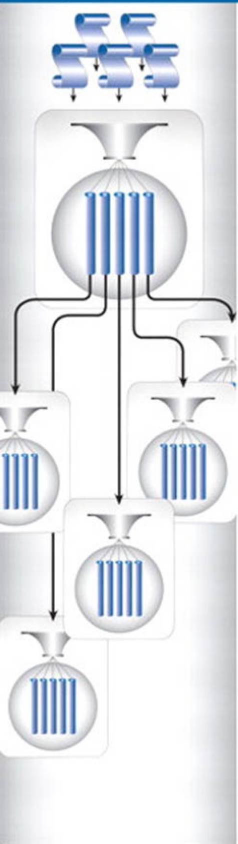
Addison-Wesley Information Technology Series



Software Pipelines and SOA

Releasing the Power of Multi-Core Processing

Cory Isaacson

Foreword by **Patrick Leonard**
VP, Engineering & Product Strategy
Rogue Wave Software

Library of Congress Cataloging-in-Publication Data

Isaacson, Cory.
Software pipelines and SOA : releasing the power of multi-core processing / Cory Isaacson.—1st ed.
p. cm.
Includes bibliographical references and index.
ISBN 0-13-713797-4 (pbk. : alk. paper)
1. Parallel processing (Electronic computers) 2. Computer software—Development. 3. Web services. I. Title.

QA76.58.I82 2008
004'.35--dc22 2008040489

ISBN-13: 978-0-13-713797-8
ISBN-10: 0-13-713797-4
Text printed in the United States on recycled paper by RR Donnelley in Crawfordsville, Indiana.
First printing, December 2008

Foreword

Multi-core hardware is the new normal. Major computer chip vendors have essentially halted the regular increase in CPU clock speeds that reigned for almost a half century in response to issues like power consumption, heat output, and unpredictability of quantum physics (to paraphrase Einstein, CPUs shouldn't play dice …).

Instead, they are using multi-core architectures to deliver increased processing power in place of faster clock speeds. Although this is a logical move, a large percentage of existing software applications cannot take advantage of the processing power on the additional cores, and they often run even slower due to reduced clock speeds in multi-core CPUs, setting up what could be called the Multi-core Dilemma.

In general, the Multi-core Dilemma applies across the spectrum of programming languages—Java, C#, C++, etc. This is why major technology vendors are investing heavily in research intended to lead to the next generation of programming environments. But what about the software that has already been written? The reality for any software application is that to benefit from multi-core, the application must either be written to be multi-threaded, or it must be in a container that effectively makes it multi-threaded.

There is no "plug-and-play" solution, but there are development tools and containers available that can help with the Multi-core Dilemma for many use cases. There are not, however, many good methodologies for solving this problem. In *Software Pipelines and SOA,* Cory Isaacson outlines a systematic, logical approach for planning and executing the move to multi-core.

This hardware trend will create a tectonic shift in the software industry as billions of lines of code are migrated, optimized, or rewritten to take advantage of multi-core hardware. Practical, logical approaches will be essential to making this transition smoothly.

Now that parallel computing has moved from being an edge case to being a common requirement for enterprise software, enabling applications to run in parallel can't be limited to only the most experienced programmers. *Software Pipelines and SOA* describes several techniques for bringing parallel computing to mainstream software development.

For example, one technique for making parallel computing happen throughout your development group is to separate your concurrency model from the application logic, much as you have your data and UI layer separate from the main business logic. Doing so allows feature developers to focus on the application functionality without having to worry about explicitly threading at design time. In addition, it can be an effective technique for migrating existing single-threaded applications into multi-core environments.

In addition, *Software Pipelines and SOA* discusses the connection between service-oriented architectures and multi-core. The basic approach is to treat your application as a collection of services and deploy a container that can run multiple instances of those services.

Using this link between SOA and multi-core, services can be a key part of your concurrency model. By separating concurrency from application logic, you can make the migration of existing applications to multi-core much simpler and enable more effective building of new parallel applications. It also makes it much easier to reconfigure (rather than recode) your applications, then continue to optimize them and move to new generations of hardware—from 2 and 4 cores to 8, 16, 32 … 128, and beyond. Designing enterprise applications in a service-oriented architecture makes it easier to separate concurrency from application logic, so that they work together.

There is work involved if you have a monolithic application, but it is still significantly less than rewriting. If you plan to use a container, make sure that it can handle your business application requirements, which might include message ordering, forks and joins in the business process, human interaction, and long-running processes.

Most parallel computing approaches use traditional multi-threaded programming, a "thread-level" approach. *Software Pipelines and SOA* describes a "service-level" approach that can provide a way to move to multi-core that requires less effort and is more configurable. It complements, rather than replaces, the traditional thread-level approaches.

Moving your existing applications to multi-core takes some planning, but it might not be as much work as you think. Design a solid concurrency model, and your existing applications can continue to serve you for years to come. *Software Pipelines and SOA* provides a great road map for getting there.

Patrick Leonard
VP, Engineering & Product Strategy
Rogue Wave Software
pleonard@roguewave.com

Preface

We're now in the multi-core era. As consumers of computing power, we've all come to expect a never-ending increase in power, and CPU manufacturers are now using multi-core processors to continue that long-standing trend. If we want to take full advantage of this enormous capacity, our business applications must "do more than one thing at a time." However, traditional parallel computing methods (such as multi-threading, SMP, and clustering) are either limiting or extremely difficult to implement—especially when used on top of application components that weren't originally designed for a parallel world.

Software Pipelines architecture is a new architecture that specifically addresses the problem of using parallel processing in the multi-core era. It is a new approach to the problem. Pipeline technology abstracts the complexities of parallel computing and makes it possible to use the power of the new CPUs for business applications.

We wrote this book primarily for software architects, application developers, and application development managers who need high-performance, scalable business applications. Project managers, software quality assurance specialists, and IT operations managers will also find it useful; however, the main focus is software development. Our intention was to make the book as applicable as possible, and to provide tools that you can quickly learn and apply to your own development challenges.

The book is divided into four sections, which we'll describe in this preface.

Pipelines Theory

The Pipelines Theory section, Chapters 1 through 5, covers the following topics:

- How pipelines work, including the fundamental concepts and underlying theory of Software Pipelines
- What pipelines can accomplish
- Methods for applying Software Pipelines
- Pipelines Patterns, including various ways to apply Software Pipelines in business application scenarios, setting the stage for the examples in later chapters

As the foundation for the remainder of the book, this section is appropriate for all readers. If you're a software architect or an application developer, you should definitely study this section first. If you're reading the book from a managerial perspective, or if your interest is more general and less technical, you can focus on just this section.

Pipelines Methodology

The Pipelines Methodology section, Chapters 6 through 13, shows how to implement Software Pipelines by using the step-by-step Software Pipelines Optimization Cycle (SPOC). To illustrate how the methodology works, we use it to solve a business problem for a fictitious example company, the Pipelines Bank Corporation (PBCOR). In each chapter we present a new step, then show you how we used the step in our PBCOR example.

This section will be of interest to all primary audiences of the book, including project managers. The PBCOR examples get into a fair amount of technical detail; therefore, application development managers might want to skip over the more complex examples.

Pipelines Examples

The Pipelines Examples section, Chapters 14 through 22, contains code examples based on the reference Pipelines Framework we developed for the book. We've included examples for each main Pipelines Pattern from the Pipelines Theory section. You can use these as guides for applying Software Pipelines directly to your own real-world applications.

