A Scheduling Algorithm in Distributed Systems for an Optimized Cloud

Jason Childers
Vineet Pandit
Thomas Reed

Abstract
This paper is to show an optimal job scheduling algorithm for distributed computing resources. The algorithm attempts to schedule smaller jobs locally and larger jobs remotely. Unlike current distributed scheduling algorithms which keep the entire network active, our algorithm only activates clients and network connections as computing demand increases. We intend to pave the way for future research into ways the algorithm can be adapted to decrease power consumption and network utilization.
Table of Contents

Abstract
Introduction
Implementation
Data Generation
Data Analysis
  Total Job List Processing Time for N clients
  Total Throughput for N clients
  Throughput per Job Category
  Fraction of Total Jobs executed Locally vs Remotely, per category
  Local versus Remote Scheduling
Conclusions
Recommendations for Future Research
Attached Documents
Introduction

This paper shows an optimal algorithm for scheduling of tasks in a distributed network by outsourcing larger processes from one or more servers onto a small number of clients, filling up one client before moving to the next and only activating new parts of the net as computing demand increases. Our intention is to show a theoretical basis of how distributed scheduling can be concentrated onto clients selected optimally to minimize power consumption and network bandwidth usage. The algorithm defined in this paper shows that tasks can be optimally scheduled between a net of servers and clients to provide a total throughput which is greater than an experimental group consisting of non-networked systems.

Current algorithms for calculating in distributed or cloud networks 'shotgun' tasks into a wide net, leading to widely varied latency and significant wastage in both network bandwidth and power consumption. A wide dispersal of tasks leads to increased power consumption as otherwise idle and sleeping resources are awakened to handle a small number of tasks, as well as increased network bandwidth caused by multiple destinations, paths of non-optimal length, and overburdening of the network with commands to execute small jobs better executed locally. Little work has been done into scheduling algorithms to optimally distribute work among a computing network. Our paper is to show a theoretical basis for such an algorithm, paving the way for future work into practical implementations.

Power consumption is a prime determinant of available computing power. Data centers consume an obscene amount of power, which is expensive both to purchase and in creating the infrastructure to handle it. Consequently, Green Computing is a significant trend in the industry. Our algorithm intends to reduce power consumption by concentrating computing on a minimum number of clients and network elements, only awakening more when necessary and thus allowing idle resources to remain in lower power states until they are needed.

Network bandwidth is the second primary determinant of available computing power. Wide-area networks are typically measured in terms of their lowest common denominator, typically gigabits per second in today's world, but needs measured in terabits to petabits per second due to device proliferation and end-user service demands. Activating a limited number of clients and selectively choosing workloads to schedule locally rather than send through the network can reduce network overhead and open future research into further reducing network overhead through assigning clients based on hops in their route to minimize the number of occupied network connections.

Implementation

Our algorithm consists of several broad steps:

1. Start zero or more clients, then start one or more servers by passing them a list of
clients and whether to execute a file of jobs or use randomized job creation.

2. Run a producer-consumer model with five sizes of jobs (Extra Small, Small, Medium, Large, Extra Large) simulating job runtimes by sleeping a thread for a time synonymous with the size of the job. The consumer will have multiple threads on both local and remote systems.

3. Consumer will attempt to schedule all Xtra Small and Small jobs locally until local threads fill, then schedule remotely. It will attempt to schedule all other jobs remotely until remote threads fill, then schedule locally.

4. The consumer will block when all threads it can access are full.

5. Each server will terminate after a certain number of jobs and return useful statistics about job throughput and runtimes

Data Generation

Our program was constructed so that it would generate output without affecting the runtime of the algorithm itself. The output generated was crafted so it would show throughput (jobs completed/second) for the program as a whole and for specific job categories for both local and remote execution. In this way we hoped to show both that the algorithm functioned as desired and to highlight patterns in the output that could show how the algorithm would perform outside of our specific test cases.

