Scale Up vs. Scale Out

June 2011

Executive Summary

This paper discusses the difference between multi-core concurrency (often referred to as the *scale-up* model) and distributed computing (often referred to as the scale-out model).

While the two models seem similar, in the practical sense they are very different. Only the scale-out model enables leveraging the power of multiple machines while also reducing failure and downtime incidence. However, this approach can also involve increased system overhead.

So, is it possible to choose between the two approaches? What factors should be considered? And how does the evolution of multi-core technology affect the need to choose?

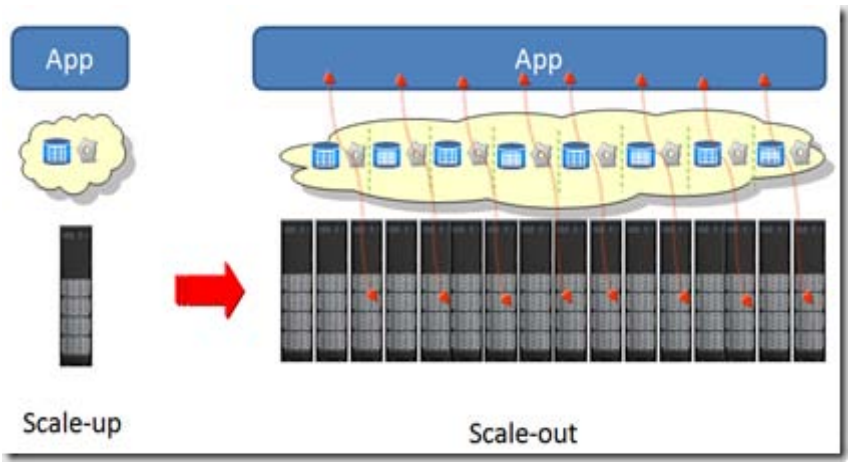
The Difference Between Scale-Up and Scale-Out

Scaling up, also known as **concurrent programming**, is one of the most common methods for utilizing multi-core architecture in the context of a single application. Concurrent programming on multi-core machines is often done through multi-threading and in-process message passing, also known as the Actor Model.

Scaling out, also known as **distributed programming**, does something similar by distributing jobs across machines over the network. There are different patterns associated with this model such as Master/Worker, Tuple Spaces, BlackBoard, and Map/Reduce.

Conceptually, the two models are almost identical, as in both cases a sequential piece of logic is broken into smaller pieces that can be executed in parallel. Practically, however, the two models are fairly different from both implementation and performance perspectives. The root of the difference is the existence (or lack) of a shared address space.

In a multi-threaded scenario you can assume the existence of a shared address space, and therefore data sharing and message passing can be done simply by passing a reference. In distributed computing, the lack of a shared address space makes this type of operation significantly more complex. Once you cross the boundary of a single process you must deal with partial failure and consistency issues. Also, the fact that you cannot simply pass an object by reference makes the process of sharing, passing, or updating data significantly more costly (compared with in-process reference passing), as you must deal with passing of copies of the data, which involves additional network and serialization and de-serialization overhead.



Choosing Between Scale-Up and Scale-Out

The most obvious reason for choosing between the scale-up and scale-out approaches is **scalability/performance**. Scale-out enables you to combine the power of multiple machines into a virtual single machine with the combined power of all its component machines, so that you are not limited to the capacity of a single unit. In a scale-up scenario, you have a hard limit -– the scale of the hardware on which you are running. Thus, a clear factor in choosing to scale out or up is whether or not you have enough resources within a single machine to meet your scalability requirements.

Reasons for Choosing Scale-Out Even If a Single Machine Meets Your Scaling/Performance Requirements

Today, with the availability of large multi-core and large memory systems, the likelihood of a single machine covering your scalability and performance goals increases. And yet, there are several other factors to consider when choosing between the two options:

- **Continuous Availability/Redundancy:** You should assume that failure is inevitable, and therefore having one big system is going to lead to a single point of failure. In addition, the recovery process will be relatively long, which could lead to extended downtime.
- **Cost/Performance Flexibility:** Because hardware costs and capacity tend to fluctuate over time, IT organizations generally prefer to have the flexibility to choose the optimal configuration setup at any given time or opportunity to optimize cost/performance. If your system is designed for scale-up only, then you are locked into a certain minimum price driven by the hardware being used. This might be even more relevant if you are an ISV or SaaS provider, where the cost margin of your application is critical to your business. In a competitive situation, the lack of flexibility could actually kill your business.
- **Continuous Upgrades:** Building an application as one big unit will make it harder or even impossible to add or change pieces of code individually, without bringing the entire system down. In these cases it is probably better to decouple your application into concrete sets of services that can be maintained independently.
- **Geographical Distribution:** If your application must span data centers or geographical locations to handle disaster recovery scenarios or to reduce geographical latency, you are forced to distribute your application and the option of putting it in a single box doesn't exist.