This section is for software architects and application developers, the roles directly involved in pipelines implementation. In addition, IT operations managers will find it helpful to read the configuration sections, which show how to modify the scalability of an application without modifying the actual application components.

We recommend that you read the first three chapters of this section in detail. These basic chapters include Chapter 14, "Hello Software Pipelines"; Chapter 15, "Scaling Hello Software Pipelines"; and Chapter 16, "Additional Pipelines Router Configurations." After that, you might prefer to scan the more advanced examples in Chapters 17 through 22, then concentrate on the ones that most apply to your specific application scenarios.

The Future of Software Pipelines

In the final section we tell you about the future we envision for Software Pipelines architecture. There are plenty of greenfield areas that can be developed, and it is our hope that this section will inspire readers to help move the technology forward into the mainstream.

Conventions

In our examples, when we present a section of code or XML, refer to a command, or refer to a code element, we'll use a monospaced font, for example, `<pipelines-distributor>`. For names of components, such as services, clients, and distributors, we'll use an italic monospaced font, for example, *`Distributor1`*.

The Web Site

We've established a Web site for Software Pipelines technology at softwarepipelines.org. The site is for readers of the book and for anyone else who is interested in using or advancing the Software Pipelines architecture. You can download the following items from the site:

- Tools and sample report templates for the Software Pipelines Optimization Cycle (SPOC) methodology
- Source code for the reference Pipelines Framework
- Complete source code for all examples in the book
- Articles and discussions on pipelines technology and its applications

We hope you find Software Pipelines as exciting as we've found it, and that you take this opportunity to capitalize on its capabilities and use it to help overcome your own performance and scalability challenges.

Introduction

Throughout IT history, professional developers have searched for ways to enhance the performance of critical business applications. The computer industry has tried to provide the answer time and time again, and we've seen plenty of solutions, architectures, and approaches. Obviously, the problem is not a minor one. In today's information-based economy, companies often succeed or fail because of their software performance. There are abundant examples: banking systems, trading systems, call center operations, reporting services, and many others—they all depend on applications with high-performance requirements. In these industries, viability is directly tied to the speed at which a company conducts business. A slow, inadequate, or unresponsive application is damaging to operations and the bottom line; ultimately, it can literally kill or impair the organization that relies on it. And there is no end in sight; as we depend more and more on the exchange of information to do business, performance will lag behind demand even further.

There's an additional problem. Simply achieving faster performance of individual components isn't always enough. If your company installs a new application, if your business expands, or if your data volume rises, you may suddenly need an order-of-magnitude increase in performance—five, ten, twenty times or more.

Another vector is also critical: How fast can you adapt your software to meet new needs and competitive threats? The popularity and rapid adoption of service-oriented architecture (SOA) is hard evidence of the demand for more flexible software systems.

SOA is a superior technology. Compared to earlier trends in IT architecture, SOA delivers better on its promises. But it presents its own challenges. If you're using SOA for development, it's even more important to address performance and scalability, because of the following factors:

- In general observation, SOA demands significantly more computing power from a system than earlier monolithic or tightly coupled designs.

- The very notion of loosely coupled services implies message-centric application development. Developers not only have to write traditional processing logic; they also have to handle message transmission, validation, interpretation, and generation—all of which are CPU- and process-intensive.
- As more organizations use SOA, we can expect messaging volume to explode and put a tremendous load on existing IT systems. The potential for adverse effects will escalate.

Predictions show that over the next year or two, organizations using SOA will run into performance issues. This is nothing new; historically, each time the business world adopts a new software architecture, it suffers through growing pains. In the past twenty years, the shakeout period for each new major paradigm shift in software development has lasted about one to three years for a given evolutionary phase (any early J2EE user can attest to that). During that time, businesses gradually adopt the new design, and while doing so, they face significant performance- and scalability-related problems. In many cases software developers cannot overcome the steep learning curve; many projects end in outright failure when the deployed application doesn't perform as expected.

Until recently, hardware was the saving grace for such immature architectures. Whenever the computer industry made a significant advance, mostly in CPU performance, performance bottlenecks could be fixed by plugging in a faster chip or by using some other mechanical solution. That advantage is now gone. We've hit a plateau in microprocessor technology, which comes from physical factors such as power consumption, heat generation, and quantum mechanics. The industry can no longer easily increase the clock speed of single CPUs. Therefore, for now and the foreseeable future, CPU vendors are relying on multi-core designs to increase horsepower. The catch is that if you want to take advantage of these new multi-core chips, your software must implement parallel processing—not a common capability in the majority of today's applications.

Let's sum up what today's businesses really need from their software architecture:

- A practical approach to parallel processing, for performance and scalability
- Flexibility, to enable the business to adapt to market and other environmental changes

Creating an application with these characteristics is not easy, especially when using traditional means. Further, making such a paradigm shift work in the real world requires the talent, business knowledge, and technical expertise of the professional developer. In short, the professional developer needs a set of tools designed to meet these objectives, enabling a new level of parallel processing for *business* applications.

Therefore, what is needed is a flexible, sensible, and practical approach to parallel processing. The Software Pipelines technology was developed to be that approach, offering the professional developer a usable set of tools and capabilities to enable scalable processing for today's competitive business application environment.

What Do People Think about Parallel Processing?

As part of our research for this book, we wanted to find out what the software community thinks about parallel processing, so we conducted a statistical analysis of associated Web documents. Our analysis tool compares the usage of terms in Web documents, along with their frequency, in order to indicate the overall trend for a given subject. The results are intriguing; they confirm the importance of parallel processing as a solution for modern computing challenges.

To run the analysis, we based our search on the subject "software" and looked for references to related terms in the context of that subject. We started with the following terms:

- Multi-core
- Multi-threaded
- Parallel processing
- Parallel programming

We've included several charts in this section to show you the results. The first chart, Figure I.1, shows how often people use each term. As you can see, *parallel programming* is the most popular term, followed by *multi-core*, and then *parallel processing*. This gives us a good idea of how the software community talks about the subject.

To get a more detailed query, we cross-linked each term with the following attributes:

- Complex
- Hard
- Important
- Knowledge
- Useful

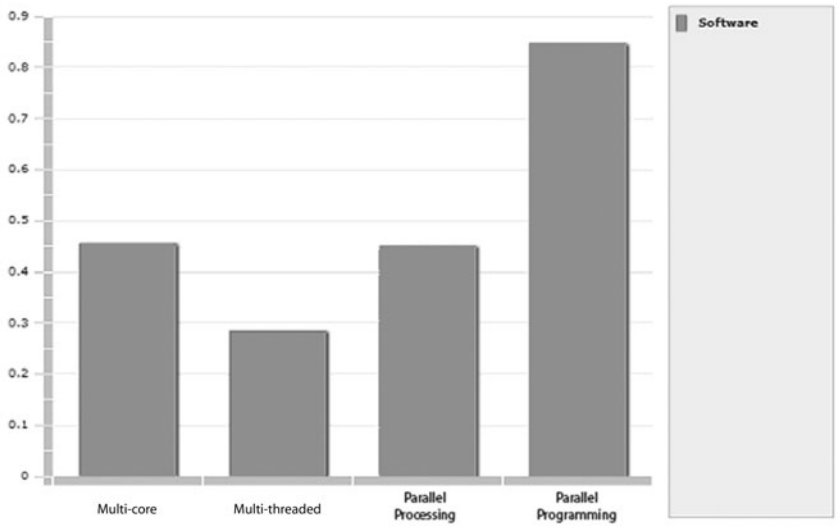


Figure I.1 Software query

In Figure I.2 you can see the relationship of each attribute to *parallel processing*. Parallel processing is perceived as "useful" and "important," its two strongest attributes.

