView` is able to send more than 300 messages per second. With 1024 engines connected however, the results are unsatisfactory as it is shown that the `DirectView` is only able to send less than 2 messages per second to each of the targeted engines.

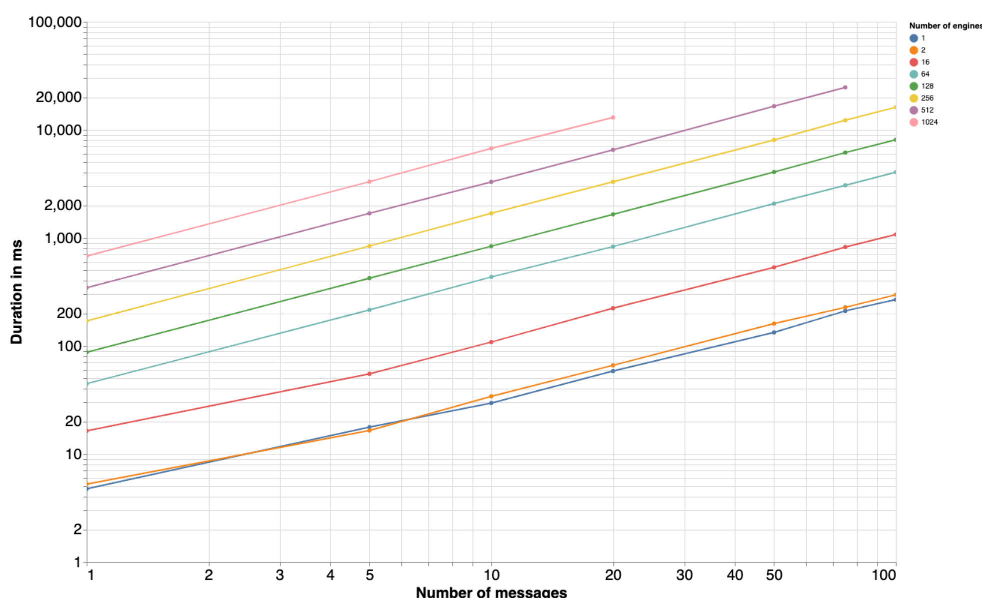

Figure 4.2: Showing measured time for sending N messages per engine for M engines. Here it can be seen that sending 20 messages per engine to two engines takes about ~66µs. For 1024 targeted engines it takes 1300µs to send 20 messsages to each. Missing data points for the two upper lines is because the benchmark timed out. If `Direct1View.apply` perfectly scaled with the number of engines these lines would be overlapping.

The results can be seen in more detail in Figure: 4.2. In the figure each line represent a different number of targeted engines used as a parameter for the benchmark, the x-axis is the number of messages sent to each engine and the y-axis is the duration it took to get a reply from all the targeted engines. What can be seen in this figure is that the `DirectView` is quick when only connected to a small number of engines, it takes 297µs to send 100 messages when targeting only two engines. The yellow line however shows how much worse the performance is when connected to 256 engines.

It shows that sending 100 messages to 256 engines and receiving the replies takes ~16.2 seconds. With 512 targeted engines it takes ~25 seconds for only 75 messages, with 100 messages the benchmark stops working and times out after 180 seconds. With 1024 targeted engines the benchmark stops working and times out already for 50 messages.

4.3 Profiling IPython Parallel

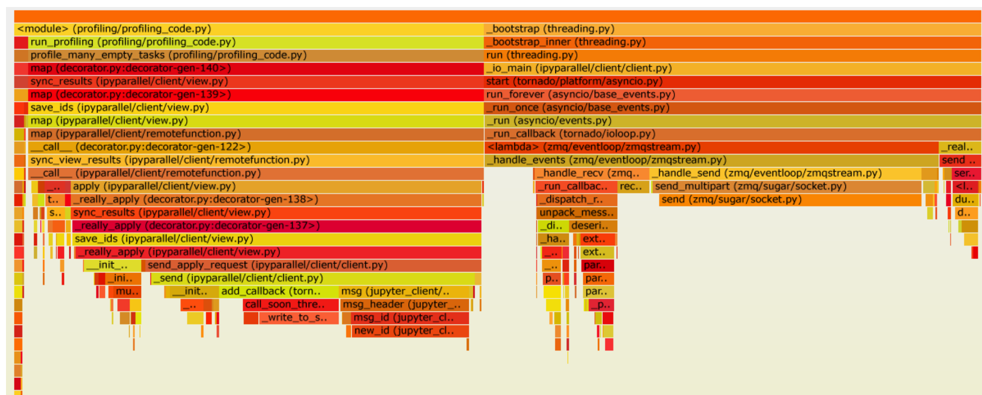

Figure 4.3: Profiling execution of `LoadBalanced.apply` on the client

After having some insight into how the performance characteristics of the `DirectView` looked like with different numbers of engines, it was decided that a more detailed analysis was needed. A script simulating the same usage patterns as the benchmark explained in 4.2.1 were created. Using the py-spy profiler to attach to the client process and the scheduler process the run time of the different functions of the code could be measured. For profiling, the mux-scheduler did not prove very interesting. Because the mux-scheduler uses a compiled function provided by pyzmq, the profiler wouldn't be able to measure what is actually happening inside that function. The other reason why the mux-scheduler was not so interesting to profile is because a new scheduler would most likely have to be implemented in python to have more control over the message relaying. So for this analysis the `LoadBalancedView` and the task scheduler were considered more interesting as the task scheduler is implemented in python and more likely to be similar to a new scheduler implementation. The results from profiling the client can be seen in Figure: 4.3. Each bar is a function call and the width of the bar represents how much time was spent in total during profiling from the function was called until it returned. As was expected, the charts show that the function where the most time was spent during execution were the functions that handle sending and receiving messages. What was surprising however is that the profiling showed that a significant amount of time was spent by the client creating new messages ids.

4.3.1 Optimization of the Message Id Creation

With the profiling results as the motivating factor, a deep dig into the code base revealed what was causing such a large amount of time being spent creating message ids. Each instance of the `Client` class gets an instance of the `Session` class from Jupyter Client[11]. This `Session` instance is used by the views to create and send messages. Analyzing the `Session` code it turned out that the reason why so much time was spent there was because the `msg_id()` method was creating an id string consisting of 16 random bytes as hex-encoded text for every new message created. These ids are guaranteed to be random, but they come with a cost because they are expensive to compute. The `Session` instance on the `Client` already has a unique id, adding a total messages counter to the session and appending the message number to the session id and using that as the message id saved some time from having to compute the unnecessarily robust randomness of the original implementation.

Function	**Measured runtime**
`msg_id()` before patch	5.06 µs ± 376 ns
`msg_id()` after patch	1.26 µs ± 149 ns
`Session.serialize()` before patch	53.2 µs ± 3.54 µs
`Session.serialize()` after patch	42.3 µs ± 3.48 µs

Table 4.1: Comparing the run time of the original and the new `msg_id()`. Numbers are mean ± std. dev. of 7 runs, 10000 loops each

Table 4.1 shows that even though the change was small, the speed up gained was significant. The `msg_id()` numbers comes from timing the runtime of the original `msg_id()` and the new one with timeit[12]. The `Session.serialize()` numbers comes from measuring how long it takes to create a new message and serialize it using the original and the optimized version of `Session.serialize` with timeit. The optimized functions were included in a pull request to the Jupyter Client repository and already millions of users are benefiting from this change as the `Session` class is used in a lot of different context.

4.3.2 Profiling the Task Scheduler

Figure 4.4: Profiling execution of `LoadBalanced.apply` on the task scheduler

To get some ideas about how to implement a new scheduler, the Task Scheduler was also profiled with py-spy. The results are shown in Figure: 4.4. As can be seen on the wide bars on the bottom of the figure most of the time spent in the scheduler is spent parsing dates. That was a surprising and interesting result as there isn't really any good reason why the scheduler would be doing that. Digging deep into the code it was discovered that as part of deseralizing the header of received messages, the scheduler was spending a lot of time trying to parse the date fields of the header. Again the Jupyter Client `Session` class was the reason for why the performance was worse than expected. Specifically the `extract_dates()` function used in Session.deserialize() was slowing down the scheduler significantly. The reason why the scheduler needs to deserialize the message header is so that it can retrieve the message id so that it knows where to relay the message, but the date fields in the header is not actually needed for the scheduler. To optimize the performance of the task scheduler a plan for rewriting the actual `extract_dates()` or `Session.deserialize()` functions was considered. This optimization could not be merged into the Jupyter Client repository because it would be a backward-incompatible change in a widely used library. The solution was then to only apply the opimization when `Session.deserialize` is used in the IPython Parallel schedulers. The functionality of `extract_dates()` was replace with the `echo` function that just returns the date unparsed dates and some extra checks were added to the places in the code base that actually needed to know about dates, parsing them only if necessary. This reduced the time spent deserializing message headers to a negligible amount and was a good way to get really familiar with how the task-scheduler implementation works.

Function	**Measured runtime**
`Session.deserialize()` before patch	174 µs ± 22.4 µs
`Session.deserialize()` after patch	32.2 µs ± 3.6 µs

Table 4.2: Comparing the run time of the original and the new `Session.deserialize()`. Numbers are mean ± std. dev. of 7 runs, 10000 loops each

Chapter 5

Investigating New Scheduler and View Implementations

After having gotten familiar with the current view and scheduler implementations and having analyzed their performance characteristics to figure out their pain points, 3 different strategies for dealing with the problems of duplication and sequential code execution were planned out. To solve the problem of duplication, a `BroadcastView` that only sends 1 message to the scheduler per call to `BroadcastView.apply()` was suggested as a possible solution. Using this view, two different scheduler implementations were suggested:

1. A Coalescing scheduler that accumulates replies from the engines and sends one accumulated reply to the `Client`

2. A Non Coalescing scheduler that relays the replies to the `Client` as soon as they arrive at the scheduler

To overcome the problem of sequential code execution in the scheduler a third scheduler implementation was suggested, a Spanning Tree Scheduler where multiple schedulers run in parallel and distribute the workload among themselves. This chapter will explain in detail how the new view and the suggested new schedulers were implemented and their advantages and disadvantages.

