Parallel Processing Techniques for Electronic Commerce Systems

by

Haonan Zhang

A Thesis Submitted to the Faculty of Graduate Studies of
The University of Manitoba
in Partial Fulfillment of the Requirements for the Degree of

Master of Science

Department of Computer Science
The University of Manitoba
Winnipeg, Manitoba, Canada

THE UNIVERSITY OF MANITOBA

FACULTY OF GRADUATE STUDIES

*****

COPYRIGHT PERMISSION

Parallel Processing Techniques for Electronic Commerce Systems

BY

Haonan Zhang

A Thesis/Practicum submitted to the Faculty of Graduate Studies of The University of Manitoba in partial fulfillment of the requirement of the degree

OF

Master of Science

Abstract

With the increased confidence in, and the use of the Internet and the World Wide Web, the number of electronic commerce (e-commerce) transactions is growing rapidly. To increase throughput and reduce response time for e-commerce transaction requests, parallel processing techniques can be used. The parallel processing techniques addressed in this thesis focus on parallelizing on-line transaction processing (OLTP). The aims of this thesis are to:

1. characterize e-commerce transactions with a view to find those aspects that would benefit from parallel processing;
2. evaluate current parallel processing techniques to determine those techniques that can be applied to e-commerce transactions; and
3. provide a reliable, flexible, and scalable design of an e-commerce transaction processing system that uses parallel processing techniques to deal with data intensive transactions and to process those transactions faster.

My contributions of this thesis are as follows. First, I addressed the opportunities that could apply parallel processing techniques in the business logic tier and the database tier in multi-tier e-commerce systems. Second, I developed several generic parallel algorithms for e-commerce systems. Third, I implemented a prototype e-commerce transaction processing system that used those parallel processing techniques.

Acknowledgements

First, I would like to express my deepest appreciation to my supervisor, Dr. Sylvanus A. Ehikioya, for sharing his knowledge and providing me with many helpful comments. Without his suggestions and constant support, this thesis would not be possible. A special thank goes to my co-supervisor, Dr. Carson K. Leung, who provided me with detailed feedback and insightful advice on my thesis and my oral examination. Without his constant guidance, I could not have finished this thesis.

I would also like to thank my thesis examination committee, Dr. Carson K. Leung, Dr. Vojislav B. Misic and Dr. Robert D. McLeod, for evaluating this thesis, and Dr. Neil Arnason for chairing my thesis defense.

Many thanks go to Dr. Neil Arnason and Dr. Peter Graham for their advice, suggestions, and help during my research.

Finally, I would like to thank my wife, my son, and my parents for their endless love, emotional support, and continued encouragement. I dedicate this thesis to my wife, my son, and my parents.

Table of Contents

List of Tables

List of Figures

Chapter 1 Introduction

Electronic commerce (e-commerce) is the use of computers and telecommunication technologies to share business information, maintain business relationships, and conduct business transactions. Its genesis is traced back to the Electronic Data Interchange (EDI) activity in the 1960's. EDI refers to the set of activities that are related to the electronic facilitation of the transactions between venders and buyers (purchase orders, waybills, manifests and schedules). Currently, e-commerce depends mostly on the Internet as the underlying platform. Business transactions are events that serve the mission of a business. A transaction provides the primary means by which a business interacts with its suppliers, customers, partners, employees, and the government. Transactions are significant because they capture and/or create data about and for businesses [WB97]. Examples of transactions include purchases, orders, sales, reservations, shipments, invoices, and payment processing.

E-commerce began in the 1990's and was largely driven by the invention of the World Wide Web. The adoption of e-commerce has led to many new business models. The most important models of e-commerce are business-to-business (B2B), business-to-consumer (B2C), and consumer-to-consumer (C2C). *C2C e-commerce* refers to the use of the Internet by consumers to provide goods and information to other consumers; it offers an effective way to exchange goods and information between consumers. *B2C e-commerce* mostly refers to the use of the Internet by a business to provide goods and services to customers; it offers consumers a fast and efficient way to access various products and services from retailers all over the world

without leaving home. *B2B e-commerce* refers to the use of the Internet between businesses to order products, receive invoices, and make payments; it reduces production costs, accelerates ordering processes, and improves inventory management. By exploiting efficiency, economy, and speed of the Internet, e-commerce simplifies and reduces the cost of processes involved in business transactions. The parallel processing techniques proposed in this thesis can be easily applied to B2C and B2B e-commerce systems.