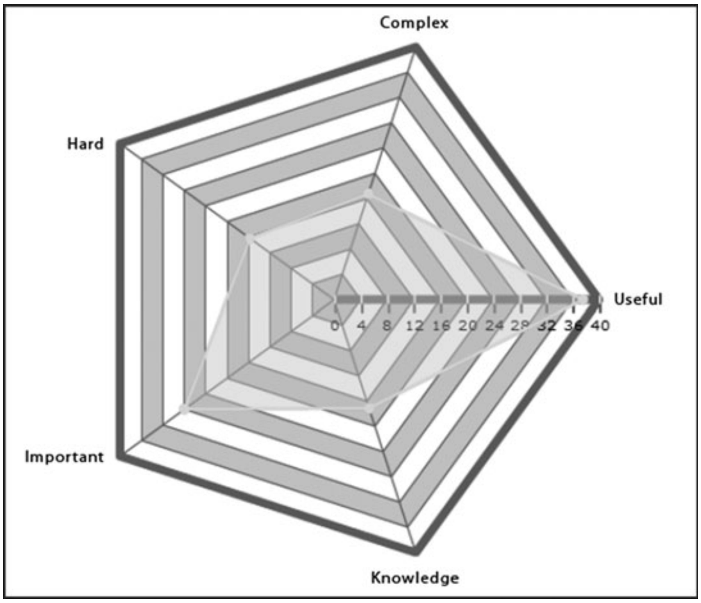


Figure I.2 Attributes for *parallel processing*

Figure I.3 shows *parallel programming* and its attributes. Parallel programming is definitely "important," but a high percentage of documents also mention that it is "hard." It's interesting that "knowledge" has a high rank, which is not surprising, given the difficulty of parallel programming and the general lack of experience with its techniques.

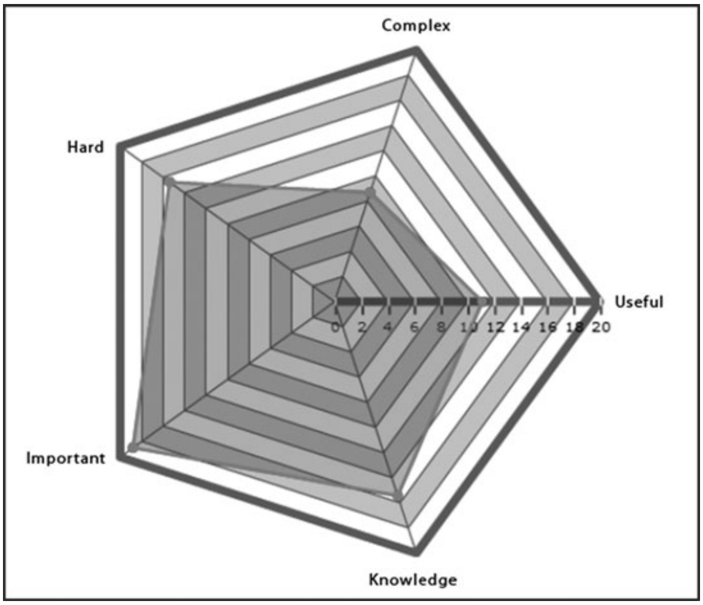


Figure I.3 Attributes for *parallel programming*

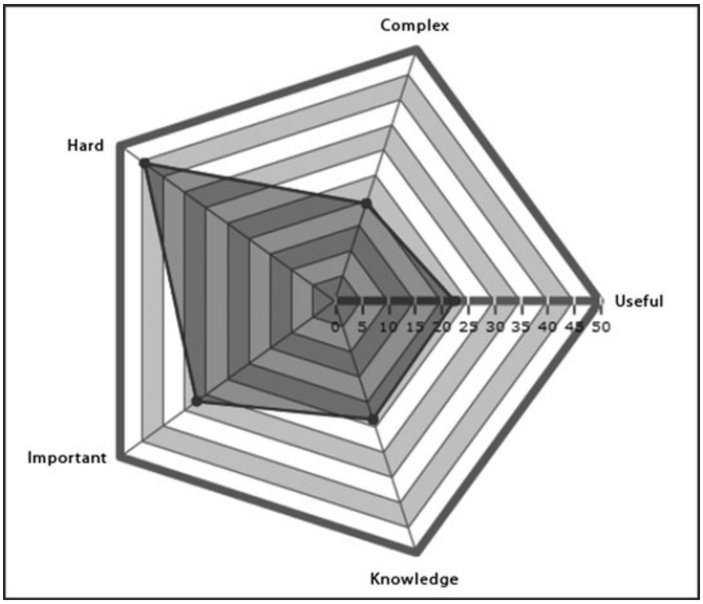


Figure I.4 Attributes for *multi-core*

Figures I.4 and I.5 show the attributes for *multi-core* and then *multi-threaded*. Both charts show responses similar to what we found for *parallel programming*.

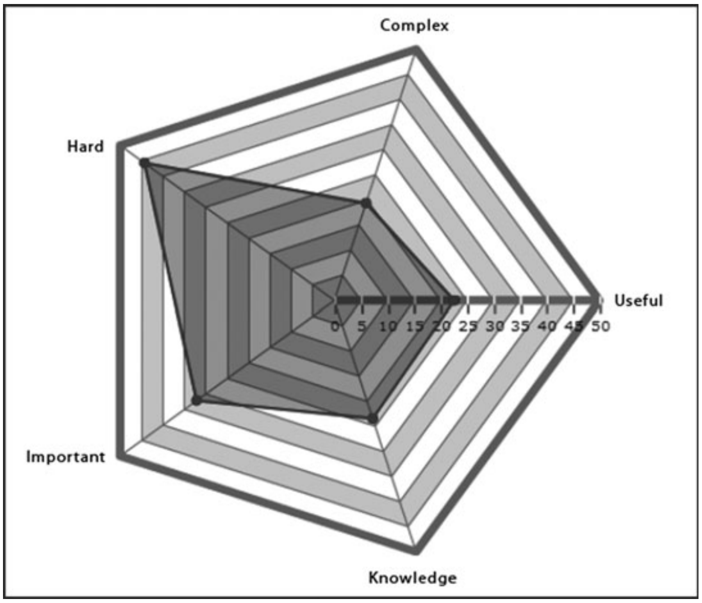


Figure I.5 Attributes for *multi-threaded*

In Figure I.6 we've included a chart with all terms and attributes to show the relative strength of each combination. You can see that parallel processing is considered "useful," and that parallel programming is both "important" and "hard."

What conclusion could you draw from all of this? It appears that people who talk about software are saying that parallel processing is important, but it's not easy. We're hoping we can help make it easier. The goal of Software Pipelines, and our goal in writing this book, is to provide a practical and useful set of techniques to address the challenge, and our intention is that you will find it helpful in your own application development and business management.

