riskfocus

2013 – White Paper

Specialist Services

Job tuning for high performance in clusters

TABLE OF CONTENTS

Executive Summary

Clusters are a fact of life today in most financial organizations. In the ideal case, you would simply dispatch as many jobs as you have machines in your cluster and get perfectly scalable performance. Realistically however, extracting high performance from a cluster requires detailed knowledge of cluster topology, machine architecture and job characteristics. This white paper discusses some principles to guide your design choices when writing cluster software.

Basics

What is a job?

A job in the terminology of this paper refers to a single request made to a node in the cluster asking it to perform some work. For example, when trying to compute the PV of a portfolio a job may represent a request to value a number of trades.

The actual job request can be of different types: it can be a few bytes – just a list of trade-ids in a database to value. Or, it can be a very large, complex object that represents all the trades along with associated market data, reference data etc. – everything needed for the valuation.

The machine that runs the job can be set up in multiple ways – there might be 1 multi-threaded process that accepts jobs as they come in and runs them on different threads as they become available. Alternatively, each job might be run by a different process that is started up to run that job and then dies when the job is finished.

What are the factors that affect job performance?

On a single machine, the primary determiners of speed are the CPU, RAM and disk. The cluster adds an important 4th factor – the network. The network and local disk are usually interchangeable depending on the nature of the application – i.e. sometimes it's faster to get data from a local hard drive; other times it's faster to get data from RAM on another machine on the network.

Job characterization

In this section we discuss some of the properties and requirements of your jobs that you can tune to achieve better harmony with your cluster's hardware resources.

What is the usual pattern in slow cluster jobs?

In large clusters it's very common to find that a few jobs dominate the run time – most of the time, the majority of jobs will run correctly and quickly. But, there will always be outliers that, for a variety of reasons (hardware failure, network issues, other processes on the machine taking up resources) run dramatically more slowly. Since the entire task is considered complete only when all jobs have completed, these jobs become the bottleneck that determines how long the task takes to finish.

How many machines/nodes do you have?

This can be difficult to answer in some environments because some grid providers allow customers to request different configurations for the same task. For example some of your "nodes" may have 4 cores and 8 Gb ram while others have 2 cores and 4 Gb ram. The important thing for this process is to come up with a standard minimum size and make the other configurations multiples of that size. That allows for simpler reasoning and consistent scaling calculations for a specific task. This minimum size should be consistent with your job size and requirements. If the minimum configuration size is too small to complete the job successfully, it's not very useful. However, if the minimum configuration can run the specified job in some constant number of seconds, then you can build larger configurations that add more RAM or more CPU to achieve consistent reductions in job runtime.

If you have multiple kinds of jobs that have different requirements, then you can build out several different configurations that emphasize different things. For example, you can have a "high-cpu" configuration that has 16-32 cores but only 32 Gb RAM total. To this you can add a "high-memory" configuration that has 32 Gb of RAM but only 4 cores – thus giving each core access to much more RAM.

For the specific task that needs to be computed, how many jobs are launched?

If there are fewer jobs than there are machines then you will lose parallelism. With the same number of jobs and machines, you're hopefully taking advantage of all your available computational power. If your jobs take a while to complete though, you should have more jobs than you have machines – that way if some machine falls behind, another machine can start executing the same job. Also, if the jobs are smaller, then each one will finish faster. Additionally, if a machine crashes or hangs for a long time, you don't lose a lot of work if your jobs are small.

It's possible to take this to extremes though – let's say each job is tiny and takes 10 milliseconds to execute. And the time required to get a new job to a machine is 5 milliseconds. In this case, half the time your machines are sitting idle waiting for new jobs to be communicated to them. As a rough rule of thumb, the jobs size should be such that communication time is at least an order of magnitude less than computation time. In some cases this is achievable by creatively changing the job request. For example, you may simply dispatch a portfolio Id which contains all the trades you want to value instead of sending over all the trade objects on the network.

What is the distribution of times taken per job?