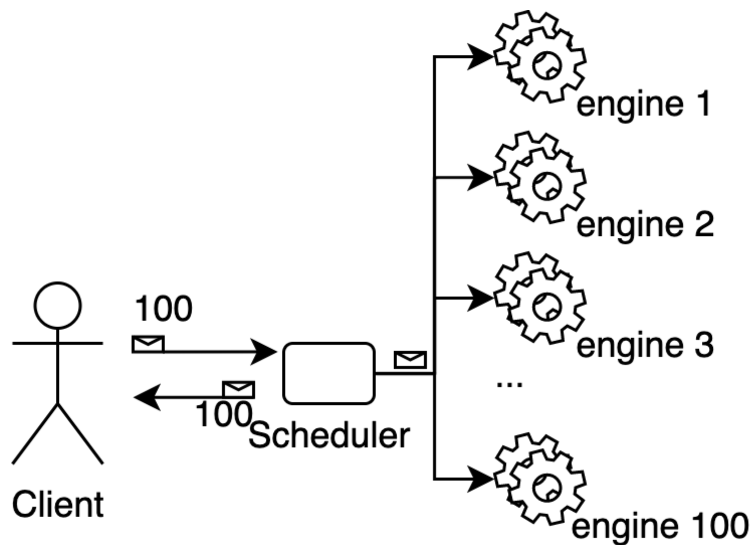

Figure 5.1: Message flow for the `DirectView`. The `Client` sends 100 messages to the scheduler which is then relayed to 100 engines. The replies from the engines are relayed directly back to the `Client`.

5.1 Implementing a New View

For the implementation of the new `BroadcastView`, the `DirectView` implementation was used as a starting point. The way the `DirectView.apply` works is as follows: `apply` takes a function `f` and optional arguments and keyword arguments to call `f` with. `apply` serializes the function, the optional arguments and the keyword arguments. Then a list of the targeted engine ids for the messages is created and for every targeted engine id a new message is created by packing the provided function `f` and its arguments. Some metadata is appended to the message and the targeted engine id is put in the header. The `Client.send_apply_request()` method is called to actually send the message. This method returns a `future` object containing the message id and is stored in a list. Once a message is sent for every targeted engine, the `DirectView` creates an `AsyncResult` object with the list of `futures` and waits, either blocking or non-blocking depending on the configuration, to get a reply from all the targeted engine. When a reply comes in, the `Client` checks if the message id from the parent header of the message maches any outstanding `future` objects, if it does it resolves the matched `future` with the result from the message. The `Client` knows that it has gotten all the expected replies for an `apply` call when all the outstanding `futures` of the `AsyncResult` object for that call has been resolved.

The Number of Messages Sent Using the DirectView

Using the `DirectView` with N targeted engines, 1 call to `DirectView.apply` sends N messages to the scheduler, the scheduler relays the N messages to the engines. The engines then reply N messages and the scheduler

relays the N messages back to the `Client`. This means that in total $4N$ messages are sent between the `Client` and the engines for each call to `DirectView.apply`.

5.1.1 BroadcastView

The largest bottleneck of the `DirectView` is that it has to send 1 message per targeted engine, this is because of the way the Multiplexing Scheduler is implemented as just a simple relay for messages. Because the message content is the same for all the messages sent to the engines, with a more complex scheduler implementation 1 message should be enough. Reducing the number of messages from 1 per engine to just 1 per call to `apply` was the first improvement applied for the `BroadcastView`. The scheduler still needs to know which engines to relay the messages to, so the targeted engines ids is included as a list in the metadata of the message.

BroadcastView for a Coalescing Scheduler

```
def coalescing_apply(self, f, args, kwargs, block):
        msg_content = serialize(f, args, kwargs)
        targets = self.get_target_ids()
        metadata = {"targets": targets, "is_coalescing": True}
        message_future = self.client.send_apply_request(
                msg_content,
                metadata
        )
        self.client.outstanding.add(message_future.msg_id)

        result = AsyncResult(self.client, message_future)
        return result.get() if block else result
```

The above pesudocode describes how `apply` looks like for the Coalescing Scheduler. After having sent the message, the `Client` has to wait for all the replies to return from the engines. For the Coalescing scheduler this part is simple, because the `BroadcastView` is sending only 1 message and expecting only 1 reply, it could be done in the same way as in the `DirectView`. The downside of this implementation is that the `Client` has to wait for all the replies to come back to the scheduler before getting the accumulated reply, this makes it different from the `DirectView` where the `Client` can use the partial results as soon as they arrive from the engines.

BroadcastView for a Non Coalescing Scheduler

```
def non_coalescing_apply(self, f, args, kwargs, block):
        msg_content = serialize(f, args, kwargs)
        targets = self.get_target_ids()
        metadata = {"targets": targets, "is_coalescing": True}
        message_future = self.client.send_apply_request(
```

```
        msg_content,
        metadata
)
futures = []
for targeted in targets:
        msg_id = f'{message_future.msg_id}_{target}'
        future  = self.client.create_message_future(msg_id)
        self.client.outstanding.add(new_msg_id)
        futures.append(future)
result = AsyncResult(self.client, futures, targets=targets)
return result.get() if block else result
```

Figuring out how to deal with replies from the Non Coalescing Scheduler proved to be slightly more complicated than for the Coalescing Scheduler because the `Client` will receive N replies for N targeted engines while just sending 1 message to the Scheduler. The solution here was to send 1 message to the scheduler and create N message `futures` for messages that are not sent, but only used to identify expected replies. This was done by using the message id from the one message that is actually sent to create 1 `future` for each targeted engine. Using the message id from the original message and the targeted engine id as a message id for the `future` allows the `Client` to identify which `AsyncResult` the incoming replies belong to. These `futures` are then added to the `AsyncResult` object in the same way as the other cases. With the Non Coalescing Scheduler the replies from the engines will be relayed immediately to the `Client` in the same way that is done with the `DirectView`. This is makes it possible to use partial results as they arrive and is a possible advantage over the Coalescing Scheduler.

5.1.2 New Schedulers

In order to investigate possible new scheduler implementations, 3 different scheduler were made and benchmarked to figure out how to best optimize IPython Parallel. The `BroadcastView` and the new schedulers are all implementations of the SPMD (Single Program, Multiple Data) parallel programming pattern. The `DirectView` with the Multiplexing Scheduler can also be understood as an implementation of the same pattern, but the new schedulers and view should scale better with the number of engines in terms of performance.

5.1.3 Non Coalescing Scheduler

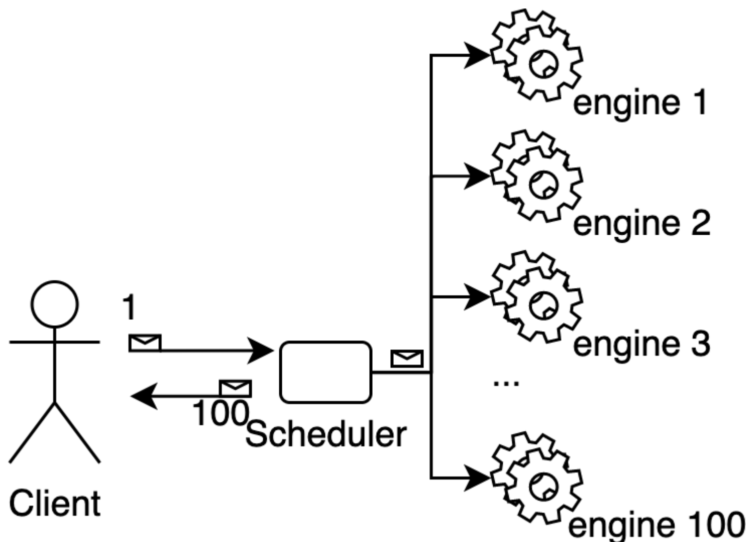

Figure 5.2: Message flow for the Non Coalescing `BroadcastView`. The `Client` sends 1 messages to the scheduler which is then relayed to 100 engines. The replies from the engines are relayed directly back to the Client.

The first and least complex new scheduler that was implemented was the Non Coalescing Scheduler. The idea that brought this scheduler to life was that it could be possible to save a lot of time on the `Client` by avoiding the duplicate work done by reducing the number of messages sent from the `Client` to the scheduler. For this to be possible, the new scheduler had to be implemented in Python. There already exists a Scheduler implementation for IPython Parallel written in Python, the Task Scheduler, so this was used as starting point. The Task Scheduler had implemented a lot of the functionality that would be useful for the new schedulers, so this was generalized and reused for the new implementations. The Non Coalescing Scheduler listens for messages from the `Client` over a `ZMQ` stream and for replies from the engine on another stream. When a message arrives from the `Client`, the scheduler gets the list of targeted engine id from the metadata of the message. Then for every targeted engine, the targeted engine id is appended to the original message id and used as the message id for a new message for the engine. The new message is then relayed to the targeted engine over the engine stream. When the replies arrives from the engine, they are relayed back to the `Client` immediately.

The Number of Messages Sent Using the Non Coalescing BroadcastView.

Using the Non Coalescing `BroadcastView` with N targeted engines, 1 call to `BroadcastView.apply` sends 1 message to the scheduler. The scheduler then sends N messages to the engines and receives N messages back. When the replies arrive at the scheduler from the engines they

are relayed back to the `Client` as soon as they come in. This adds up to a total of $1 + 3N$ messages sent for N engines for each call to `BroadcastView.apply`. This means that when targeting many engines, the Non Coalescing `BroadcastView` is sending close to 25% less messages than the `DirectView` ($1 + 3N$ vs $4N$).