We tested the program with two primary variable categories: number of clients and size of the job list. The number of clients was restricted from between 0 and 3, which should be sufficient to highlight general patterns. Though the program is capable of running with multiple servers, the program was not constructed to allow both servers to start simultaneously and so data points with multiple servers would be uncontrolled and thus irrelevant. We tested with four job lists of length 50, 100, 250, and 500. The job list was controlled by randomly creating a list of 50 jobs with an approximately equal number of jobs in each category and then repeating that exact list to reach the desired size. We also generated two additional lists skewed towards smaller jobs and larger jobs and likewise expanded them to the desired size.

Data Analysis

Total Job List Processing Time for N clients
Our input is a fixed-size list that is persistent between runs. Therefore, the first step is to determine how long the algorithm needed to execute the entire list. As expected, we found that the total execution time to complete the list decreased significantly with additional scheduling clients.

Throughput is measured as the total number of jobs completed per second by the entire program across all job categories. As expected, the algorithm displayed significant gains in throughput as the number of clients increased. However, the gains appear to be only asymptotically increasing with the number of clients rather than linearly increasing. The asymptote appears to be higher for larger job lists. Additionally, throughput for a small job list is significantly less than for larger job lists even at 0 clients.
Throughput per Job Category

To explain this errant behavior, we also measured throughput for specific job categories with differing size lists. This measured jobs completed versus time between a job being created by our producer and when it finishes executing.

Results are difficult to gain with great precision with low throughput numbers, but a pattern begins to emerge because of the restriction we placed on attempting to schedule Xtra Small and Small jobs locally and larger jobs remotely. Specifically, we see that smaller job lists generate higher throughput for Small and Xtra Small jobs and that larger job lists have an insignificant difference between category throughputs despite the restriction on size-location priority.

This behavior can be explained by the structure of the job list and size of the categories. As the program was written, Small and Xtra Small jobs took a fraction of a second, Average took 1 second, Large took 10 seconds, and Xtra Large took 25 seconds. This means that the largest 40% of jobs were consuming greater than 95% of the processing time. Our limited number of clients led to almost every thread being consumed by Large and Xtra Large jobs. For size 50 and 100 job lists, throughput was still high due to the large number of available threads compared to the size of the job list, but the 250 and 500 length job lists caused all remote threads to fill up, triggering the use of local threads for large and xtra large jobs and thus crippling the throughput of small and xtra small jobs. We can plot this behavior.
The following graphs are from a 3-client run and plot the percentage of local versus remote execution for different categories as the size of the job list increases.

As the graphs show, increasing the size of a job list has a drastic effect on the distribution of the largest categories of jobs. Xtra Small and Small size jobs never execute remotely on our inputs, but at list sizes of 250 and 500, where their throughput was noted to decrease dramatically in the above graphs, larger jobs begin competing with them in local thread space.

The theoretical asymptote is also a function of job size. Mathematically, Large and Extra Large jobs would consume approximately 95% of our processing time if all categories were equally likely. That means that, if we had an arbitrarily large number of clients, 95% of our threads would be filled with large and extra large jobs. This has two practical implications: First, as we approach a number of clients equal to the percentage time the largest 2/5th of jobs take, we saturate the network with those largest 2/5th and threads begin to become available to the bottom 3/5th of jobs. If correct, running the system with 19 clients and a very large job size would saturate the remote network with Large and Xtra Large jobs and leave the local machine open for smaller jobs. Second, adding additional clients beyond that point would begin yielding highly predictable and at most linear performance increases, pending the capabilities of hardware and infrastructure to support that many clients. Further research is suggested into very large cluster
sizes and job lists. We suspect that our algorithm is capable of significantly more than our testing would indicate.

The behavior of total throughput for very small lists also results from Large and Xtra Large jobs. Our program will produce threads until the file list is completed, then wait for all threads to finish being executed. This leads to a situation where a Large or Extra Large job is produced near the end of the list and the entire program will hang waiting for the single thread to complete. The program could be modified to chart when specific jobs are completed, but doing so could conceivably have created a performance impact and thus was rejected for our project’s scope. Again, very large cluster sizes and job list lengths would reduce this skew significantly.