Is It Really Possible to Choose Between Scale-Up and Scale-Out?

Choosing between scale-out/scale-up based on the criteria outlined above might seem like a straightforward proposition. If the given machine is not big enough, just couple several machines together to get the capacity you need, and you're done. However, with the speed in which network, CPU power, and memory advance, the answer to the question of what your system requires at a given time might be very different than the answer a month later.

To make things even more complex, the gain between scale-up and scale-out is not linear. In other words, when switching between scale-up and scale-out results in a significant drop in what a single unit can do, as all of a sudden you must deal with network overhead, transactions, and replication into operations that were previously done just by passing object references. In addition, you will probably be forced to rewrite your entire application, as the programming model is going to shift quite dramatically between the two models. All this makes it fairly difficult to answer the question of which model is best.

Beyond those few very obvious cases, choosing between the two options can be difficult, and maybe even almost impossible.

Which brings up the next point: What if the process of moving between scale-up and scale-out were seamless -- not involving any changes to your code?

I often use storage as an example of this. In storage, when we switch between a single local disk to a distributed storage system, there is no need to rewrite the application. Why not have the same seamless transition for other layers of the application?

Designing for Seamless Scale-Up/Scale-Out

To get to a point of seamless transition between the two models, there are several design principles that are common to both the scale-out and scale-up approaches.

Parallelize Your Application

1. **Decouple:** Design your application as a decoupled set of services. “All problems in computer science can be solved by another level of indirection" is a famous quote attributed to Butler Lampson. In this specific context: If your code sets have loose ties to one another, the code is easier to move, and you can add more resources when needed without breaking those ties. In the given case, designing an application from a set of services that does not assume the locality of other services enables you to handle a scale-up scenario by routing requests to the most available instance.
2. **Partition:** To parallelize an application, it is often not enough to spawn multiple threads, because at some point they are going to hit a shared contention. To parallelize a stateful application it is necessary to find a way to partition the application and data model so that the parallel units share nothing with one another.

Enabling Seamless Transitions Between Remote and Local Services

The pattern outlined in this section is intended to enable seamless transition between distributed and local service. It is not intended to make the performance overhead between the two models go away.

The core principle is to decouple the services from anything that assume locality of either services or data, thus enabling switching between local and remote services without breaking the ties between them. The decoupling should happen in the following areas:

1. **Decouple the communications:** When a service invokes an operation on another service it is possible to determine whether that other service is local or remote. The communication layer can be smart enough to go through more efficient communication if the service is local, or go through the network if the service is remote. The important thing is that the application code is not changed as a result.
2. **Decouple the data access:** Similarly, abstract data access to the data service. A simple abstraction would be a distributed hash table, where the same code can be used to point to a local in-memory hash table or to a distributed version of that update. A more sophisticated version would be to point to an SQL data store where the same SQL interface would point to an in-memory data store or to a distributed data store.

Packaging Services for Best Performance and Scalability

Having an abstraction layer for services and data enables using the same code whether the data is local or distributed. Through decoupling, the decision about where services should live becomes more of a deployment question, and can be changed over time without changing any code.

In the two extreme scenarios, this means that same code can be used for scale-up only by having all the data and services co-located, or scale-out by distributing them over the network.

In most cases, it wouldn't make sense to go to either of the extreme scenarios, but rather to combine the two. The question then becomes at what point to package the services to run locally and at what point should to start to distribute them to achieve the scale-out model.

To illustrate, consider a simple order processing scenario that requires the following steps for the transaction flow:

1. Send the transaction
2. Validate and enrich the transaction data
3. Execute it
4. Propagate the result

Each transaction process belongs to a specific user. Transactions of two separate users are assumed to share nothing between them (beyond reference data, which is a different topic).

In this case, the right way to assemble the application to achieve the optimal scale-out and scale-up ratio is to have all the services needed for steps 1-4 co-located, and therefore set up for scale-up. You would scale-out simply by adding more of these units and splitting both the data and transactions among them based on user IDs. This unit-of-scale is often referred to as a processing unit.