The job runtime distribution is usually very revealing. For example, you may find out that some specific jobs always run slower – this can be a hint that those jobs are fundamentally doing more work or needing some data from a slow data source (a cache might be required for just these pieces of data).

Another critical question this answers is the one above – if you know how long your job computation is taking, and you know how long the entire task takes to complete, the difference is the communication time – if this is high, then you're spending too much time shipping data back and forth over the network. So this can be another hint that you might need a cache or to make the job size larger.

Are there any patterns in the slow jobs?

Answering this requires knowledge of the data being processed. Usually people tend to break up work in terms they think are equal. E.g. for a portfolio of 1000 trades, a natural way to break it up might be 10 jobs with 100 trades each. But the trades are not necessarily equal. Some might require more computation than others to value – perhaps IR Swaps require some reference data that is slow to get, while futures can be valued very fast. Without considering these factors, you can end up with some jobs that are all IR swaps making them much slower than all the rest.

One way to see if this is happening is to run the same job multiple times on different nodes – if it's slow on multiple iterations on different machines, chances are it's not an accident that the particular job is taking so long – it's the nature of the data it's processing. Once you know this, you can zero in on specific areas to investigate.

Job runtime and mechanical sympathy

So far we've been looking at characterizing jobs based on the data they need and how long they take to run. But even apart from requiring different kinds of data or being intrinsically expensive to compute, there's a lot of ways in which the environment the job runs in can affect how fast it runs. In this section we list some of the questions to ask when trying to tune the compute environment to provide the best performance for specific tasks.

Are jobs CPU bound or I/O bound?

This is perhaps the most basic question to ask and also one of the most important ones. If your jobs are I/O bound – e.g. they write to disk frequently or spend much more time retrieving data from the network than they do actually computing results, then increasing CPU power is not going to speed things up whereas a small cache in RAM might do wonders.

A simple way to figure out whether your jobs are IO or CPU bound is to look at the various machine counters available. In Unix/Linux, running "top" will go a long way towards showing you how much CPU your process is consuming. Other tools like iostat, netstat etc. can be used to monitor network activity. That said, the best ways to find out how much time your code is spending I/O bound is simply to instrument it. Given that disk activity takes milliseconds as does most network activity, some code that captures the time taken by db/network/logging functions etc. is very cheap to add and will give you an accurate view of how much time your application is spending waiting for data.

It's very rare to find an application that is purely CPU bound or purely I/O bound. Most applications are a bit of both – they get some data in an I/O bound fashion and then process it in a CPU bound fashion. For most applications, the goal is to tilt the balance as far as possible towards the application being CPU bound – since that's the fastest part of the system. So for example, if you have a CPU bound system, you would have about the same number of threads as cores – since each thread will use a significant fraction of the core. But for an I/O bound system you can afford to have many more threads since each one will spend a significant amount of time waiting and will therefore use much less CPU.

Some applications don't ever hit this wall – for example caches tend to serve a large number of requests a second. But a 1 Gigabit network card can be saturated by a modern processor running at well below 100% utilization. In a case like this, you're not going to get to 100% CPU utilization – because your network card will have maxed out well before that. Here, you might try to see if there are any other jobs that require very little network access but do a lot of CPU intensive work that could possibly share this machine. That way you are at least not wasting resources.

What else is running on the machine?

This is a vital question to answer. All performance analysis is based on the core assumption that the machine environment is unchanging. If this is false, then your performance testing numbers will not be representative of reality and it's quite easy to end up wasting a lot of time in blind alleys optimizing things that turn out to not be the bottleneck.

The simplest way to know what else is running is to have a dedicated cluster with careful control over what is running. If this is not possible, it's important to take regular snapshots of what all the other processes on the system are at the beginning of the run and at regular intervals during the run. This allows runs from different days/weeks etc. to be compared against each other. If you find a particular run is very slow and at the same time you notice a process running that wasn't present earlier, it's a clue that this process may be stealing resources from your job.

Another oft-overlooked problem in this realm are regularly scheduled maintenance jobs – e.g. a virus checker kicking in and running seeks over the hard drives while your application is trying to read a multi gigabyte file from disk.