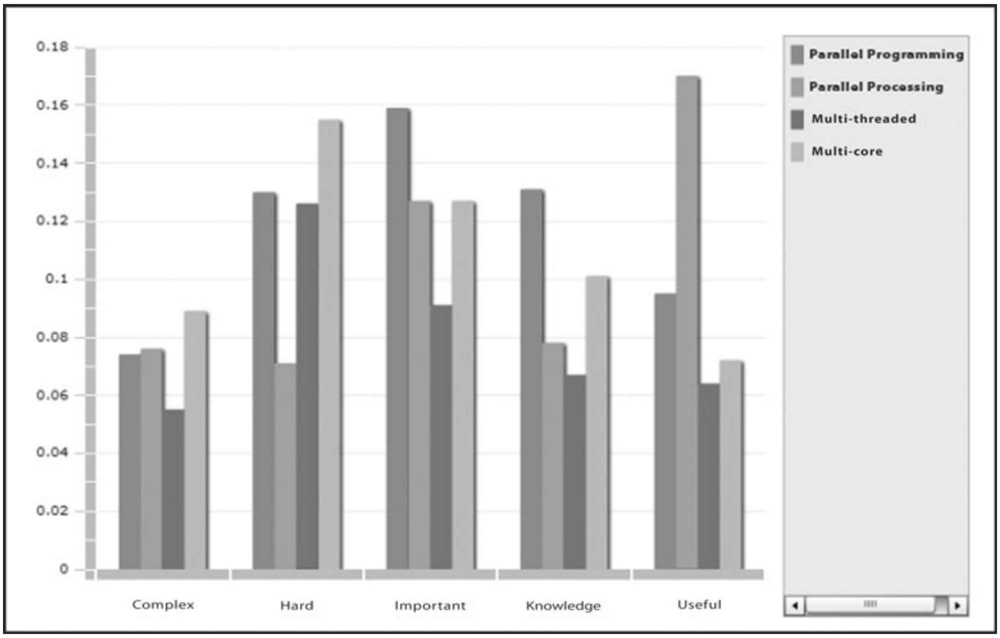
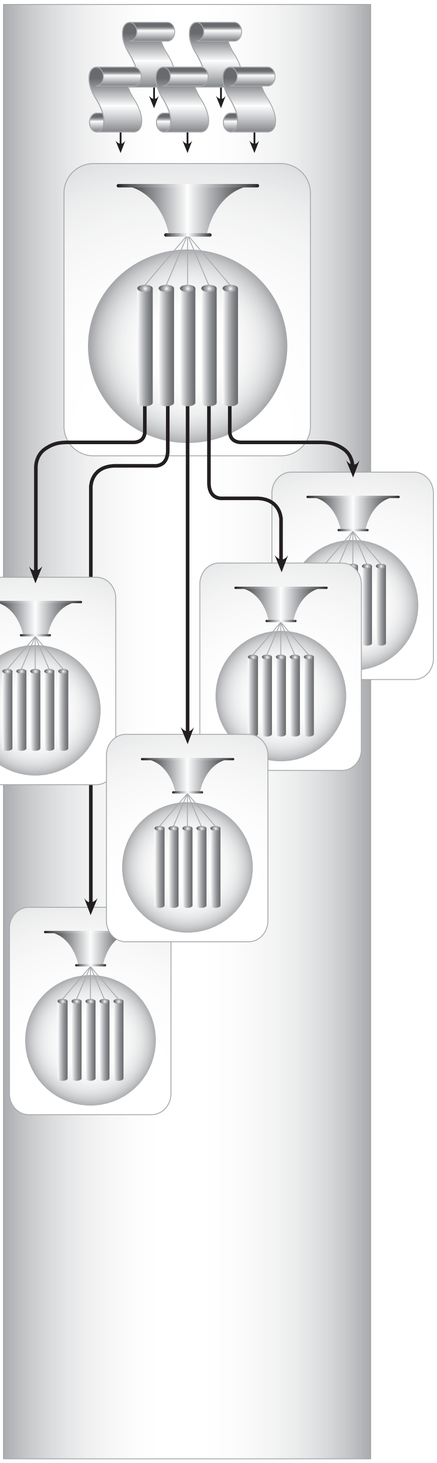


Figure I.6 Attributes for all terms

CHAPTER ONE

Parallel Computing and Business Applications



If you own, manage, or work with a critical business application, you're most likely dealing with performance problems. The application can't handle the ever-increasing data volume, it can't scale to meet new demand, or its performance is never good enough or fast enough. You need a higher level of performance; or even more daunting, you may need an order-of-magnitude increase so you can multiply the number of transactions your application can handle. In today's computing environment, there's really only one way to get there: Utilize a parallel architecture to run multiple tasks at the same time.

The fundamental concept of parallel architecture is this: Given a series of tasks to perform, divide those tasks into discrete elements, some or all of which can be processed at the same time on a set of computing resources. Figure 1.1 illustrates this process.

To do this, you have to break the application into a series of steps, some of which can run in parallel. However, that's really hard to do if you're working with existing business applications that do not lend themselves to such decomposition. Whether monolithic or object-oriented, most modern applications are tightly coupled, and that makes it hard to decompose a given process into steps.

Over the years, computer scientists have performed extensive research into parallel architecture and they've developed many techniques, but until now they focused on techniques that don't easily lend themselves to busi-

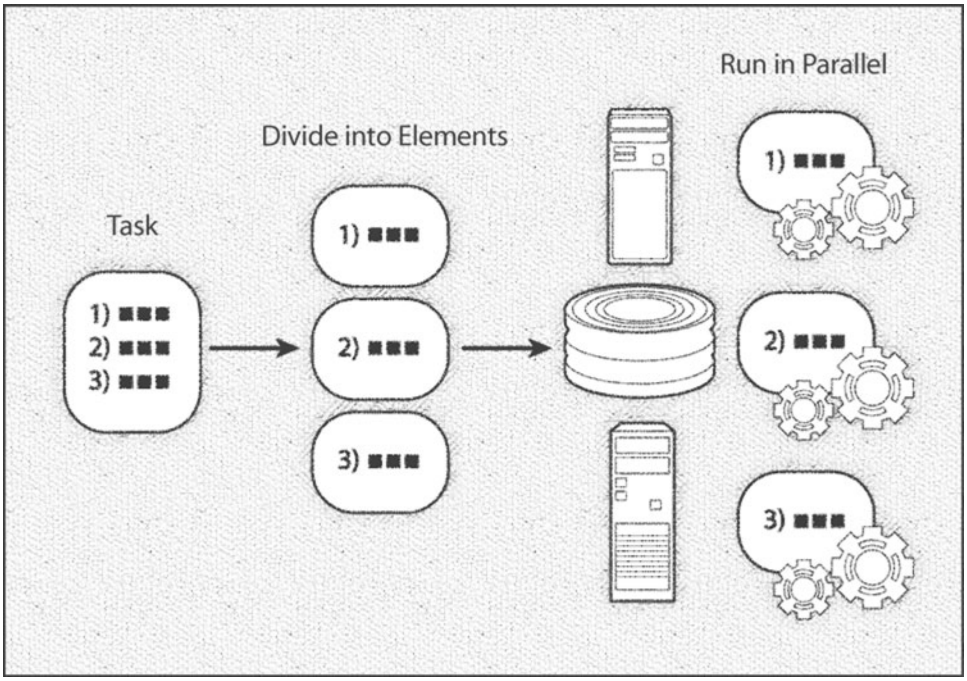


Figure 1.1 The fundamental concept of parallel architecture

ness systems. At the same time, demand for greater performance started overreaching the limit of most business applications, and the recent trend toward a service-oriented approach has made the challenge even greater. Parallel processing can fix the problem, but common existing techniques are either too complex to adapt to typical business transactions, or they don't even apply to the business arena.