Transactions and transaction systems were introduced many years ago [GR92], and are closely related to databases. A *database* is a collection of related data. A *transaction* can be viewed as a set of recorded state changes in the records of a database (induced by some specific business processes); it commonly exhibits four properties: atomicity, consistency, isolation, and durability (ACID) [GR92]. *Atomicity* means that a transaction is either all done or not done, but it cannot be partly executed. *Consistency* means a transaction should not violate the integrity of data. *Isolation* means that each transaction is processed separately and is not interfered by the processing of other transactions. *Durability* means that once a transaction is completed successfully, its result is stored in the database and will not be affected by failures [GR92]. My prototype e-commerce system implemented in this thesis maintains the ACID properties of transactions.

The transaction processing system is a critical component of any e-commerce system that must manage the transactions between thousands of concurrent clients and back-end systems. Traditional sequential transaction processing techniques may fail to

meet e-commerce system requirements such as high throughput and high performance. To overcome the limitations of sequential processing, such as poor performance and poor throughput, parallel processing[1] techniques could be used to deal with the demands of e-commerce transactions. In this thesis, I focus on using parallel processing techniques to improve the performance and throughput of e-commerce transaction processing systems.

E-commerce transactions include both on-line analytical processing (OLAP) and on-line transaction processing (OLTP) transactions. The OLTP of e-commerce transactions manages data and processes orders. The OLAP of e-commerce transactions analyzes historical data from OLTP e-commerce systems and provides reports in support of management decisions. In general, OLTP deals with the atomic level of data, needs fast responses, and normally follows standard procedures and well-defined workflows. OLAP focuses on providing analysis capability to management and typically deals with billions or even trillions of transaction records spanning periods from several days to decades.

Although some parallel implementations of OLAP transactions [GC97, GC99] and many parallel implementations of databases [DG92, Fur04, RHN05, WTCY94] have been proposed, to the best of my knowledge, parallel implementations of OLTP transactions specific to e-commerce are rarely existed.

---

[1] *Parallel processing refers to the use of a collection of processing elements for cooperatively solving problem fast.*

However, I observe that there are many opportunities to apply parallel processing techniques to OLTP in an e-commerce system. Hence, in this thesis, I focus on proposing parallel processing techniques for e-commerce OLTP transactions. To elaborate, typical architecture of an e-commerce system is three-tiered client/server architecture consisting of the GUI tier, the business logic tier, and the database tier. In the business logic tier, one could use a single processor server using multithreading or a multi-processor server to process many transactions concurrently. In the database tier, the database could be fragmented horizontally[2] and distributed to multiple database servers or simply replicated over a number of servers. When a user transaction is processed, the transaction could be processed in parts across multiple database servers simultaneously. For instance, when a user wants to find a particular product, he/she could submit the search criteria to the server. The server in the business logic tier could then transform the search into a query. The query could then be forwarded to different systems for processing. For each system, the query could be executed against parallel databases in the database tier.

E-commerce systems are complex. A single transaction may include several logical steps. Some of these steps have dependencies between them, while others do not. The steps that have no dependencies can be executed concurrently.

As described above, there are opportunities to apply parallel processing techniques when an e-commerce system processes e-commerce transactions.

---

[2] *Horizontal fragmentation divides a database relation "horizontally" into multiple non-overlapping sub-relations having the number of columns but fewer rows. In other words, each horizontal fragment is a collection of rows of the original database relation.*

Identifying what to parallelize and which parallel techniques to use, and incorporating them into a flexible and scalable design for a parallel e-commerce system is the focus of this thesis. Thus, the aims of this thesis are to:

1. characterize e-commerce transactions with a view to find those aspects that would benefit from parallel processing;
2. evaluate current parallel processing techniques to determine those techniques that can be applied to e-commerce transactions; and
3. provide a reliable, flexible, and scalable design of an e-commerce transaction processing system that uses parallel processing techniques to deal with data intensive transactions and process those transactions faster.

1.1 Scope and Contributions

This thesis describes an e-commerce system design that is three-tiered architecture (the GUI tier, the business logic tier, and the database tier) system using different parallel processing techniques. The design helps e-commerce systems that deal with large dataset to get faster response. This thesis also describes an implementation of a prototype e-commerce transaction processing system that is developed to demonstrate the feasibility of the design. I compare my implementation with the implementation of an e-commerce system that uses traditional *sequential* processing techniques, and highlight the performance improvement brought by my design.