5.1.4 Coalescing Scheduler

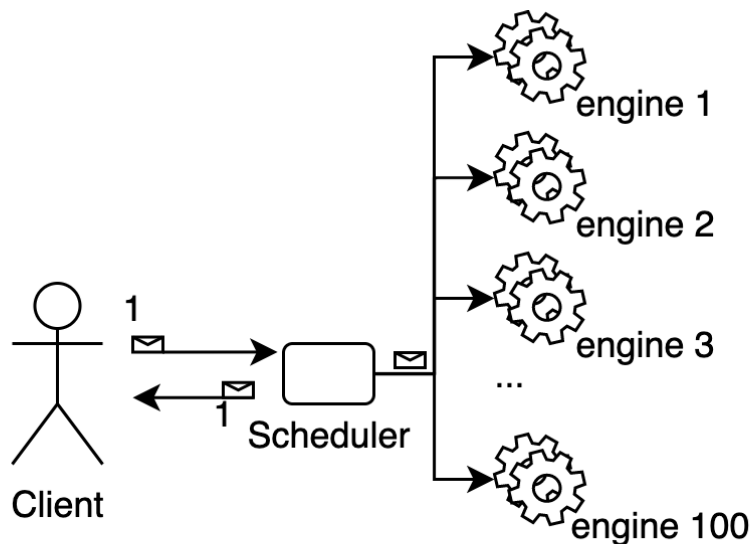

Figure 5.3: Message flow for the Coalescing `BroadcastView`. The `Client` sends 1 messages to the scheduler which is then relayed to 100 engines. The replies from the engines are accumulated in the scheduler and relayed as 1 message back to the Client.

To investigate if there would be a significant performance difference gained by replying only 1 message for each call to `apply` to the `Client` instead of 1 reply per engine, a Coalescing Scheduler was developed. The Non Coalescing Scheduler was used as a starting point as they would be very similar. The key difference on the relaying from the `Client` to the engines between the Coalescing and the Non Coalescing Scheduler is that the Coalescing Scheduler keeps track of which messages has been sent to the targeted engines. When the Coalescing Scheduler receives a reply from an engine, it saves the message buffers from the reply. Once all the engines has returned a reply, the scheduler merges all the accumulated replies together into 1 new message and relays it to the `Client`.

The Number of Messages Sent Using the Coalescing BroadcastView.

Using the Coalescing `BroadcastView` with N targeted engines, 1 call to `BroadcastView.apply` sends 1 message to the scheduler. The scheduler then sends N to the engines and receives N messages back. The messages are accumulated and merged at the scheduler and sent back to the `Client`.

This adds up to a total of $2 + 2N$ messages sent for N engines for each call to `BroadcastView.apply`. When N is high this means that the Coalescing `BroadcastView` is sending close to half as many messages as the `DirectView` ($2 + 2N$ vs $4N$) .

5.1.5 Spanning Tree Scheduler

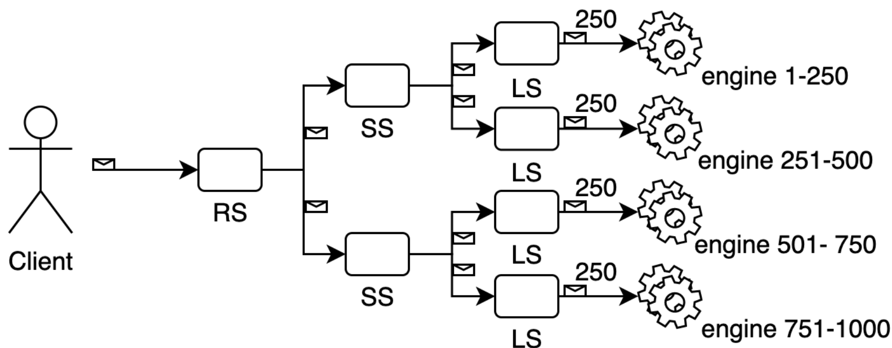

Figure 5.4: Message flow for the `BroadcastView` with the Spanning Tree Scheduler with depth = 2 and 1000 targeted engines. The `Client` sends 1 message to the root scheduler (RS), the root relays the message to two sub schedulers (SS). The sub schedulers then relays the the message to the leaf schedulers (LS). Finally the leaf schedulers relays the message to the engines, sending to 250 engines each.

The Multiplexing Scheduler is extremely fast because it's implemented in C and it's only relaying messages without doing anything to them. That's why the bottleneck in terms of performance with this scheduler/view combination is in the `DirectView` and not the scheduler for most cases. With having significantly reduced the amount of work done by the `BroadcastView` compared to the `DirectView`, this bottleneck was removed. Using the Non Coalescing and Coalescing Schedulers the new bottleneck would be in the scheduler itself because Python is inherently slower than C and because the new schedulers are doing more work parsing the messages. To solve this bottleneck a Spanning Tree Scheduler was suggested. This scheduler would consist of multiple schedulers sharing the workload and running in parallel. There would be some additional overhead to the runtime from having to do more sends over the network between the schedulers, but the performance gains from having multiple schedulers working in parallel should make up for that when the number of targeted engines increases past a certain threshold.

The Coalescing Scheduler was used as the starting point for the Spanning Tree Scheduler, the main difference being that instead of having only 1 scheduler running in 1 process, the Spanning Tree Scheduler can have multiple scheduler running in separate processes. A Spanning Tree Scheduler with depth = 0, will work in the same way as the Coalescing Scheduler. As illustrated in Figure 5.4, the `BroadcastView` sends one

message to the root scheduler. Then the root scheduler gets the list of targeted engines id from the metadata, splits the list in two equal halves and sets each half as a new targeted engine ids list for two new messages. The two messages are then relayed to the two next sub schedulers in the tree. If there are more sub schedulers in the tree, the sub schedulers does the same as the root scheduler and relays the two new messages to the next schedulers in the tree. When a message reaches a leaf scheduler, the leaf schedulers uses the targeted engine ids list to relay the message to the targeted engines. When the replies comes back from the engines, they are accumulated and merged on each level until the scheduler has received their expected replies from the engines or the sub schedulers they are connected to. Then the accumulated reply is relayed back to the previous scheduler in the tree until it hits the root scheduler which relays the final accumulated message back to the `Client`.

The Number of Messages Sent Using the Spanning Tree

Using the Spanning Tree scheduler with N targeted engines, 1 call to `BroadcastView.apply` first sends 1 message to the root scheduler. Then the is copied and sent sent down the tree, each sub scheduler sending two messages to their connected sub schedulers until the messages reaches the leaf schedulers. The leaf schedulers then sends a total of N messages to the engines and receives N messages back. When the replies arrive at the leaf schedulers from the engines they are accumulated until all the replies have returned. Then they are merged and sent back to the sub schedulers. The sub schedulers accumulate and merge their two replies and sends them back as one. The root scheduler finally sends an accumulated messages containing all the replies back to the `Client`. The number of messages sent from the root to the leaves is $2^{D+1} - 2$ where D is the depth of the Spanning Tree. The same number of messages are returned from the leaves to the root. Using the Spanning tree the same number of messages is sent between the `Client` and the schedulers and the schedulers and the engines as the Coalescing `BroadcastView`. This means that the total messages sent is $2(2^{D+1}) + 2 + 2N$ for N engines. For lower numbers of D and high numbers for N this is still a significantly smaller number than the $4N$ for the `DirectView`.

Using a Non Coalescing version of the Spanning Tree, the number of messages sent from the root to the leaves is still $2^{D+1} - 2$. The Non Coalescing version relays the messages immediately back through the spanning tree. That means that the number of sends for the replies going back through the Spanning Tree is $N * D$. This gives a total of $1 + 2^{D+1} - 2 + N * D + 2N$ messages for the `BroadcastView` with a Non Coalescing scheduler. That means more messages than the `DirectView` when D is ≥ 2.

Part III

Conclusion

Chapter 6

Results

To confirm that the new scheduler implementations are actually better than the existing one, a number of different benchmark simulating different typical usage patterns were made to compare how the performance scales with then number of engines for the `BroadcastView` vs the `DirectView`. The first benchmark tests the performance of sending large messages to the engines, but returning nothing. The second benchmark tests the performance of sending large messages to the engines and returning them back to the scheduler. The last important benchmark that will be presented in this chapter shows the performance when the `Client` sends multiple messages to many engines.

6.1 Comparing the DirectView vs the BroadcastView

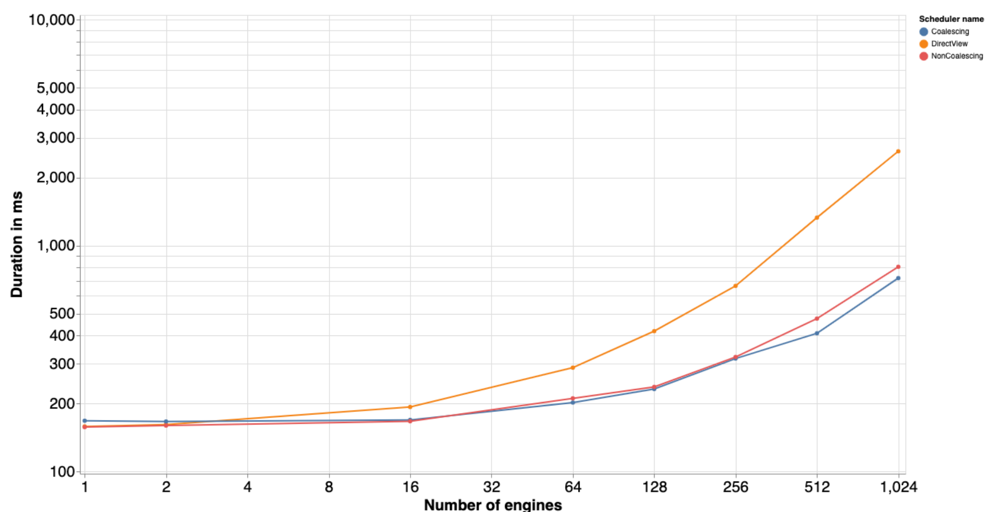

Figure 6.1: Benchmark results showing how long it takes to send a ~2MB message to different numbers of engines. The yellow line shows how the `DirectView` scales with the number of engines compared to the `BroadcastView`.

After having developed and tested the Non Coalescing, Coalescing and Spanning Tree scheduler it was decided that because they are so similar, it would be better to have just one Broadcast Scheduler that could be configured to replicate and combine the functionality of the three different implementations. Merging the schedulers also makes sense because it's easier to maintain and develop just one scheduler. This also makes it easier to run tests as instead of having to run tests on three different schedulers the tests can test one scheduler running with different configurations.

