SCALING UP A SCIENTIFIC COMPUTATION

Summary

Our Customer was implementing a new software-based medical device, processing patient data and performing unusually detailed predictive modeling to improve the treatment of patients with a targeted disease and chose to use Haskell as the programming language for it. The whole core technology was seen as a single pure mathematical function -- patient data in, predictive analysis out. While the Customer used the power of Haskell to get the software working -- they had a safe and effective prototype device -- but the vast amount of computation involved meant that it would need multiple CPU cores to run quickly. In theory, the use of Haskell should have meant they could just "turn on" native multicore support for more throughput, solving the problem automatically.

Unfortunately, despite their use of Haskell's native parallel/multicore programming features, the performance was not improving as expected as more CPUs were added. Performance stopped improving significantly past 4 cores, and in fact stopped improving at all approaching 10 cores. Further investigation actually showed slower performance above 12 cores.

FP Complete was tasked with investigating why the software was not scaling on a single machine as they had hoped, and if possible, allow scaling to much larger (18 core) machines and even beyond a single machine to a cloud-based "compute cluster" of big servers.

The problem became clear after a short period of intense research by FP Complete- as a memory-managed language, Haskell relies critically on garbage collection -- the recycling of unused memory space. Knowing the importance of fast garbage collection to good performance, and knowing that our Customer's application used and released large amounts of memory, we suspected the problem might be here.

We investigated further and found inefficient behavior of the Haskell garbage collector when managing a single heap accessed by many concurrent threads. To establish the source of the problem we took advantage of the instrumentation in the Glasgow Haskell Compiler runtime. Specifically, we were able to precisely measure how much time was spent garbage collecting, and how parallelized the garbage collector itself was. We discovered that as we added CPUs garbage collection took a higher and higher percentage of the total time -- up to 60% of the times in some cases, compared to less than 10% with one CPU. Using a tool called ThreadScope, it became clear that as the number of CPUs grew, the garbage collection pauses would become more and more frequent (since memory would be allocated faster, given the parallel work). The pauses would also be longer, given that they had to traverse a larger heap. This, coupled with each garbage collection pause blocking all the threads each time, caused the slowdown our Customer was experiencing.

Usually these problems are solvable using data structures that exert less pressure on the garbage collector. However, this kind of restructuring would have required broad changes to how the core algorithm of the device are structured, incurring significant costs and delays, and possibly

complicating the regulatory filing. We needed to find a solution to the scalability problem without modifying the Customer's core code.

Name: Fortune 500 company in pharmaceutical space

Industry/Sector: Software Based Medical Device

Project Type: High Performance Computing, Big Compute

Technology Used: AWS Cloud, Redis, functional programming

Project Requirements: Isolate the problem and develop a solution that left the Customers' proven core code intact.

The Solution

FP Complete quickly confirmed that the problem stemmed from Haskell's practice of recycling unused memory and worked around this problem by developing a platform capable of easily distributing Haskell computation across distinct processes.

Since the inefficiencies we were experiencing were due to Haskell's garbage collector having trouble managing a very big heap accessed by multiple threads, one way to mitigate the problem is to split up the work in many small heaps serving different threads or processes. Thus, garbage collection in one memory heap would not block users of the other heaps. We decided to distribute the workload across several separate single-threaded processes, each with its own heap, so that each would be very productive while still being able to distribute work across vast computing resources.

To distribute work efficiently, we developed a design using a master-slave architecture with the master orchestrating work to slaves. The slaves communicate with the master via messages over TCP sockets, and thus can be on the same machine or on separate machines -- a "two for one" win since the same code could support multi-core as well as multi-machine scaling. We distributed large chunks of work per message, to ensure that the latency in message-passing would not significantly affect performance.

The Customer's predictive model uses a statistical approach called time-series Monte Carlo, which explores thousands of possible evolutions over time. Thus, as a first attempt the master would orchestrate the work as follows: The master node is initialized with the initial states. When the master node needs to evolve a state, it transfers it to a slave node, which can be another process on the same machine, or on another machine across a network connection. The slave node executes the compute-intensive "evolution" function, and then returns the new state to the master. The master inspects the new states as they come from the slaves and continues evolving them. The problem with this first approach is that the states are relatively big -- in the hundreds of kilobytes per state, for a total of several gigabytes of states alive at any given time. This makes transferring them to a slave each time impractical.

We developed an algorithm capable of persisting the states on slave nodes, and then evolving them directly on the slaves without having to transfer them each time. Moreover, the master also take care to dynamically rebalance the work when a slave ends up having a backlog of states to evolve while others are idling. This keeps the communication traffic to a minimum while achieving good work balance across slave nodes. Crucially, all this management was handled outside the core regulated code, the mathematical analysis.

This allowed us not only to improve performance by 100% on a single 18-core machine, but also to distribute the computation across multiple machines, achieving latency reductions of up to 600% compared to the best performance we could get with the single-process version.

We choose this approach since it allowed us to achieve a dramatic speedup without any change to the core algorithm, and maintaining the output of the device byte-identical to the original version at all times, thus avoiding the burden of verifying the output twice, given the highly regulated environment the device operates in.

New Challenges for FP Complete

A remaining bottleneck was converting Haskell objects to streams of bytes that could be transferred as a message. This led to optimization work that culminated in the development of a new serialization library called “store” which improves the Haskell status quo by several orders of magnitude. With the Customer’s permission we released this library under a permissive open source license, following a string of FP Complete releases aiming to improve the Haskell ecosystem while protecting Customer confidentiality. This helps to reduce future maintenance work for the Customer, by bringing the open-source community to bear on maintaining the generic function over time.

Conclusion

We dramatically improved the program’s speed and scalability by building a framework to distribute the workload efficiently across multiple processes on multiple CPU cores, and then across multiple machines. We included the “middleware” infrastructure supporting these improvements as part of FP Complete's High Performance Computing framework, allowing for easy deployment of the scaled-up device on Amazon’s AWS cloud computing platform.

At the end of this optimization effort we doubled the performance of the device on the 18-core machines that we were using, and enabled us to gain a latency reduction of up to 500% by distributing the work across several machines.

As the resulting measurement showed, the distributed version of the software is already twice as fast on a single 18-core machine, and is then capable of scaling beyond one machine for even faster simulations. For example, the plot shows the time the simulation takes for 2 machines (36 cores) and 4 machines (72 cores), where the simulation time is slightly more than 3 minutes, while the best time we could get using the local version of the software is around 20 minutes. This is not only advantageous for the product, but also a boon to development: programmers testing changes or data scientists tuning parameters can iterate 6 times faster.

Moreover, we integrated this platform to distribute work into a broader high-performance computing (HPC) framework that allows the Customer to easily and reliably schedule requests and wait for responses. This framework allows multiple Customers, even across a remote Web API, to schedule work that is automatically and flexibly distributed across a cluster. The cluster can expand and shrink with no downtime, creating and deleting cloud compute machines to quickly react to changing demands. This HPC framework is architected to be extremely easily to deploy and scale, taking advantage of the AWS computing platform (both in the form of the EC2 and ElastiCache) and Docker containers to maximize reliability and minimize maintenance costs.

Having a direct handle on how the work is distributed let us integrate the strategy to distribute work into the Customer's broader strategy to deploy the medical device reliably and scalably. This means that we're able to use the hardware we are paying for as efficiently as possible, and we're able to scale seamlessly as the load increases.