Before we show you the solution, let's look at the existing techniques for parallel computing. The three main approaches are

- Mechanical solutions used at the operating system level, such as symmetric multiprocessing (SMP) and clustering
- Automated network routing, such as round-robin distribution of requests
- Software-controlled grid computing

Mechanical Solutions: Parallel Computing at the Operating System Level

Symmetric Multiprocessing

SMP automatically distributes application tasks onto multiple processors inside a single physical computer; the tasks share memory and other hardware resources. This approach is highly efficient and easy to implement, because you don't need specific, detailed knowledge of how SMP divides the workload.

Mechanical solutions such as SMP are very useful as generic one-size-fits-all techniques. To get the most out of SMP, however, you have to write applications with multi-threaded logic. This is a tricky job at best and is not, in general, the forte of most corporate IT developers. Plus, SMP is a black-box approach, which can make it very difficult to debug resource contention. For example, if you have shared software components and run into a problem, finding the cause of the bug may be very hard and time-consuming.

There's another drawback: Resource sharing between processors is tightly coupled and is not optimized for any particular application. This puts a lid on potential performance gain, and when you start scaling an application, shared resources will bottleneck at some point. So you might scale an application to eight processors with great results, but when you go to 16, you don't see any real gain in performance.

Clustering

In clustering, another widely used mechanical solution, separate physical computers share the workload of an application over a network. This technique provides some capabilities for automatic parallel processing and is often used for fail-over and redundancy.

Clustering techniques are automated and contain some inefficient functionality. If you're not using centralized resources, the system has to copy critical information (or in some cases, *all* information) from one node to another whenever a change in state occurs, which can become a serious bottleneck. As is the case with SMP, clustering is often effective up to a point—then adding hardware results in severely diminished returns.

Automated Network Routing: Parallel Computing by Predetermined Logic

In this technique you use some type of predetermined logic to divide application requests. One common approach is round-robin routing, where the system distributes requests evenly, one after the next, among a set of physical computers. Each computer provides exactly the same application functionality. A good example and use case for round-robin is a Web application, in which the system shunts each Web page request to one of several available processors.

Although this approach is useful for certain applications and can be useful as part of a Software Pipelines design, it is also very limited; the router has no logic for determining the best route for a given request, and all downstream processors perform identical tasks. Further, business applications often demand strict "order of processing" requirements, something that simple round-robin logic cannot accommodate.

Grid Computing: Parallel Computing by Distribution

All of the techniques covered so far have their uses, but you can't use them for massive scalability, and they don't work for transaction-based, message-oriented applications. You can scale them mechanically and automatically to a certain level, at which point the overhead of maintaining shared or redundant resources limits performance gains. If you need greater scalability, grid computing is a better choice.

In grid computing the system distributes discrete tasks across many machines in a network. Typical grid architecture includes a centralized task scheduler, which distributes and coordinates tasks with other computing facilities across the network.

Grid computing can deliver far higher throughput than the automated approaches described earlier, but it puts a significant burden on the developer. You must explicitly write the code for dividing tasks, for distributing tasks, and for reassembling the processed results.

Most importantly, grid computing is primarily designed to solve the "embarrassingly parallel" problem—long-running, computation-intensive processes as found in scientific or engineering applications. Grids are very beneficial for the typical use cases, such as modeling fluid dynamics, tracing the human genome, and complex financial analytics simulations. In each of these applications you divide a massive, long-running computation among multiple nodes. This divides the problem into smaller, similar tasks, which interact predictably with computational resources. However, this is not as useful for business applications, given their transactional nature, mixed workload requirements, and ever-changing volume demands.

Parallel Computing for Business Applications

Business applications are very different from engineering or scientific applications. They have the following traits:

- They process transactions.
- They process tasks with mixed workloads. Quite often you can't predict the size of each task, or what the processing requirements might be.
- The workload varies widely throughout a given time period. It might even change from day to day, or from one hour to the next.
- They often have requirements that defy the very concept of performing multiple tasks in parallel. For example, first in/first out (FIFO) transactions (which are very commonly used) must be done in an exact, ordered sequence.
- They almost always use a database or other centralized resource that bottlenecks and caps off transaction throughput.

Up to now, research on parallel computing concentrated mostly on mechanical solutions with limited scalability, or on grid-based scientific and engineering applications that lie outside the business domain. What we need is a new, simpler way to implement parallel computing for businesses. This new approach must support the following requirements:

- It must handle a wide variety of business application needs.
- It must provide ultimate scalability.
- It must maintain critical business requirements.
- It must be easy to implement by corporate IT developers.

In reality, there's no automagic answer for scaling business applications, because each organization has very different needs and requirements. The ultimate solution requires the right tools, architecture, and approach, and it must focus on business applications. But more importantly, the solution requires the expertise of the professional developer—the invaluable corporate resource who possesses both a full understanding of the technology and an intimate knowledge of the business domain.

The challenge of finding a business-oriented approach to parallel processing is answered by Software Pipelines. The architecture is highly scalable and flexible. It executes business services independent of location, and in such a way as to maximize throughput on available computing resources, while easily meeting a vast array of complex business application requirements.

The Solution: Software Pipelines

Imagine the ideal implementation for a business environment:

> You can divide any application process or portion of a process into discrete tasks or services and perform them anywhere in a given network (local or remote), in parallel with other tasks whenever possible. You can define the granularity of tasks to fit the specific needs of each application; the size can range from coarse-grained (such as Web services) down to fine-grained (such as class/method calls). In addition, the system optimizes resource utilization of all available facilities, because it dynamically shifts available resources to handle current demand.

The idea is simple, but the details are often complex, with a multitude of potential variations and design patterns. The solution is Software Pipelines architecture, which supports the following features and capabilities:

- You can decompose business processes into specific tasks, then execute them in parallel.
- It has virtually unlimited peer-to-peer scalability.
- It's easier on the developer because it provides an easy method for distributing and executing tasks in parallel—on one server, or across many servers.
- It's specifically designed for business applications, particularly those that use, or can use, SOA.
- It handles a high volume of transactions, both large and small, and is therefore ideal for mixed-workload processing.
- The design gives you control of throughput and task distribution, which means that you can maximize your computing resources.
- You can scale upward by using parallel architecture, while still guaranteeing the order of processing—a key business requirement in many mission-critical applications. This is a huge benefit over previous approaches.
- Because the architecture supports so many configurations and patterns, you can create a wide variety of application designs.

These features also allow you to take full advantage of today's multi-core processors, distributing transactions within and across servers at will.

The fundamental component in Software Pipelines is the pipeline itself, defined as follows:

> An execution facility for invoking the discrete tasks of a business process in an order-controlled manner. You can control the order by using priority, order of message input (for example, FIFO), or both.

Essentially, a pipeline is a control mechanism that receives and performs delegated tasks, with the option of then delegating tasks in turn to other pipelines in the system as required. This means you can use pipelines as building blocks to create an unlimited variety of configurations for accomplishing your specific application objectives.

You can group multiple pipelines into fully distributed, peer-to-peer pools; each pipeline processes a portion of an application or process. And because you can configure each pool to run on a specific local or remote server, the system can execute tasks anywhere on a network.

A pipeline can route tasks to other pipelines through a Pipeline Distributor, its companion component. The Pipeline Distributor is defined as follows:

> A virtual routing facility for distributing a given service request to the appropriate pipeline (which in turn executes the request) within a pipeline pool. The distributor is colocated with its pool of pipelines and effectively front-ends incoming service requests.
>
> The distributor routes service requests by evaluating message content. Routing is based on configuration rules, which you can easily modify without changing individual business services. You can route requests by using priority, order of message input (such as FIFO), or both.