One way to guarantee that no other program is usurping resources from your application is to use quotas and/or priority bands. If your priority band is high enough the operating system will ensure that whenever your program is in a runnable state, everyone else is forced to wait. This has 2 problems however. First, at the organizational level there needs to be clear guidance about cluster usage in terms of what applications are running at what priority and at what times. The other factor is that the OS only thinks your application is runnable when it has CPU work to do – so for example if your application reads some data from disk, the OS will run other programs while your program is waiting for that data to come back. Normally this is fine – but what if there's a low priority process that's also trying to read data from disk. The disk might be busy servicing that other request and hence your request is going to take longer than usual. There's not a whole lot you can do about this – systems exist to allow disk/network rate limiting to make sure that no application overwhelms the disk but this is a difficult area to tune.

Perhaps the best way to see if this is a problem is to have a very carefully maintained test environment in which you can benchmark your code before running it on a cluster. If your timings in your benchmark environment match those in your cluster then you have some assurance that there aren't unknown factors that are causing delays in production. However if your cluster timings are markedly off from your benchmarks, that can at least give you a clue that it's time to take a closer look at what your code is sharing the machine with.

Should you design using processes or threads?

This trade-off is important in multi-core systems i.e. practically every system available today. If you are running a single process per job, without multiple threads, your code will be simpler and easier to debug and maintain. On the other hand, you'll be duplicating work and using more resources. Multiple threads can share data and resources like network connections and file descriptors. Used carefully, multiple threads can run faster with much less overhead than multiple processes. But the downside is that multi-threaded code can be difficult to get right – when one thread crashes, typically your whole process is going to go down. Also in languages like Python with its Global Interpreter Lock, the environment imposes constraints on when it's actually possible for multi-threaded code to run in true parallel fashion.

The answer to the trade-off obviously depends on what kind of work you're doing. If say for example, your jobs have absolutely nothing in common with each other then it makes sense to run multiple processes – since there's no sharing involved, the gains from running multiple threads are minimal compared to the simplicity and robustness of multiple processes. On the other hand, perhaps your code benefits from a large in-process cache. Then multiple threads might make sense because they can share read access to this cache instead of a multi-process scenario where each process has it's own cache potentially wasting a lot of memory with duplicated data.

Are the machines sized correctly for the jobs and vice versa?

The question here is whether resources are being wasted and whether the jobs have been sized correctly for the system they're running on. E.g. if the system is tuned for 4 cores and 8 Gb of RAM, then that's how machine resources should be sized – having 4 core systems with 10 Gb of RAM implies that 2 GB of ram is being wasted. Similarly, if it's impossible to get 4 core systems that have less than 10 Gb of RAM, perhaps jobs should be resized to be slightly larger to effectively use the 10 Gb of RAM – or the 2 Gb can be used as a cache. The principle is simply to use everything you have.

Often people look at CPU and RAM and decide that the machine/job sizing is accurate. But disk and network I/O are also factors to consider here. If you have 3 virtual machines each running 1 process that's writing to disk at full speed, it might make sense to install multiple disks on that machine otherwise your code will be running at 1/3 speed since it's contending for a shared disk. The moral as always is to be aware of all the ways your code interacts with and requires resources from the machine so that you can make an accurate estimate of what it needs to run.

If there is a threading system being used, is queuing a concern?

This is not necessarily true in all environments, but in many cases, a multi-threaded system implies that there's a listener thread that is waiting for jobs to come to the machine at which point it dispatches them to multiple worker threads for execution. The problem here goes back to one of job sizing. If jobs are not sized equally for processing time, it's possible that a machine will get say 5 small jobs and 2 large jobs to run on 2 threads. If the small jobs show up first, then they'll be executed speedily. But if the large jobs show up first, the small jobs will end up queued behind them, waiting for them to finish. This is not necessarily a problem in throughput oriented systems but if the system is latency sensitive then this can be a major problem.