Although the research described in this thesis is in the context of e-commerce systems, the ideas presented in the parallel e-commerce system design can be

extended to other multi-tier data intensive applications. My contributions of this thesis are as follows:

- I address those opportunities that could apply parallel processing techniques in a multi-tier e-commerce system. Some of the opportunities were not fully explored in pervious researches, namely the opportunities in the workflow in the business logic tier. In this thesis, we process not only data but also independent workflows in the business logic tier in parallel.
- I develop several generic parallel algorithms that could deal with typical e-commerce system operations.
- I design and implement a prototype e-commerce system that applies those parallel processing techniques.

All these contributions help us to provide a reliable and scalable platform for e-commerce systems.

1.2 Organization

The remaining of this thesis is organized as follows: Chapter 2 provides background information on e-commerce transactions and transaction systems, and on parallel processing techniques. In addition, the chapter also introduces the middleware framework used in this thesis. Chapter 3 analyzes some typical e-commerce transactions that can benefit from applying parallel processing techniques and gives the algorithms that could be applied to these transactions. Chapter 4 describes an implementation of the prototype e-commerce transaction processing system.

Chapter 5 describes and analyzes the performance result of the prototype e-commerce system. Chapter 6 provides conclusions and discusses possible future work.

Chapter 2 Background Literature

An e-commerce system involves different technologies, including databases and workflows. Moreover, features, such as real-time behavior, dynamism, concurrency and distribution, characterize e-commerce transactions [Ehi97]. Furthermore, formal design methodologies are desirable to design reliable e-commerce systems [Ehi01]. To appreciate the research problem presented in this thesis, one better fully understands e-commerce related issues and technologies such as databases, workflows, e-commerce transaction characteristics, and the e-commerce formal design. Moreover, to apply parallel techniques, a thorough understanding of parallel processing from both the hardware and the software perspectives is essential.

2.1 E-commerce Transactions and Transaction Systems

E-commerce transactions maintain the ACID properties, and are characterized by features such as real-time behavior, dynamism, concurrency and distribution. An e-commerce system is inherently distributed in nature. A single e-commerce transaction may consist of interactions across multiple departments of an organization, may even access data from multiple external organizations [Ehi01, WVD98], and may possibly involve concurrent operations. Because of the concurrent operations, data (such as inventory data) can also be changed dynamically in an e-commerce system as in some other types of systems. Such dynamic changes could influence the processing of transactions. For example, when several customers check the availability of a certain item, which is the only one left in an e-commerce site, the site might report

that the item is still available. However, when these customers request the item concurrently, only the first order will be processed successfully. The inventory data will be updated when the first order is committed. Then, the item will become unavailable. Orders from other customers will fail because the inventory data has been updated. An e-commerce system also requires real-time responses. The result from the response will determine further steps of the process. For example, when a customer provides his/her credit card information to check out the contents of his/her shopping cart (i.e., decides to finalize the purchase), the system will forward the credit card information to a credit card company for processing. A real-time response (within an acceptable time-period) from the credit card company is required to continue the process. The result from the credit card company (approval or disapproval) will determine the further steps of the process. If the e-commerce system does not get any response from the credit card company in the required time, an exception process will be executed. These characteristics of e-commerce transactions (real-time behavior, dynamism, concurrency and distribution) should be considered and reflected in the design of the e-commerce system.

An e-commerce transaction must also be interoperable with legacy systems and infrastructures. Most e-commerce systems are developed on top of existing systems. Therefore, to preserve the investment in legacy systems, e-commerce transactions must integrate with these legacy systems.

2.2 Parallel Processing

In parallel processing, a program is divided into multiple fragments among multiple processors, and these fragments are executed simultaneously. The objective of using parallel processing is to speed up[3] and/or scale up[4] the processing of transactions. Parallel processing architecture can be categorized into three groups according to the particular system resources that are shared: shared memory systems, shared disk systems, and shared nothing systems [TV02, Zom96].

A *shared memory system* has multiple processors, a global shared memory, shared disks and shared input/output devices. All processors have full access to the shared memory through a common bus. Communication between processors occurs via the shared memory. A shared memory system has its merits, such as fast memory access and easy administration. However, it suffers from limited scalability since the shared memory can become a bottleneck.