In Figure 1.2 you can see how pipelines work with distributors. Requests go to the first distributor, which splits them off onto three pipelines. The second pipeline delegates a request to the third pipeline, and the third pipeline sends a request to another distributor, which in turn splits requests onto five pipelines.

By using pipeline and distributor components, you can build a fully distributed, multilevel series of interlinked pipelines—and achieve massive scalability through parallel processing.

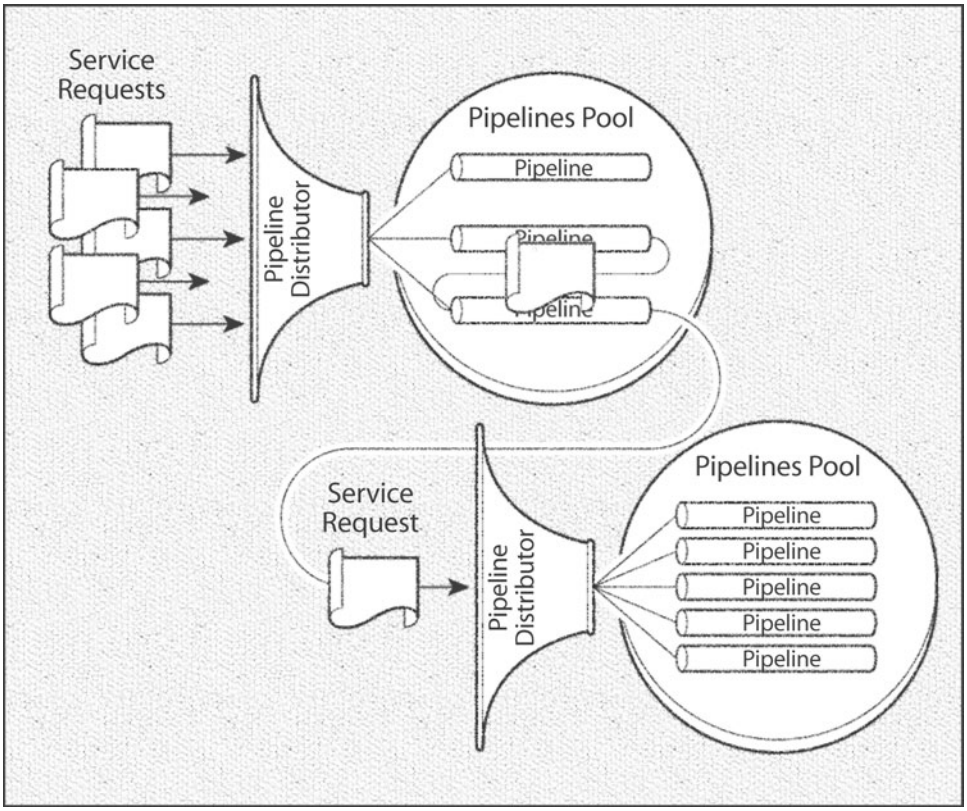


Figure 1.2 The Pipeline Distributor and its relationship to pipelines

Fluid Dynamics

It's easy to visualize Software Pipelines by comparing them to a network of hydraulic pipelines, which transport and direct the delivery of water or oil. Such a system has physical limitations:

- The input source delivers a particular maximum volume to downstream resources.
- Each downstream pipeline or receptacle (including subsidiary downstream pipelines and downstream destinations that process the delivered stream) must accommodate the input volume, or the entire system backs up.

In other words, all channels in the system must accommodate the maximum volume of flow. If they can't, the flow stops or slows down, or even breaks the system.

The same principles apply to Software Pipelines, but it's far easier to avoid bottlenecks. All you have to do is move some of the processing load to other pipelines. The example in the next section shows how to do this.

Software Pipelines Example

To show you how Software Pipelines work, we'll use a banking example. A large bank has a distributed network of ATMs, which access a centralized resource to process account transactions. Transaction volume is highly variable, response times are critical, and key business rules must be enforced—all of which make the bank's back-end application an ideal use case for parallel pipelines. We must apply the following business requirements:

- Make sure each transaction is performed by an authorized user.
- Make sure each transaction is valid. For example, if the transaction is a withdrawal, make sure the account has sufficient funds to handle the transaction.
- Guarantee that multiple transactions on each account are performed sequentially. The bank wants to prevent any customer from overdrawing his or her account by using near-simultaneous transactions. Therefore, FIFO order is mandatory for withdrawal transactions.

Before we cover pipeline design, let's take a look at the traditional design for a monolithic, tightly coupled, centralized software component. You can see the main flow for this design in Figure 1.3.

The simplicity of this design has several benefits:

- It's very easy to implement.
- All business rules are in a single set of code.
- Sequence of transactions is guaranteed.

However, this design forces every user transaction to wait for any previous transactions to complete. If the volume of transactions shoots up (as it does in peak periods) and the input flow outstrips the load capacity of this single component, a lot of customers end up waiting for their transactions to process. All too often, waiting customers mean lost customers—an intolerable condition for a successful bank.

To use Software Pipelines to solve this problem, we'll do a pipeline analysis. The first step is to divide the process into logical units of parallel work. We'll start by decomposing the steps required for processing. Figure 1.4 shows the steps of the ATM process.

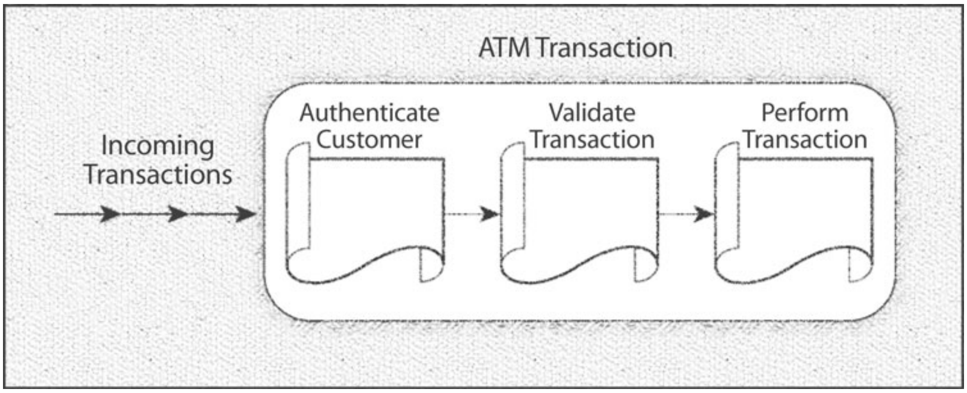


Figure 1.3 Traditional design for an ATM application

The steps are

- Authenticate the user (customer).
- Ensure the transaction is valid. For example, if the transaction is a withdrawal, make sure the account has sufficient funds to handle the transaction.
- Process the transaction and update the ATM daily record for the account.

Now that we understand the steps of the business process, we can identify the pipelines we'll use for parallel processing. To do this, we determine which portions of the business process can execute in parallel.