6.1.1 Pushing Large Messages

```
class PushMessageSuite:
        params = [
                [1, 2, 16, 64, 128, 256, 512, 1024],
                [
                        DirectView,
                        CoalescingBroadcastView,
                        NonCoalescingBroadcastView
                ],
                [1000, 10 000, 100 000, 1 000 000, 2 000 000]
        ]
        def time_large_message(self, engines, View, bytes):
                view.targets = range(engines)
                reply = View.apply(
                        lambda x: x,
                        np.array([0] * bytes, dtype=np.int8)
                )
                reply.get()
```

The above pseudocode describes how this benchmark works. The view `Client` sends a message to each of the targeted engines, varying the sizes of the messages and the number of targeted engines. This benchmark was made just to measure and compare how long it takes for the different schedulers to send a message of different sizes to the engines, but not receiving anything back. That's why the function provided as an argument for `apply` here only returns `None`. The usage pattern simulated here is the one of the best possible scenarios in terms of scaling performance for the `BroadcastView` because only the messages on the way from the `Client` to the scheduler have a significant size.

Using an example where each message is 1MB and the `Client` is targeting 1000 engines really illustrate why the `BroadcastView` is expected to perform much better here. With the `DirectView` the `Client` is first sending 1000 messages to the scheduler, then the scheduler relays each of these messages to the engines. That's 2 * 1000 messages of 1MB each, in total 2GB of data sent from the `Client` to the engines. The `BroadcastView` sends 1 message to the root scheduler. The root scheduler then relays the message sto 2 sub schedulers each. The sub schedulers relays them to 2 more sub schedulers and this continues until the messages gets to the

leaf scheduler where they are relayed to the engines. For this benchmark the `BroadcastView` was configured to have depth = 2. Using the formula described section 5.1.4: $2(2^{D+1}) + 2 + 2N$ the total amount of bytes sent can be calculated. However since the `Client` is only sending big messages one way, the amount of messages being sent to the engines is the interesting thing so the formula is divided by 2. That means that to send a 1MB message to 1000 engines, the `Client` has to send $(2^{2+1} - 2) + 1 + 1000$ messages of 1MB each. That's a total of 1007 messages or 1.007GB of data, almost half the amount of bytes compared to how much the `DirectView` has to send. In addition to that the gains from parallelism from having multiple leaf scheduler relaying messages at the same time should make the `BroacastView` the clear winner here. Since the engines are returning `None`, the time spent getting replies here will be negligible for large messages compared to the time spent sending messages to the engines. Figure: 6.1 confirms that the performance of the `BroadcastView` is indeed scaling much better for the case of large messages pushed to many engines.

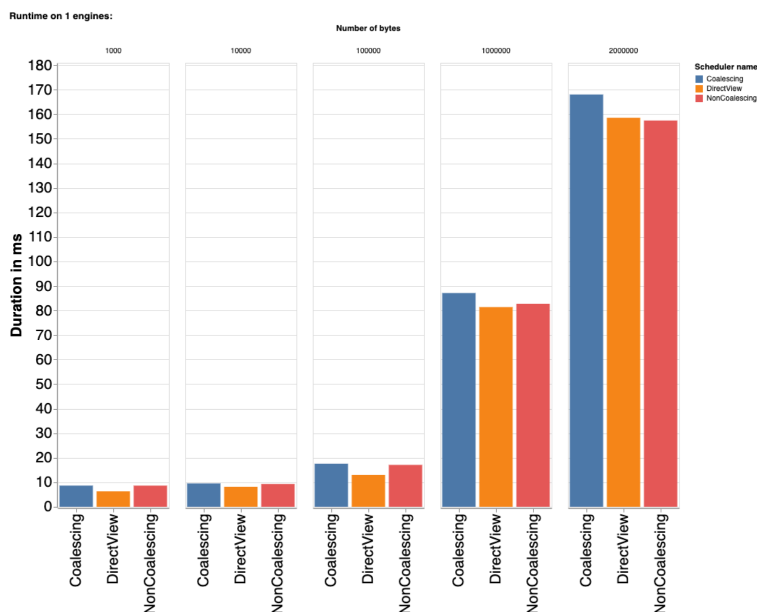

Figure 6.2: This chart shows the runtime for the push benchmark for different message sizes for 1 targeted engine. The blue bar is the Coalescing `BroadcastView`, the yellow is the `DirectView` and the red is the Non Coalescing `BroadcastView`.

For the most basic case in this benchmark, when the `Client` is only targeting 1 engine, the results are as expected and not very interesting. All of the schedulers are very fast for this easy case and there are only insignificant differences in the runtime between them. Since there is only one engine, the amount of messages sent between the `Client` and the engines is very low for all the implementations, so the times spent sending data is not the a big factor for this case. For the smaller messages the results show that the `DirectView` is slightly faster, what is seen here is the

Bytes	DirectView	Non Coalescing Broadcast	Coalescing Broadcast
1000	6.30	8.62	8.63
10 000	8.15	9.29	9.52
100 000	12.97	17.08	17.56
1000 000	81.41	82.77	87.12
2000 000	158.57	157.45	168.06

Table 6.1: Duration in µs for the push benchmark when targeting 1 engine.

overhead caused by having to send messages between the sub schedulers for the `BroadcastView`.

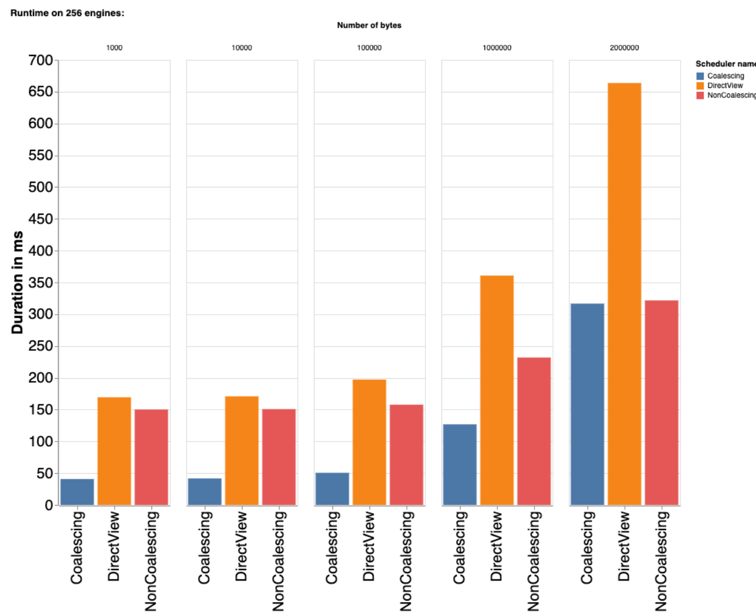

Figure 6.3: This chart shows the runtime for the push benchmark for different message sizes for 256 targeted engines.

Bytes	DirectView	Non Coalescing Broadcast	Coalescing Broadcast
1000	169.39	150.14	40.72
10 000	170.86	150.79	41.75
100 000	197.12	157.77	50.58
1000 000	360.79	231.87	126.82
2000 000	663.57	321.71	316.71

Table 6.2: Duration in µs for the push benchmark when targeting 256 engines.

For the case with 256 the Coalescing `BroadcastView` really shows its advantage over the other schedulers. For the 4 smallest message sizes, it is much faster than the `DirectView` and the Non Coalescing `DirectView`.

This is because the `DirectView` and Non Coalescing `BroadcastView` are relaying back 256 messages to the `Client` and handling many replies is much slower than sending just one small message. The Non Coalescing `BroadcastView` is slightly faster than the `DirectView` here for the 3 smallest messages and significantly faster for the 2 larger messages because it's just sending 1 message to the `Client` when the `DirectView` is sending 256. For the ~2MB message the Non Coalescing and the Coalescing `BroadcastView` is almost equally fast. This is because in this case the amount of data sent is so large that the time spent sending it from the `Client` to the engines is much higher than the time spent handling replies.

The `DirectView` is sending 2 * 256 2MB messages, a total of 512MB of data. Using the formula from 5.1.4 again to calculate the amount of bytes being sent, the `BroadcastView` is only sending $1 + (2^{2+1} - 2) + 256$ 2MB messages, a total of 263MB. From this it would be expected that the `BroadcastView` is slightly less than twice as fast as the `DirectView` in this case. The numbers show that it is actually slightly more than twice as fast, this is because the speedup from parallelism in the leaf schedulers.

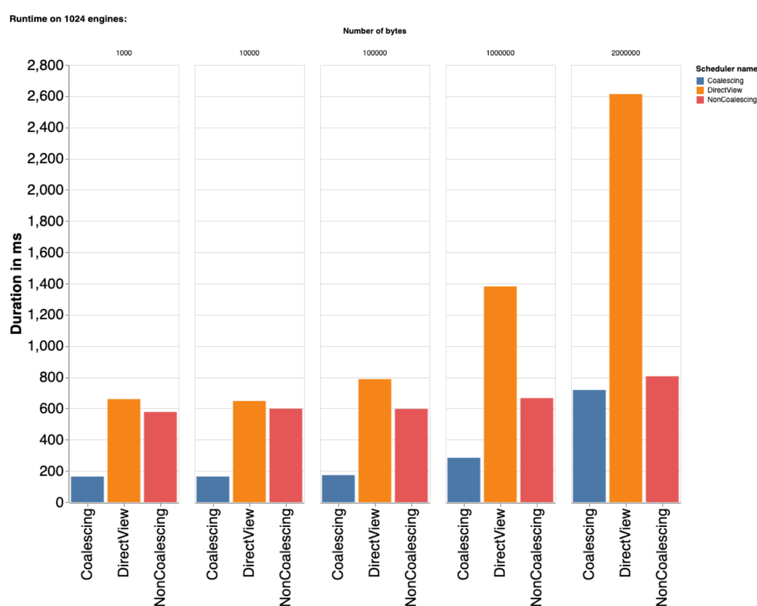

Figure 6.4: This chart shows the runtime for the push benchmark for different message sizes for 1024 targeted engines.