A *shared disk system* has multiple processors, shared disks and other input/output devices. Each processor, however, has its own memory and cache. All processors have full access to all the shared disks and other shared resources through a shared interconnection network. The shared disk system is designed to have high availability. All the data should be accessible even if one node dies. Since any node can access all the data in a shared disk system, global concurrency control is required. Therefore, a

---

[3] *An objective of using parallel processing is to decrease the time taken for processing transactions when the number of processors increases.*

[4] *Another objective of using parallel processing is to sustain the response time per transaction when the numbers of processors and of transactions increase.*

shared disk system requires explicit message exchange for system wide synchronization. The performance of a shared disk system is reduced because of the need for inter-node synchronization.

A *shared nothing system* has multiple processors where each processor has its own memory, cache, disks and other input/output devices. Nodes are connected via an interconnection network. To access the memory of another node, the user must pass a request through the interconnection network. A weakness of a shared nothing system is its high communication latency. However, such a system has several advantages, including improved scalability (as a shared nothing system can be easily extended to hundreds or even thousands of nodes) and high availability. Hence, *in this thesis, I use a shared nothing system.*

To leverage the benefits of different parallel processing system models and to overcome their limitations, several combinations of different parallel processing systems have also been proposed. For example, several shared disk systems can be connected as part of a larger shared nothing system [BFV96].

2.3 Parallel Databases

The parallel processing architecture introduced in Section 2.2 allows multiple computers to share data, software, and/or input & output devices. To take advantage of such architecture, developers designed parallel databases in which multiple instances of a single physical database can run on multiple computers. Comparing

with traditional databases, a parallel database can provide higher performance, higher availability, more flexibility, and better ability to provide high throughput.

In a parallel database, if the execution of a transaction is decomposed into multiple processing steps, a set of corresponding database requests can be executed in parallel. There are three kinds of database parallelism [DG92, RNS96, WTCY94]:

1. *Inter-query parallelism* occurs when two or more independent database requests are executed at the same time, thereby increasing overall throughput in the database system.
2. *Inter-operation parallelism* occurs when two or more operations within a single database request are executed simultaneously. For example, there might be SCAN and JOIN operations in a query. In this case, SCAN and JOIN operations could be executed in parallel.
3. *Intra-operation parallelism* is obtained by executing the same operation on different data fragments[5]. For example, a database can be horizontally fragmented into several fragments, and these fragments are distributed across several database servers. There might be a SCAN operation in a query. The SCAN can be sent to these database servers and be executed concurrently in parts. In this thesis, I use intra-operation parallelism.

[5]*In parallel systems, relations can be divided into smaller fragments and stored at different sites. Each fragment is a logical unit of data.*

These three forms of parallelism are only theoretical models. In reality, the parallelism depends largely on the implementation. However, realizing these parallelisms is a complicated task.

Parallel database system architecture can use the parallel processing architecture discussed in Section 2.2. For example, in the shared memory architecture, data and messages are exchanged through the shared memory. Advantages of using such architecture include easier synchronization, low overhead for load balancing. However, this architecture does not scale very well. On the other hand, the shared nothing architecture is more popular due to its scalability and high reliability. It can be easily built from existing machines using an interconnection network [MD97].

In parallel database systems, the database can be physically distributed to different database servers by fragmenting and replicating the data. Fragmentation divides relations into horizontal or vertical partitions. Fragmentation is desirable because it reduces the size of relations involved in user queries. Each of the fragments may also be replicated based on the user access patterns [KPAG99]. Partitioning and distributing the data helps improve the performance of a parallel database system because it allows each data section to be processed individually. Similarly, data replication helps improve the performance by providing an opportunity for each individual data set to be processed individually.

In addition to the static data placement method described in the previous paragraph, some dynamic data placement methods also exist. A dynamic data allocation method identifies data fragments that are frequently accessed and moves

data from those nodes to other nodes. So, data in different nodes can be accessed evenly throughout the system. Dynamic data allocation will prevent a certain node to be overburdened, and thus it will reduce chances of system failure [RM95]. When data are fragmented and distributed, a database query specified in global relations is transformed into one that operates on fragments. Data localization techniques are used for the transformation. In data localization, opportunities for parallel execution are identified and unnecessary work is eliminated. Localization involves changing the order of operations, determining the execution sites for various distributed operations, and identifying the best-distributed execution algorithms for distributed operations [MOW97]. When data are distributed, the high communication overhead on complex queries involving joins can lead to serious performance overhead. Thus, many different query optimization solutions have been proposed [Fur04, RHN05, ZCRS05]. Parallel hash-join algorithms such as hybrid hash join or GRACE [YM98] consider dynamic partitioning and allocation intervening relation fragments into processors for fast join processing. These strategies allocate a hash range to each processor, which builds a hash table and redirects hared relation fragments to the corresponding processor. Placement dependency [LCK96, HL00] is another successful relation fragment collocation strategy.