To sum up, choosing the optimal packaging requires:

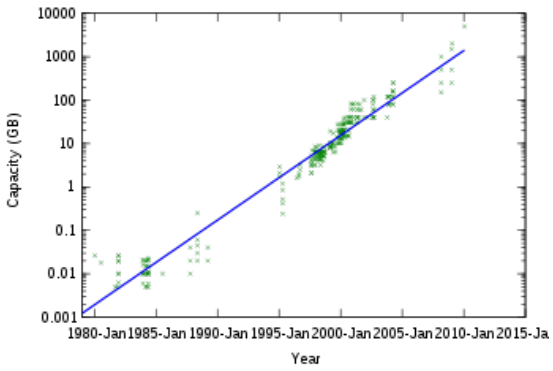
1. Packaging services into bundles based on their runtime dependencies to reduce network chattiness and number of moving parts.
2. Scaling-out by spreading application bundles across the set of available machines.
3. Scaling-up by running multiple threads in each bundle.

The entire pattern outlined here is also referred to as Space Based Architecture. A code example illustrating this model is available here.

Conclusion

Today, with the availability of large multi-core machines at significantly lower prices, the question of scale-up vs. scale-out becomes more common. There are more cases in which applications can be packaged in a single box to meet performance and scalability goals.

An analogy that I have found useful in understanding where the industry is going with this trend is to compare disk drives with storage virtualization. Disk drives are a good analogy to the scale-up approach, and storage virtualization is a good analogy to the scale-out approach. Similar to the advance in multi-core technology today, disk capacity has increased significantly in recent years. Today, we have xTB data capacity on a single disk.



PC hard disk capacity (in GB).The plot is logarithmic, so the fitted line corresponds to exponential growth

Interestingly, the increase in capacity of local disks did not replace the demand for storage, quite the contrary. A possible explanation is that while single-disk capacity doubled every year, the demand for more data grew at a much higher rate, as indicated in an IDC report:

> *Market research firm IDC projects a 61.7% compound annual growth rate (CAGR) for unstructured data in traditional data centers from 2008 to 2012 vs. a CAGR of 21.8% for transactional data.*

Another explanation is that storage provides functions such as redundancy, flexibility, and sharing/collaboration – properties that a single disk drive cannot address regardless of its capacity.

The advances in new multi-core machines will follow similar trends, as there is often a direct correlation between advances in data capacity and the demand for more compute power to manage it, as indicated here:

> *The current rate of increase in hard drive capacity is roughly similar to the rate of increase in transistor count.*

The increased hardware capacity will enable managing more data in a less time. In addition, the demand for more reliability through redundancy, as well as the need for better utilization through the sharing of resources driven by SaaS/Cloud environments, will force us even more than before towards scale-out and distributed architecture.

So, in essence, what we can expect to see is an evolution where the predominant architecture will be scale-out, but the resources in that architecture will get bigger and bigger, thus making it simpler to manage more data without increasing the complexity of managing it. To maximize the utilization of these bigger resources, we will have to combine a scale-up approach as well.

Which brings up the final point: We should not think of scale-out and scale-up as two distinct approaches that contradict one another, but rather must view them as two complementing paradigms.

The challenge is to make the combination of scale-up/out native to the way we develop and build applications. The Space Based Architecture pattern outlined here should serve as an example of how to achieve this goal.

References

- Parallel processing patterns:
  - Actor model
  - Master/Worker
  - MapReduce
  - Space Based Architecture (GigaSpaces Code example)
  - Tuple Spaces
  - BlackBoard
  - Space Based Architecture
- Managing Tera-Scale with Cisco UCS
- NAS systems evolve to cope with unstructured data growth

About GigaSpaces

GigaSpaces Technologies provides a new generation of virtualized application platforms. Our flagship product, eXtreme Application Platform (XAP), delivers end-to-end scalability across the entire stack, from the data all the way to the application. XAP is the only product that provides a complete in-memory solution on a single platform, enabling high-speed processing of extreme transactional loads, while scaling to meet any requirement – dynamically and linearly. XAP was designed from the ground up to support any cloud environment – private, public, or hybrid – and offers a pain-free, evolutionary path from today's data center to the technologies of tomorrow.

Hundreds of organizations worldwide are leveraging XAP to enhance IT efficiency and performance. Among our customers are Fortune Global 500 companies, including top financial services enterprises, telecom carriers, online gaming providers, and e-commerce companies.

U.S. Headquarters
GigaSpaces Technologies Inc.
317 Madison Ave, Suite 823
New York, NY 10017
Tel: 646-421-2830
Fax: 646-421-2859

U.S. West Coast Office
GigaSpaces Technologies Inc.
101 Metro Drive, Suite 350
San Jose, CA 95110
Tel: 408-878-6982
Fax: 408-878-6149

International Office
GigaSpaces Technologies Ltd.
4 Maskit St., P.O. Box 4063
Herzliya 46140, Israel
Tel: +972-9-952-6751
Fax: +972-9-956-4410

Europe Office
GigaSpaces Technologies Ltd.
2 Sheraton St.
London, W1F 8BH, United Kingdom
Tel: +44-207-117-0213
Fax: +44-870-383-5135