Bytes	DirectView	Non Coalescing Broadcast	Coalescing Broadcast
1000	660.04	577.71	164.12
10 000	647.8	598.62	164.27
100 000	787.68	597.03	173.36
1000 000	1381.59	666.1	284.16
2000 000	2613.55	806	718.34

Table 6.3: Duration in µs for the push benchmark when targeting 1024 engines.

Figure: 6.4 shows that the performance scales in the same way for 1024 engines as in figure: 6.3. In this chart the speedup from parallelism is much more apparent, for such a large number of engines the `BroadcastView` is more than 3 times as fast as the `DirectView` for the 2MB message.

6.1.2 Sending and Receiving Large Messages

After having measured the performance of the push benchmark, another benchmark was created to measure how the performance changes when the data is also returned from the engines back to the scheduler. The `BroadcastView` performs really well for the push benchmark because it sends much less bytes to the engines, but it doesn't have a way of reducing the amount of bytes that comes back. The `Client` handling of replies is known to be a bottleneck, so for this benchmark it the speedup from pushing messages on the `BroadcastView` might not be enough to make a difference.

Another disadvantage here for the `BroadcastView` is that the replies are sent multiple times between the sub schedulers to reach the `Client`. To receive a 1MB reply from 1000 engines, the `DirectView` receives first 1000 * 1MB of data from the engines to the scheduler, then the 1000 * 1MB of data is relayed to the engine. In total the `DirectView` is transmitting 2GB of data here. For the `BroadcastView` with a depth = 2, 1GB of data is received at the leaf schedulers. Then 1GB is relayed to the sub scheduler, that relays it further to the root scheduler before finally being relayed to the `Client`. So for the `BroadcastView` a total of 4GB of data is transmitted in this case, the extra data transmitted here compared to the `DirectView` means that the speedup from pushing data probably won't be enough.

```
class ThrougputSuite:
        params = [
                [1, 2, 16, 64, 128, 256, 512, 1024],
                [
                        DirectView,
                        CoalescingBroadcastView,
                        NonCoalescingBroadcastView
                ],
                [1000, 10 000, 100 000, 1 000 000, 2 000 000]
        ]

        def time_large_message(self, engines, View, bytes):
                view.targets = range(engines)
                reply = View.apply(
                        lambda x: None,
                        np.array([0] * bytes, dtype=np.int8)
                )
                reply.get()
```

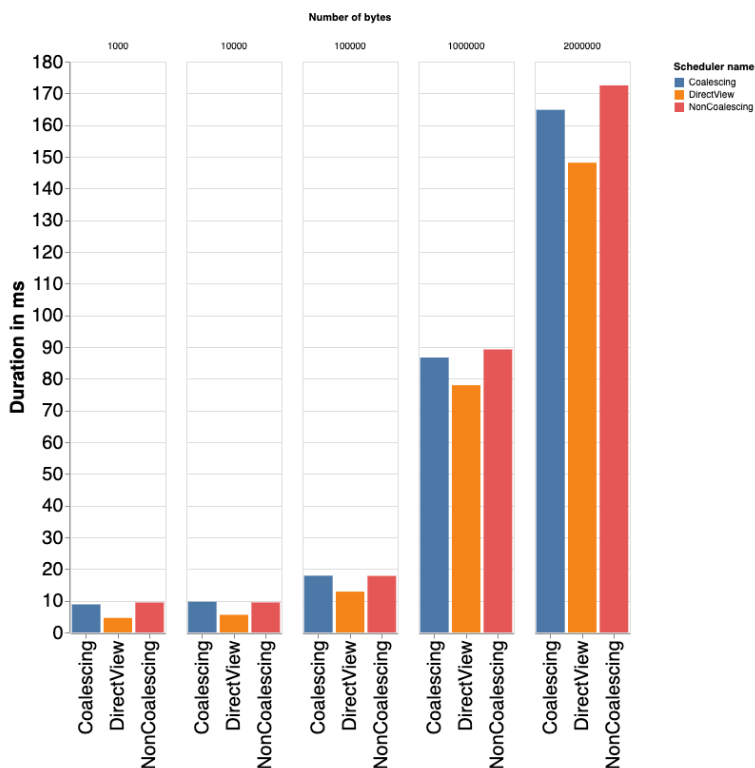

Figure 6.5: This chart shows the runtime for the benchmark for different message sizes for 1 targeted engine. The blue bar is the Coalescing `BroadcastView`, the orange bar is the `DirectView` & the red bar is the Non Coalescing `BroadcastView`

Bytes	DirectView	Non Coalescing Broadcast	Coalescing Broadcast
1000	4.58	9.45	8.85
10 000	5.56	9.47	9.69
100 000	12.88	17.83	17.9
1000 000	77.96	89.25	86.69
2000 000	148.14	172.51	164.8

Table 6.4: Duration in µs for the benchmark when targeting one engine.

Figure: 6.5 shows that when the views are targeting only one engine, the `DirectView` is faster for all messages sizes. This is to be expected as there is nothing to gain from parallelism with only 1 targeted engine because only one message is sent. The difference in runtime here shows the overhead from running Python code and the extra delay from having to relay the message between the sub schedulers for the `BroadcastView`.

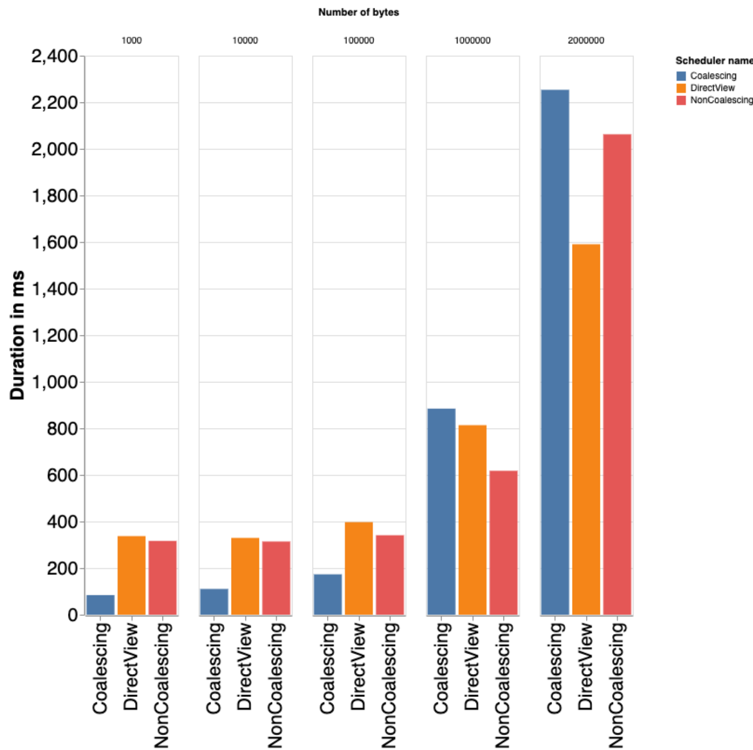

Figure 6.6: This chart shows the runtime for the benchmark for different message sizes for 512 targeted engines.

Figure: 6.6 show the benchmark results for 512 connected engines. As was expected, the Coalescing `BroadcastView` outperforms the `DirectView` for the smaller messages because of the accumulated reply that lightens the workload on the `Client`. The results for the 1MB message is interesting. In this case the Non Coalescing `BroadcastView` is the winner performance wise, while the Coalescing `BroadcastView` is slightly slower than the

Bytes	DirectView	Non Coalescing Broadcast	Coalescing Broadcast
1000	337.34	316.77	84.11
10 000	329.62	314.19	110.52
100 000	397.27	341.05	173.32
1000 000	813.7	617.82	884.62
2000 000	1590	2062.45	2253.55

Table 6.5: Duration in µs for the benchmark when targeting 512 engines

`DirectView`. What can be learned from this case is that the Non Coalescing `BroadcastView` is faster than the `DirectView` because because the speedup from pushing the data faster makes up for the disadvantage the Non Coalescing `BroadcastView` has when receiving replies. The `Client` is able to handle the large messages faster than they are being received, that's why the `DirectView` is faster here than the Coalescing `BroadcastView` because the Coalescing `BroadcastView` waits for all the replies from the engines before relaying 1 very large accumulated message (~512MB) back to the `Client`. For the case when the message size is ~2MB the `DirectView` outperforms the `BroadcastView` because the `BroadcastView` ends up transmitting significantly more data on the reply than the `DirectView` and the speedup on pushing the data doesn't make up for it.

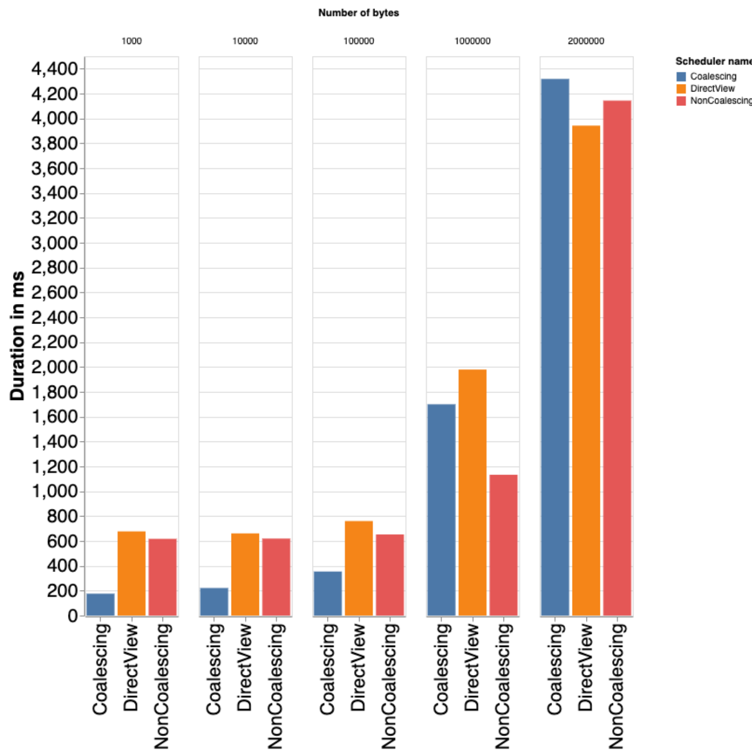

Figure 6.7: This chart shows the runtime for the benchmark for different message sizes for 1024 targeted engines.