2.4 Workflow

According to Lawrence [Law97], a *workflow* is "the automation of a business process, in whole or part, during which documents, information or tasks are passed from one participant (human or machine) to another for action, according to a set of procedural

rules". It is a flow consisting of logical steps of work that can be executed automatically or manually.

Many e-commerce business processes can be modeled as some small sequences of logical steps. These steps are executed in an order according to the data and control dependencies among them. These processes can be modeled as workflows. It is easy to automate, coordinate, and control these logical activities using workflows. Logical steps that have no dependencies among them can be processed in parallel.

The following are some examples of related work in the area of workflow in e-commerce [AFH+99, CDTA99, DDA+98, SABS02]. Alonso et al. [AFH+99] proposed the WISE (Workflow-based Internet SErvices) project, which provides a coherent platform for an enterprise network that can be seamlessly integrated with e-commerce systems. They used workflow to design and document the system. Dogac et al. [CDTA99, DDA+98] introduced architecture that integrates workflow technology in e-commerce. To deal with the difficulties, such as concurrency control and recovery control, that people face when integrating workflow in e-commerce systems, Schuldt et al. [SABS02] proposed a unified model for concurrency control and recovery for workflow. The model provides a complete framework for developing applications using workflow. All these projects use workflow in e-commerce systems, but none of them exploits the power of parallel processing.

2.5 E-commerce Design Issues

In e-commerce systems, transactions should be processed reliably and correctly in real time. Only well-designed, robust and fail-safe e-commerce systems will attain all the potential benefits of migrating and deploying business transactions online [Ehi01].

Several different formal e-commerce designs have been proposed, including (a) a formal e-commerce framework using Z [Spi92] & the Z/EVES tool [Saa99], and (b) Petri nets [BLWW95]. These proposed formal designs of e-commerce systems provide generic models of e-commerce systems. An e-commerce system developed using these formal models should provide a correct and robust transaction processing environment for e-commerce.

2.6 Limitations of Related Work

An e-commerce system deals with large number of business transactions. To obtain high performance and high throughput, a flexible and scalable design for an e-commerce transaction processing system that applies parallel processing techniques is necessary. To provide such a design, a careful analysis of e-commerce system is essential. Such analysis must include two different parts: (a) characterizing e-commerce transactions to isolate those operations or functions that would benefit from parallel processing, and (b) evaluating existing parallel processing techniques to determine those suitable to e-commerce transactions.

The sequential processing model is inadequate for an e-commerce system. Most of the related work on parallel databases discussed in Section 2.3 use parallel processing

techniques to obtain high performance and high throughput. However, they only partially exploit the opportunities that could provide benefits by using parallel processing techniques for e-commerce transactions. They focus on parallelism in the database only. In my proposed solution, in addition to the parallelism in database, I also apply parallel processing techniques in workflows. Thus, my proposed solution yields higher performance and higher throughput in the execution of e-commerce transactions.

2.7 CORBA Introduction

In this thesis, I extensively used the object-based model CORBA (Common Object Request Broker Architecture) [Obj98], which is an open, vendor-independent architecture and infrastructure that computer applications use to work together over networks.

An important characteristic of large networks, such as the Internet and corporate intranets, is that they are heterogeneous. A network might consist of mainframes, UNIX workstations, and Windows systems. The networks and protocols that connect these systems might also be diverse: FDDI, ATM, TCP/IP, and Novell Netware. In the ideal world, heterogeneity enables us to use the best combination of hardware and software for each portion of an enterprise. However, dealing with heterogeneity in a distributed environment has never been easy. In particular, the development of software applications and components that make efficient use of heterogeneous networked systems is very challenging. To solve this problem, the Object Management Group (OMG) defined CORBA.

Before introducing CORBA, I will provide a brief high-level overview of Object Management Architecture (OMA). The OMA is the framework within which all OMG adopted technology fits. It provides two fundamental models (Core Object Model and Reference Model), on which CORBA is based. The *Core Object Model* defines how objects are described and distributed across a heterogeneous environment, while the *Reference Model* describes how those objects interact. New technology specifications must fit into the Core Object Model and the Reference Model in order to be acceptable as OMG specifications.

In the OMA *Core Object Model*, an object is encapsulated in an entity with a distinct identity. The services of that object can only be accessed through well-defined interfaces. Clients issue requests to the object to perform services on their behalf. The requests do not rely on the location of objects and the language used for the implementation of objects.