For the initial ATM design (Figure 1.5), it seems safe to authenticate users in a separate pipeline. This task performs its work in a separate system, and after it

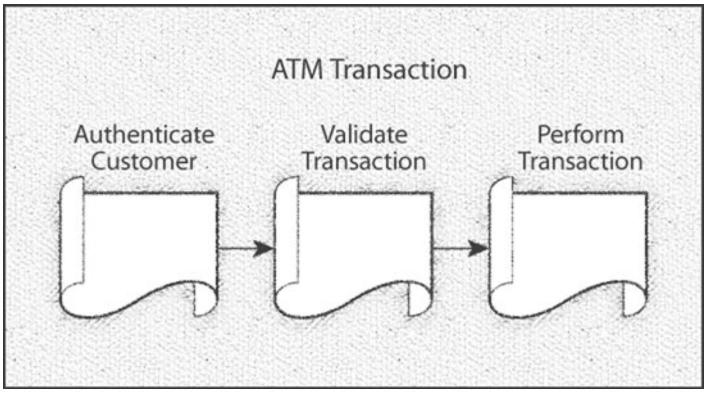


Figure 1.4 Steps in the ATM process

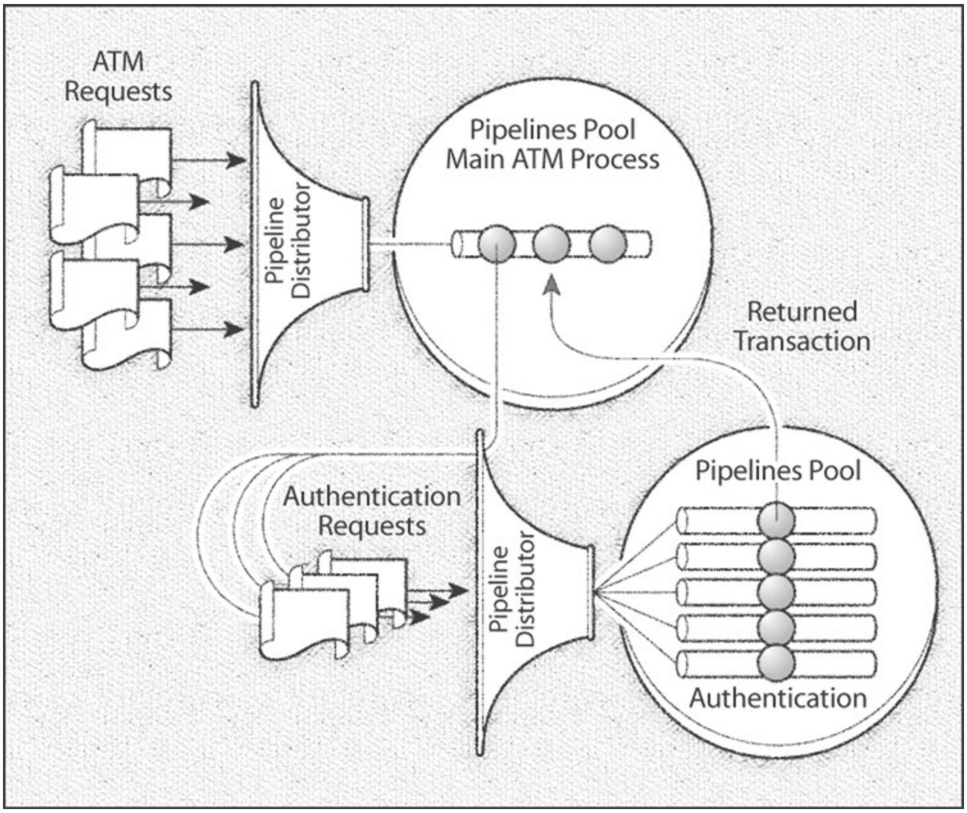


Figure 1.5 Initial pipeline design: Distribute the authentication step.

returns the authentication, the process can perform the next two steps. In fact, because we're not concerned with ordering at this stage, it's safe to use multiple pipelines for this single task. Our goal is simply to process as many authentications as we can per unit of time, regardless of order.

This design speeds up the process, but most of the work—updating the ATM accounts—is still a serial process. You'll still get bottlenecks, because the updating step is downstream from the authentication step. To improve performance by an order of magnitude, we'll analyze the process further. We want to find other places where the process can be optimized, while still enforcing the key business rules.

After authenticating a user, the next step is to validate the requested transaction. The application does this by evaluating the user's current account information. Business requirements allow us to perform multiple validations at the same time, as long as we don't process any two transactions for the same account at the same time or do them out of sequence. This is a FIFO requirement, a key bottleneck in parallel business applications. Our first configuration with the single pipeline guarantees compliance with this requirement; but we want to distribute the process, so we need a parallel solution that also supports the FIFO requirement.

The key to the solution is the use of multiple pipelines, as shown in Figure 1.6. We assign a segment of the incoming transactions to each of several pipe-

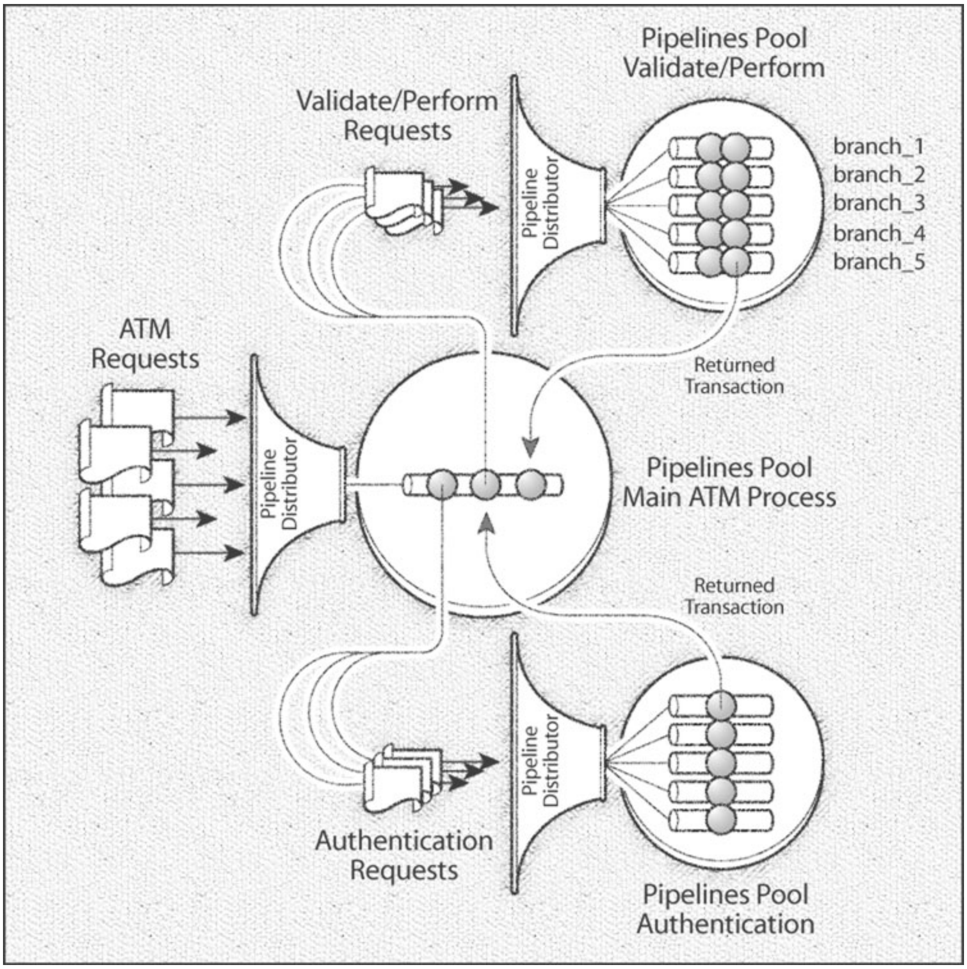


Figure 1.6 Distribute the validation step.

lines. Each pipeline maintains FIFO order, but we use content-based distribution to limit the pipeline's load to a small subset of the entire number of transactions.