Bytes	DirectView	Non Coalescing Broadcast	Coalescing Broadcast
1000	677.21	617.38	176.21
10 000	661.01	619.59	221.57
100 000	760.47	652.45	354.27
1000 000	1979.3	1132.67	1699.83
2000 000	3942.04	4143.3	4317.67

Table 6.6: Duration in µs for the benchmark when targeting 1024 engines

Figure: 6.7 shows the result for the case when the benchmark was run with 1024 targeted engines. For the smaller messages the Coalescing `BroadcastView` clearly outperforms the others again. For messages of ~1MB the difference between the Coalescing `BroadcastView` and the `DirectView` is very small though. For this case the Non Coalescing `BroadcastView` is the fastest. What can be seen here is that the `Client` is not the bottleneck and that the gains from parallelism makes the `BroadcastView` faster than the `DirectView`. The Non Coalescing `BroadcastView` is faster than the Coalescing version because the `Client` handles 1024 ~1MB replies faster than it takes the Coalescing `BroadcastView` to accumulate all the replies and relay ~1GB message to the `Client`. For the 2MB message case, the `DirectView` is again faster than the `BroadcastView`, but only by a small margin. Because there are so many targeted engines, the speedup from parallelism is high enough that the `BroadcastView` is only slightly slower than the `DirectView` because of the penalty of transmitting more data on the reply.

6.1.3 Sending Multiple Empty Messages

To compare the message flow between the different scheduler implementations a benchmark was made to measure the runtime of `apply` when the `Client` sends multiple messages to multiple engines. The benchmark is illustrated in the pseudocode bellow. The view sets N targeted engines, then sends M ~1kB messages to each engine. After sending all the messages, the benchmark waits until all the replies have come back.

```
class ManyMessagesSuite:
        params = [
                [1, 2, 16, 64, 128, 256, 512, 1024],
                [
                        DirectView,
                        CoalescingBroadcastView,
                        NonCoalescingBroadcastView
                ],
                [1, 5, 10, 20, 50, 75, 100]
        ]
```

```python
def time_async_apply(self, engines, View, messages):
        view.targets = range(engines)
        replies = []
        for i in range(messages):
                reply = View.apply(
                        lambda x: x,
                        np.array([0] * 1000, dtype=np.int8)
                )
                replies.append(reply)
        for reply in replies:
                reply.get()
```

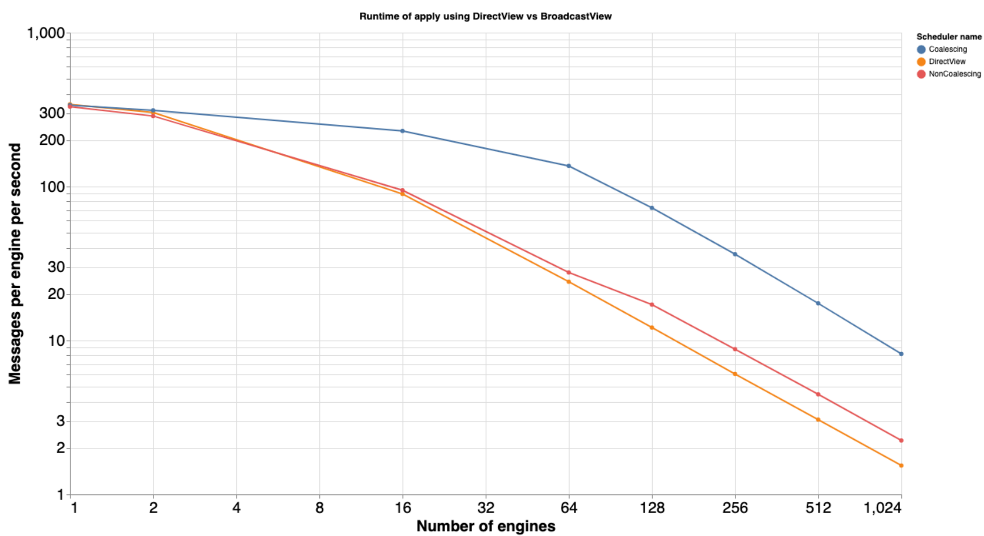

Figure 6.8: Benchmark for sending multiple smaller messages to the engines shows that the Non Coalescing `BroadcastView` performs only slightly better than the `DirectView`. However the Coalescing `BroadcastView` clearly performs better when number of targeted engines is ≥ 16

Engines	DirectView	Non Coalescing Broadcast	Coalescing Broadcast
1	342.06	330.36	337.89
2	302.57	287.07	312.70
16	89.57	94.63	229.73
64	24.08	27.65	136.15
1024	1.54	2.24	8.18

Table 6.7: Number of messages sent per second per engine for the different schedulers.

Table: 6.7 and Figure: 6.8 shows the results of this benchmark. The numbers are calculated from the time it took to send 20 messages to each engine. For 1 targeted engine, the `DirectView` has got the best performance. In this case the `BroadcastView` is slower because the

scheduler is implemented in Python, because the scheduler is doing some parsing with each message and because of the added delay of sending the message between the sub schedulers. When there is only one message being sent there is nothing to gain from parallelism. The way the scheduler is implemented, all the 20 messages will take the same path through the spanning tree each time, this means that the same scheduler is relaying all the messages to the engines. A more intelligent implementation could possibly improve on this by doing some load balancing and spreading the 20 messages over different schedulers, but it's not clear if this effort will make a difference or if the gains from load balancing will make up for the added cost of having more complex logic in the scheduler.

Already with 16 targeted engines the Coalescing `BroadcastView` shows a huge improvement on the `DirectView` by being able to send more than twice the amount of messages per second. Interestingly the Non Coalescing `BroadcastView` is performing almost as bad as the `DirectView` in this case. Because there is such a big difference between the performance of the Coalescing and Non Coalescing `BroadcastView`, what can be learned here is that for the case when the `Client` is sending 20 small messages to the engines the bottleneck must be in the way the `Client` handles replies and because the Coalescing `BroadcastView` is sending less messages in total than the Non Coalescing `BroadcastView`. For 20 messages per engine for 16 engines is 320 messages sent from the schedulers to the engines and 320 messages received at the schedulers from the engines. The only difference between the Non Coalescing and the Coalescing scheduler is that the Non Coalescing scheduler relays all the 320 replies immediately, while the Coalescing scheduler accumulates all the 320 messages and relays them as just one message to the `Client`. This shows that the `Client` is not able to process the messages as fast as they come in and that the `Client` is faster when processing just 1 bigger message instead of 320 smaller ones even though it has to wait longer for the 1 bigger message. For 1024 engines the `DirectView` was only able to handle 1.54 messages per second per engine. The Coalescing `BroadcastView` was able to handle 8.18 messages per second per engine for 1024 messages, this shows that for this case the Coalescing `BroadcastView` is a huge improvement on the `DirectView` with being able to handle ≥ 5 times as many messages per engine per second. Using the forumla from 5.1.4 again to calculate the the total number of messages sent for the case with 20 messages being sent to 1024 targeted engines. For the `DirectView` the formula is $4N$ where N is the number of engines, so $20 * 1024 * 4 = 81920$ messages sent for the `DirectView`. For the Coalescing `BroadcastView` the formula is $2(2^{D+1})) + 2 + 2N$. With $D = 2$, the total number of messages sent is $20 * (2(2^{2+1} + 2 + 2 * 1024) = 41320$. This shows that the Coalescing `BroadcastView` is sending almost half the amount of total messages, with the parallelism in the leaf schedulers also it makes sense that it is much faster here.

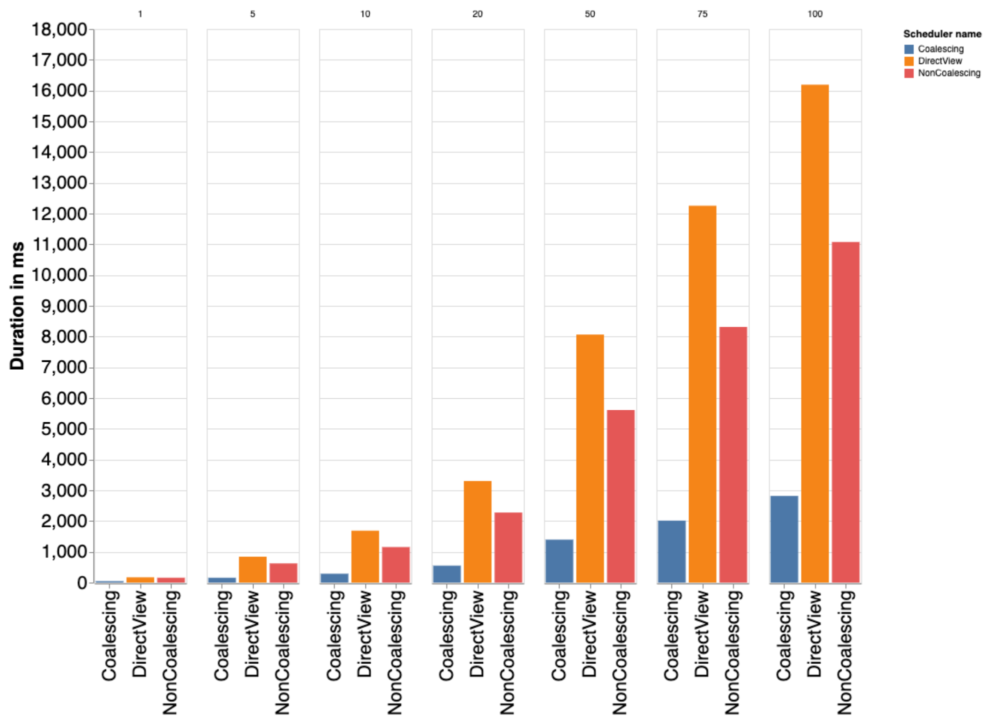

Figure 6.9: This charts shows the runtime of the benchmark when connected to 256 engines. The x-axis is the number of messages being sent to each engine. The orange bar is the `DirectView`, the red bar is the Non Coalescing `BroadcastView` and the blue bar is the Coalescing `BroadcastView`. Notice how much better the Coalescing `BroadcastView` scales with the number of messages being sent.