One way a lot of systems try to solve this is by writing exotic code to manage time slicing within the process – i.e. regardless of job size a thread only works on a job for a certain amount of time. Then it goes back to see if there are any other jobs to be run. This kind of code is hard to get right and debug for an application developer. However, the operating system is very good at this kind of task-switching work. This means that if there's no way to size jobs equally and you see this kind of behavior happening in your system, switching to a multiple process model might help.

Is packing a concern?

If you're running your own cluster that you don't share with anyone else, this isn't a problem. But clusters are expensive to buy and operate – therefore you want to use the resources you have optimally. For example, if you have a very CPU intensive process that works completely in RAM, it makes sense to run a disk-based process on the same machine. Otherwise, the disk will lie idle while the CPU based process runs.

This is not so much a problem as an opportunity. If you can accurately characterize how your job uses CPU, RAM, disk and the network you can find opportunities to pack jobs together and get maximum resource utilization.

Monitoring and accurate testing

Is the job environment repeatable?

This is the first thing to think about when you're trying to figure out how to improve the performance of your code. It's important to have a stable environment in which it's possible to get repeatable, consistent numbers on how the code performs. Another critical factor is that this system must closely match the production environment. Otherwise it's possible you'll end up blaming the wrong component for the performance figures.

If you don't have this kind of environment yet, the first thing to do is to create one or to figure out where the variances in your current test environment come from. Without this sort of monitoring and oversight, it's quite likely you'll end up wasting a lot of time chasing down the wrong paths in search of performance improvements.

How do you test for performance improvements?

Typically, developers will make changes to the code and run the new code to see if it runs faster than the old system. It's common to do 2-3 runs and take the average to see if the new system is faster. However, given how complex cluster environments are, it makes sense to do tests with some amount of statistical accuracy. The Student's T-test is particularly useful in this case for determining how close the new average runtime is to the actual runtime as well as the confidence level that the new times are statistically better or worse than the old. Wikipedia has a good article about the Student's T distribution and its uses here: http://en.wikipedia.org/wiki/Student's_t-distribution

Using a repeatable consistent test platform with a statistically plausible test analysis methodology will give you a lot more confidence in your tests and a firm foundation on which to build your performance improvement strategy.

Is there adequate monitoring in place?

Monitoring is another key component of a performance strategy. Clusters are dynamic environments – the data entering the system changes all the time, machines can die, hard drives can go bad, network performance can degrade etc. Without monitoring for example, you won't know if all of a sudden your cache hit rate is not as high as you expect it to be. Or that a certain machine is running slow and that's resulting in all its clients slowing down as well.

Your monitoring system should allow you to inspect and trace all the variables you consider important to your system's performance. This doesn't just include system variables like RAM, network speed, drive speed etc. You should also be monitoring deeper variables in your application. For example, if you have a server that queues up requests for processing them by different threads, the average queue length over the last 1 minute window is a great statistic to measure to see if the server is falling behind. Combining these code statistics with system statistics like available RAM will allow you to correlate changes in your program's behavior with changes in the environment it's running in.

Conclusion

Most software engineering practice advocates working to interfaces and abstracting away the underlying hardware. It's easy to fall into this mindset when working with cluster environments as well. However the current state of the art in cluster software is behind single-machine libraries. In this case, lifting the cover off the abstractions and knowing your machine layout and hardware and how it will interact with your design choices can bridge the gap between lack-luster performance and fast, scalable cluster computing.

BECAUSE WE DELIVER

Why Risk Focus? … **Because We Deliver!**

We are…

A specialist, solutions and services firm targeting specific Capital Markets domains

Risk Management | OTC Trade Processing | Connectivity

A niche vendor with the market experience and delivery focus to get the job done

Business Skills | Product Knowledge | Technical Specialists

A full service provider offering both people and product to meet client's needs

Professional Services | Product Development | Partnerships

New York
Suite 620 & 800
336 West 37th Street
New York, NY 10018
Tel: +1 (917) 725 6006
Fax: +1 (917) 591 1616

London
117 Waterloo Road
London, SE1 8UL
Tel: +44 (020) 7921 0518
Fax: +44 (020) 7902 1133

www.riskfocus.com | info@riskfocus.com