To implement the new design, we create a pipeline for each branch of the bank (named branch_1 through branch_5), so that each pipeline controls a subset of accounts. We want the pipelines to handle delegated transactions sequentially, so we specify FIFO order for the new pipelines.

The Pipeline Distributor checks the branch ID in each transaction (which is an example of content-based distribution), then sends the transaction to the matching pipeline.

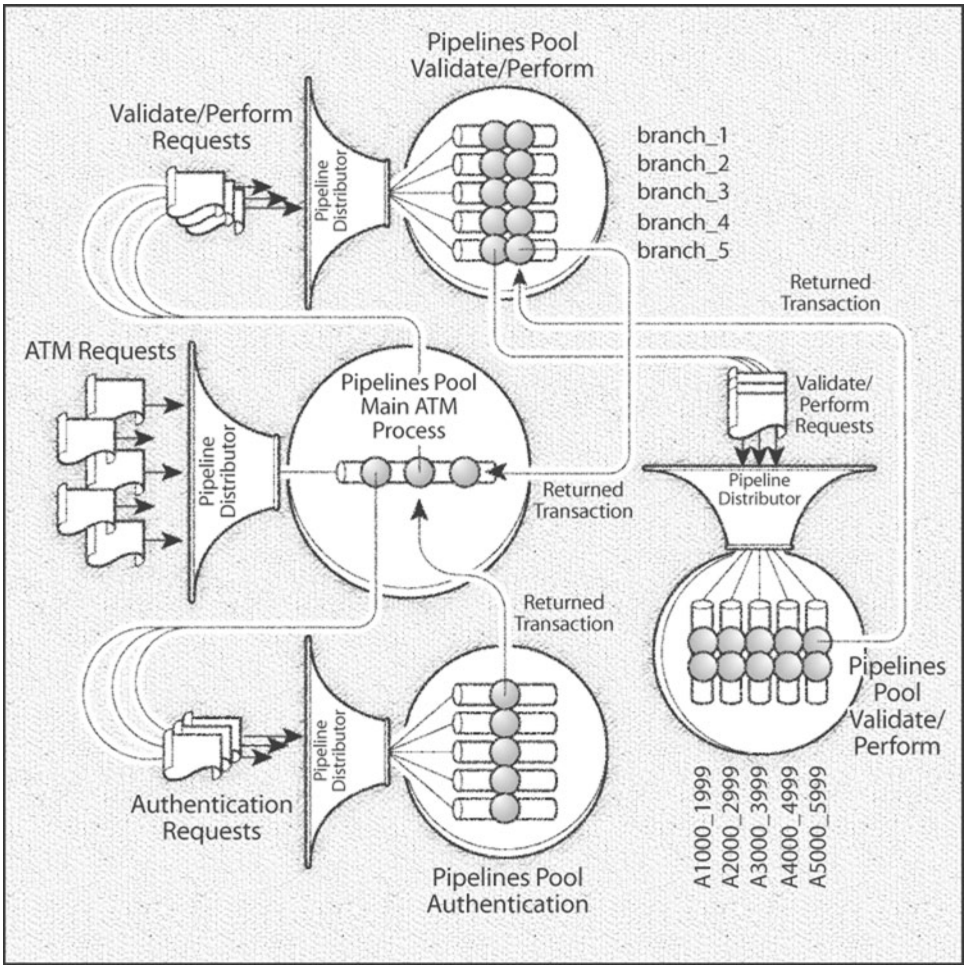


Figure 1.7 Scale the application further by adding downstream pipelines.

Now, by processing many branches in parallel, the system completes many more transactions per unit of time.

You can use this approach to scale the application up even further, as shown in Figure 1.7. Let's assume the bank has a very large branch with more than 100,000 accounts. The branch's peak transaction volume overloads the previous pipeline configuration, so we create additional downstream pipelines. The distributor divides the transactions by using a range of account numbers (A1000_1999, A2000_2999, etc.).

At this point, whenever the bank's business increases, it's a simple matter to build additional pipeline structures to accommodate the increased volume.

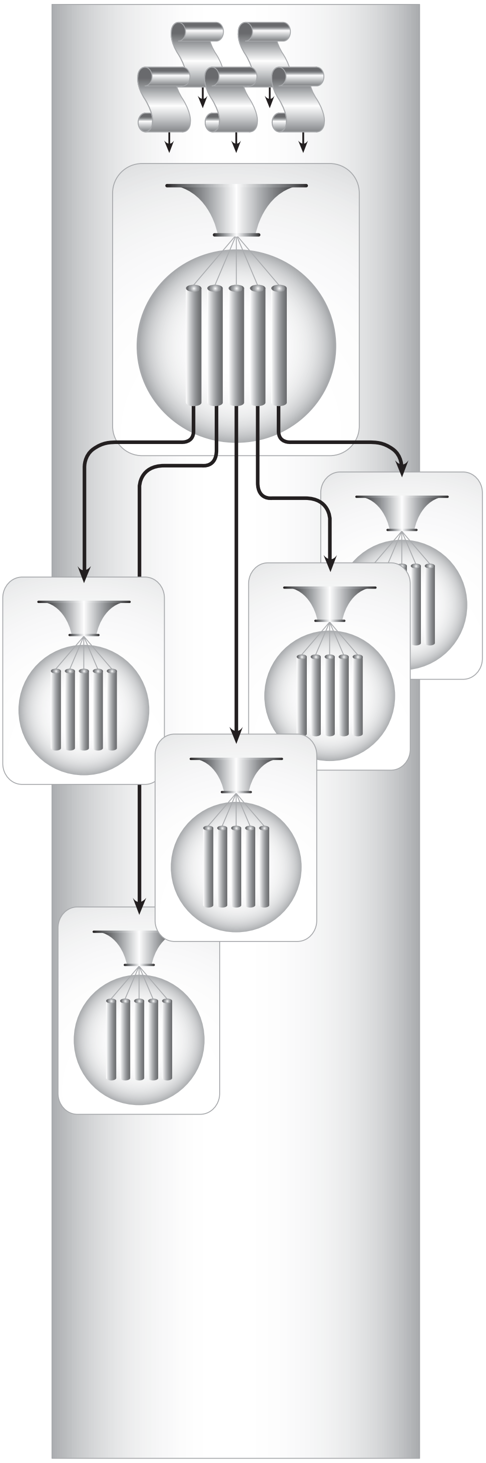
To sum up, the ATM example illustrates how you can use Software Pipelines to increase process performance by an order of magnitude. It's a simple example, but the basic principles can be used in many other applications.

Summary

Many of today's organizations are facing a hard reality: In order to meet current demand, their business applications must increase performance by an order of magnitude. And over time the problem only gets more severe—the business sector depends more and more on data, so demand is going to accelerate, not slow down.

In order to meet such daunting requirements, the capability of performing multiple tasks in parallel becomes vital. We have many solutions for improving application performance, but we've never had the technology to create a parallel software environment specifically for business applications.

Software Pipelines architecture answers this challenge at every point. It was designed specifically for business, you can easily scale your application to any size, you can maximize your resources, and best of all, you can do all this and still maintain critical business transaction and integrity requirements.



Index