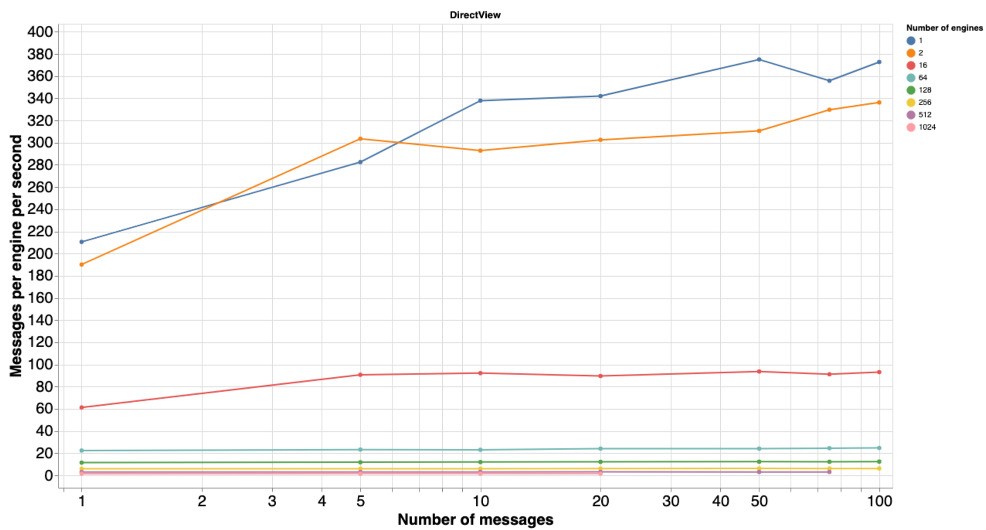

Figure 6.10: This charts shows the messages per engine per second for different numbers of messages sent with the `DirectView`. The lines represent different numbers of targeted engines. Blue line is 1 engine, orange line is 2 engines.

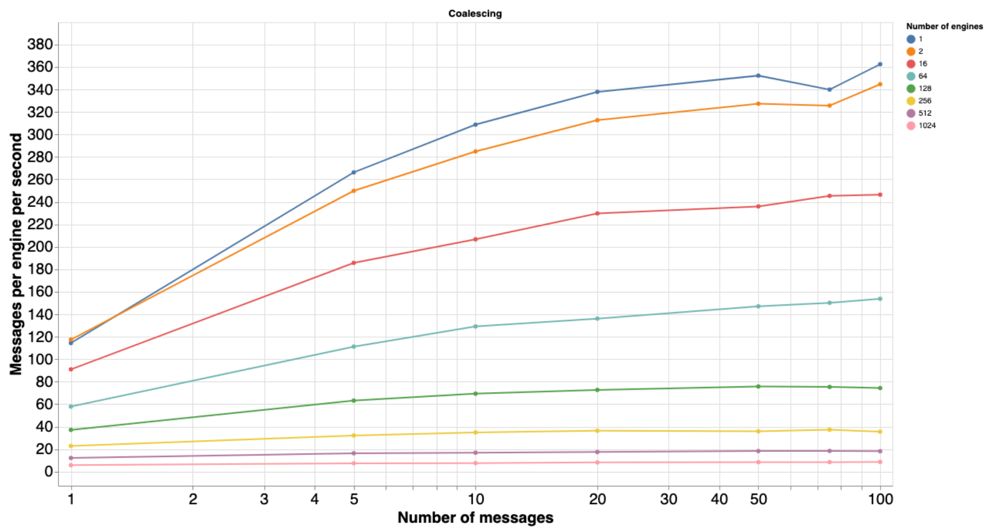

Figure 6.11: This charts shows the messages per engine per second for different numbers of messages sent with the Coalescing `BroadcastView`.

Figures: 6.10, 6.11 & 6.12 shows how many messages per engine per second the different views are able to handle. If the performance scales perfectly with the number of messages, the lines would be totally flat. The lines representing high number of engines are mostly flat, and that is good news. What is interesting with these charts though is that for the lower numbers of targeted and engines, and especially for the Coalescing `BroadcastView`, the performance seems to increase when the number of messages sent increases.

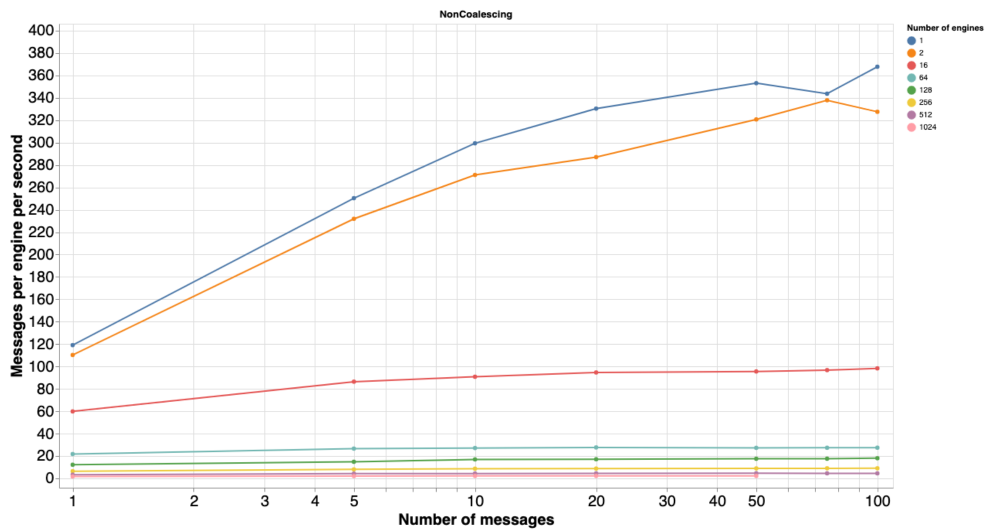

Figure 6.12: This charts shows the messages per engine per second for different numbers of messages sent with the Non Coalescing `BroadcastView`.

6.2 Figuring Out the Depth for the Spanning Tree

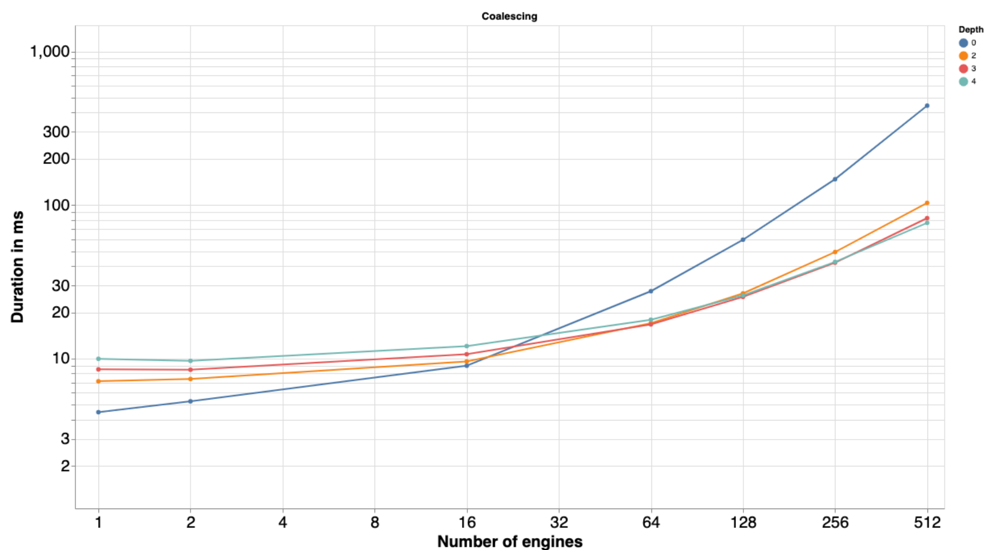

Figure 6.13: Benchmarked runtime of Broadcast Scheduler set to Coalescing and with different depth configurations. It shows that if the number of engines is less than 16 the performance gained by parallelism doesn't outweigh the added overhead of sending messages between the schedulers.

Depth	1 engine	64 engines	512 engines
0	4.48	27.49	444.02
2	7.12	17.01	103.34
3	8.51	16.7	82.23
4	9.97	17.92	76.6

Table 6.8: Measured runtime in µs for the Broadcast Scheduler set to Coalescing.

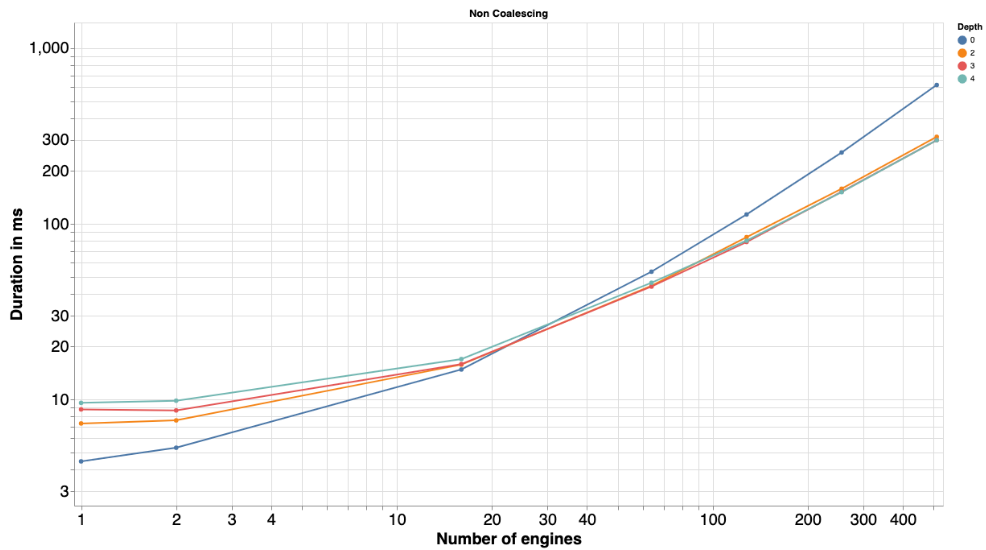

Figure 6.14: Benchmarked runtime of Broadcast Scheduler set to Non Coalescing and with different depth configurations.