Local versus Remote Scheduling
The last important data point is to show a decrease in the number of remote connections among a server-3 client trial.

Our algorithm displays significant reductions in message passing. As mentioned, lazy cloud algorithms attempt to schedule 100% of jobs remotely. Purely round-robin scheduling algorithms would reduce message passing by 25% at most as ¼ of threads are local. By contrast, our algorithm shows a 57% reduction in message passing. Continuing with the extrapolation above, we hypothesize that an arbitrarily large network would be able to reduce message passing to an asymptotic limit corresponding to the fraction of jobs prioritized to execute remotely. If the program were changed to prioritize Xtra Small, Xmall, and Average jobs locally and the net was expanded to a sufficient size, we hypothesize that we could reduce message passing by 65% or more.
Conclusions

In conclusion, we have shown that job scheduling algorithms can bring significant performance gains to multiple dimensions of a traditional distributed environment. Our scheduling algorithm that simply utilizes remote computing capacity for heavyweight jobs can realize throughput gains of approximately 400% and total execution time reduction of approximately 70% when distributed computing resource capacities are aligned with job production rates across a randomized distribution of job sizes. Additionally, if we assume that the network bandwidth costs of jobs are irrespective of size, we can calculate an approximate 55% reduction in network overhead. These numbers are subject to the constraints of our testing rather than our algorithm and actual performance gains could be higher. We leave open the opportunity for further refinements of the algorithm and underlying mathematics.

Recommendations for Future Research

While the algorithm functions as intended, significant additional work can be conducted into refining the algorithm and continued testing. We suspect a strong mathematical proof can be constructed to find an upper limit to the performance gains the algorithm could realize. Additionally, while our paper is intended to show the feasibility of such an algorithm, it leaves open the opportunity for future research into practical implementations in four major directions: optimal bandwidth reduction, optimal power reduction, mobile implementations, and cloud and distributed systems implementations.

First, our implementation schedules tasks across a minimal number of network connections. These connections are all homogeneous and no attempt to selectively activate them based on optimal criteria is made. One of the challenges with current distributed scheduling algorithms is that distributed resources, like those available in the cloud, are unlikely to be adjacent to each other on the same network segments. Future research can be conducted into optimal selection among heterogeneous connections to include number of hops, network latency, or current network load.

Second, our implementation isolates processing on a small number of clients and network connections. This is intended to show that the rest of the net can remain idle and in a low-power state, but further research can be conducted into optimally selecting among clients and network connections that minimize power consumption. Heterogeneous clients may have different power consumption depending on location or physical hardware. Additionally, hypervisor-level implementations of the algorithm could throttle a client’s CPU to balance power consumption between increasing CPU speed and activating another client.

Mobile computing is of significant importance due to the critical importance of power consumption. Depending on the smartphone app being used they may typically attempt to move all work to remote systems independent of Job size. This leaves local resources idle but in a
high-power state while waiting for a reply. Research can be done into the battery life implications of scheduling tasks locally in order to quicken the return to a low-power state.

A fourth direction for research is a cloud implementation. Currently, cloud providers sell computing capacity, only a fraction of which may be required by any specific cloud consumer at any given time. Unless an application and/or process requires its maximum amount of resources, those resources can be applied elsewhere. In this way, cloud providers can treat a single physical resource as multiple virtual or logical resources, selling a potential cycle of CPU power to multiple clients simultaneously, and further increasing margin of available capacity.

**Attached Documents**

- Algorithm code and supporting documents
  - P3/JobSchedulerClient/*
  - P3/JobSchedulerServer/*
  - P3/Makefile
  - P3/README
- Testing jobfiles
  - P3/tests/*
- Testing results *(JobSchedulerTestResults.pdf)*
- Program Flowchart *(JobSchedulerFlowChart.pdf)*