To figure out what the optimal number of concurrently running schedulers is a benchmark were made to test when the performance gains from adding more engines makes up for the added cost of having more sends between the schedulers. To do this a benchmark was made that runs `BroadcastView.apply` with the `echo` function and and a 1kB NumPy array as arguments. This benchmark measures how long it takes to send and recieve a message with a ~1kB payload to all the targeted engines. At depth = 0, there is 1 scheduler running and it's relaying the messages directly to the engines.

```
class DepthTestingSuite:
        params = [
                [1, 2, 16, 64, 128, 256, 512],
                [0, 2, 3, 4],
                [true, false]
        ]
        def time_spanning_tree_depth(
                        self,
                        number_of_engines,
                        tree_depth,
                        is_coalescing
                 ):
                self.view = BroadcastView(is_coalescing, tree_depth)
                self.view.targets = range(number_of_engines)
                reply = self.view.apply(
                        lambda x: x,
                        np.array([0] * 1000, dtype=np.int8)
                )
                reply.get()
```

The pseudocode illustrates how the depth of the Spanning Tree was benchmarked, running the `time_spanning_tree_depth()` with the defined parameters. What can be seen in Figure 6.13 is that when there's only one targeted engine there is nothing to be gained by having more than one scheduler. This makes sense because there is only 1 leaf engine doing any work in that case so there is nothing to gain from parallelism. What is interesting with these numbers is that they show that the additional latency gained from adding more levels is approximately 1.5µs per level. That means the gains from parallelism must be larger than that number. With 64 targeted engines the gains from parallelism makes up for the added delay of having 2 or 3 levels, but 4 levels is still slightly slower. With more than 64 engines the advantage of having more schedulers becomes apparent, especially with 512 engines where the run time for depth = 0 is 444.02µs vs 76.6µs for depth = 4. These numbers are similar for the Non Coalescing configuration as well. However as can be seen in Figure: 6.13 and Figure: 6.14 the difference between depth = 2 and depth = 4 is not very significant, at least in the case that was benchmarked. So for the other benchmarks the depth for the Broadcast Scheduler was set to 2.

6.3 When to Choose the BroadcastView Instead of the DirectView

As can be seen in the benchmark results the Coalescing `BroadcastView` shows a clear speedup compared to the `DirectView` in many cases. The Non Coalescing `BroadcastView` shows some speedup compared to the `DirectView`, but significantly less so than the Coalescing `BroadcastView` in most cases.

When the number of targeted engines is very low (> 16), the `DirectView` is shown to be faster. The C implementation of the Multiplexing Scheduler is extremely fast and hard to beat for simple cases with anything implemented in Python. For small numbers of targeted engines the amount of duplicated serialization and sends done by the `Client` doesn't take long enough for the `BroadcastView` to be faster. When the number of targeted engines increase above 16 the performance of the Coalescing `BroadcastView` is shown to scale much better for some cases. In the cases where the `Client` wants to send many smaller messages (<1MB) to a large number of engines, the Coalescing `BroadcastView` is faster because the `Client` is doing much less work. $4N$ sends using the `DirectView` vs $2(2^{D+1})) + 2 + 2N$ with the Coalescing `BroadcastView`.

For the case where the number of targeted engines is $\geq$ 128 and the `Client` is sending many smaller messages, the Non Coalescing `BroadcastView` is shown to be significantly faster than the `DirectView`. This speedup comes from the parallelism in the scheduler that enables it to relay messages from multiple schedulers to the engines simultaneously. The Multiple Empty Messages benchmark also shows that when the number of targeted engines is really high ($\geq$ 512) and the number of messages sent is $\geq$ 50 the `DirectView` seems to stop working as the benchmark doesn't finish. For 1024 targeted engines the Coalescing `BroadcastView` is the only implementation that is able to handle sending 75 and 100 messages to each engine and even then for 100 messages it's only taking slightly less than 12 seconds. The Send and Receive Large Messages benchmark show that when the `Client` wants to send large messages ($\geq$ 1MB) the `DirectView` is the fastest implementation, even when for large number of connected engines. The Push Large Messages benchmark shows that for a Coalescing `BroadcastView` should be selected when the `Client` needs to only send data to the engines, without caring about receiving anything in the replies.

Chapter 7

Future Work

During the development of the new schedulers some new questions came up and there was some suggestions for further improvement that would be interesting to look at. This chapter will present the main topics that came up.

7.1 Smart Defaults

The `BroadcastView` in it's current state is very configurable. The user can set the depth of the spanning tree, set the scheduler to be Coalescing or Non Coalescing and select how many engines to target. This works well when the user knows very well how the view and the scheduler works and understand the problem they are trying to solve well enough to apply the right configuration. The downside of configurability is that it can be confusing for users that don't want to spend a lot of time getting familiar with how it works. Something that could interesting would be to have the depth of the spanning tree adapt to the number of connected engines so that the tree grows if a lot of new engines are connected. Another idea that would be interesting to explore but is more complicated, is to have the scheduler switch between Coalescing and Non Coalescing based on what kind of tasks the user wants to execute. Finally it could be really interesting to explore if it's possible to merge the `DirectView` and the `BroadcastView` into just one view that automatically selects which scheduler to use based on how many engines the `Client` wants to target and the size of the message that the `Client` wants to send. In the benchmark it was shown that the `DirectView` is faster when the number of targeted engines is low or if the message size is large, so this should be possible to automate in some smart way.

7.2 Fine Grained Performance Measurements

Instead of using benchmark tools and profiling to measure the performance of the scheduler, another approach could be to use timestamps applied to the metadata of the messages at different stages to get really detailed

overview of the performance of the different schedulers. One approach could be to add a timestamp each time a message is sent from the `Client`, each time it's relayed by a scheduler or sub scheduler and when it's received and sent at the engines. Then using the timestamps and calculating the difference between them it would be possible to know very deeply how long each stage of the process of sending and receiving messages is taking.

7.3 Benchmarking Real World Examples

The schedulers were benchmarked with benchmarks simulating typical usage patterns. However, these were mostly the best case and worst case scenarios and used to measure different specific parts of the schedulers and view implementations. Actual real-world applications might be using the scheduler in a totally different way. Finding and benchmarking som real world applications using IPython Parallel could be very interesting as they would give a more realistic view of how these new implementations will affect users compared to the synthetic benchmarks that might not really represent how the users are working with IPython Parallel.

7.4 Compare Benchmarks With Other Tools

While the benchmarks show that for certain cases the new schedulers are a significant improvement on the existing one, we don't learn anything about how the performance of the schedulers compare to other similar tools. Making some benchmarks to compare IPython Parallel to tools like Dask.distributed and pyspark would be very interesting because it would show if IPython Parallel is better or worse than the alternatives.

7.5 The Partially Coalescing Broadcast Scheduler

The Coalescing `BroadcastView` was shown to be faster than the Coalescing `BroadcastView` in most of the cases that were benchmarked. The Coalescing `BroadcastView` is faster because it relays only 1 message back to the `Client` when the Non Coalescing `BroadcastView` relays N messages fro N engines. When the Coalescing `BroadcastView` is accumulating replies from the engines, the `Client` sits idle waiting for the accumulated reply. This is wasted time when it could have been processing partial results. Something that would be interesting to explore is if the Coalescing `BroadcastView` could be faster if the scheduler relays partial results, for instance every time it has accumulated a certain number of messages or on some time interval. This could possibly be a real speed up especially since the accumulated message can get very big and slow to send over the network, but also because the `Client` could already be finished processing most of the replies before the last reply from the engine to the scheduler.

7.6 Getting the Code Into the Main Repository

The new scheduler implementation has been tested and shown that it works well. At the time this thesis was submitted, the code has been included in a pull request to the main IPython Parallel repository. Before it is actually merged into the master branch it needs a little bit more work. Mainly there are some code that's left over from when there was three different schedulers implement before they were merged together as one. Also there needs to be some tests and documentation before it's completely ready.

Bibliography

[1] URL: https://jupyter.org/ (visited on 02/06/2020).

[2] URL: https://ipyparallel.readthedocs.io/en/latest/ (visited on 02/06/2020).

[3] URL: https://ipyparallel.readthedocs.io/en/latest/intro.html.

[4] URL: https://github.com/tomoboy/ipyparallel (visited on 04/08/2020).

[5] URL: https://github.com/benfred/py-spy (visited on 04/08/2020).

[6] URL: https://github.com/airspeed-velocity/asv (visited on 04/08/2020).

[7] URL: https://pandas.pydata.org/docs/ (visited on 04/08/2020).

[8] URL: https://github.com/altair-viz/altair (visited on 04/08/2020).

[9] URL: https://cloud.google.com/compute/docs/gcloud-compute (visited on 16/08/2020).

[10] URL: https://numpy.org/ (visited on 17/08/2020).

[11] URL: https://github.com/jupyter/jupyter_client (visited on 18/08/2020).

[12] URL: https://docs.python.org/3/library/timeit.html (visited on 24/08/2020).

[13] G. M. Amdahl. 'Validity of the Single Processor Approach to Achieving Large Scale Computing Capabilities, Reprinted from the AFIPS Conference Proceedings, Vol. 30 (Atlantic City, N.J., Apr. 18–20), AFIPS Press, Reston, Va., 1967, pp. 483–485, when Dr. Amdahl was at International Business Machines Corporation, Sunnyvale, California'. In: *IEEE Solid-State Circuits Society Newsletter* 12.3 (2007), pp. 19–20.