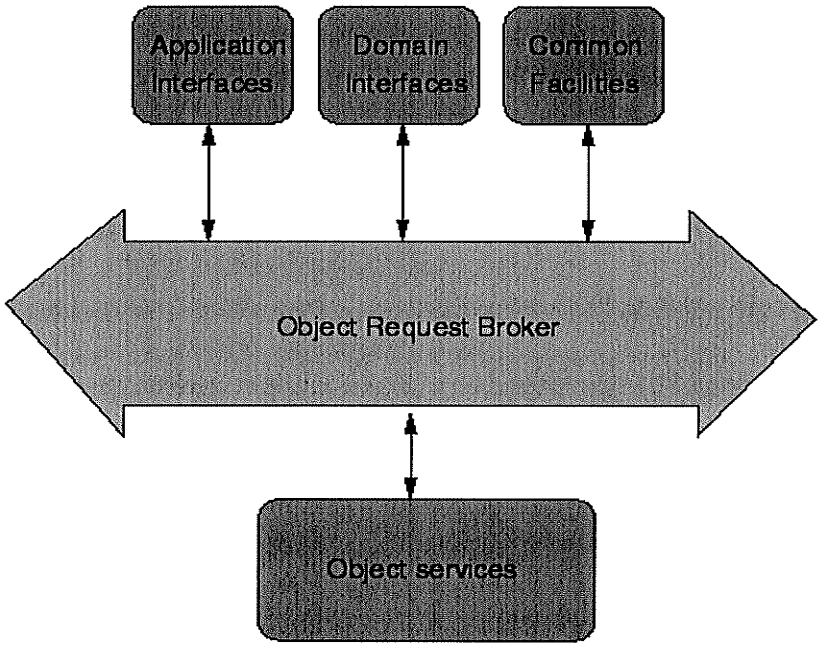


Figure 2-1 OMA Reference Model

The OMA *Reference Model* is an architectural framework that identifies and characterizes the interface and components that compose the OMA. Figure 2-1 shows the component of OMA reference model. The Object Request Broker (ORB) acts as a message bus between objects located on machines of a network, which enables clients and objects to communicate in a distributed environment. Four categories of object interfaces that can utilize the ORB component are:

- *Application Interfaces*, which are interfaces that developed specifically for a given application. These interfaces are not standardized.
- *Domain Interfaces*, which are sets of interfaces that are application domain-specific, such as telecommunications, Internet, business objects, manufacturing, and health care.
- *Common Facilities*, which are sets of interfaces for end-user-oriented facilities providing facilities across application domains. These facilities include the Distributed Document Component Facility (a compound document presentation based on OpenDoc), Time Operations & Internationalization, Data Interchange & Mobile Agents, the Business Object facility, and the Printing facility.
- *Object Services*, which are sets of interface specification providing fundamental services that are likely to be used in any program based on distributed objects. Examples are Name Service (which allows clients to locate object by name) and Trading Service (which allows clients to locate object by properties).

CORBA is a specification of the functionality of the ORB (the message bus that conveys invocation requests and results to CORBA objects). CORBA specification provides both location transparency and programming language transparency. *Location transparency* means that the client invokes operations on a CORBA object without needing to know where the object resides. The invocation of a method of remote CORBA object is as simple as the invocation of a method of a local object. *Programming language transparency* means that a programmer could implement the functionality encapsulated in an object using the most appropriate language such as C, C++, Smalltalk, Ada-95, COBOL, or Java. The language transparency is made possible by the implementation-neutral interface definition language (IDL).

IDL separates the interface from implementation, and provides the basis of agreement about what can be requested of an object implementation via the ORB. It defines what operations an object supports, the parameters types, and the return types of these operations. The programmer of the client side only needs the interfaces defined by IDL to write a client code. Through a language mapping, the client can use the data type defined in IDL.

The IDL compiler generates a stub code and a skeleton. The client can link to the stub code, which translates the client request into a request message to an object implementation. The process of translating a client's request is called *marshalling*. The implementation of the object connects to the skeleton, which unmarshals the request into programming language data types. In the same way, the implementation of the object returns the result to the client. Different IDL compilers with different

language mappings can generate the stub code and skeleton. Figure 2-2 shows how the stub, the ORB, and the skeleton works. The stub code and skeleton code are linked to client and object implementation, respectively. They also connect to the ORB Core and the run-time system for conveying requests from client and results from object implementation.

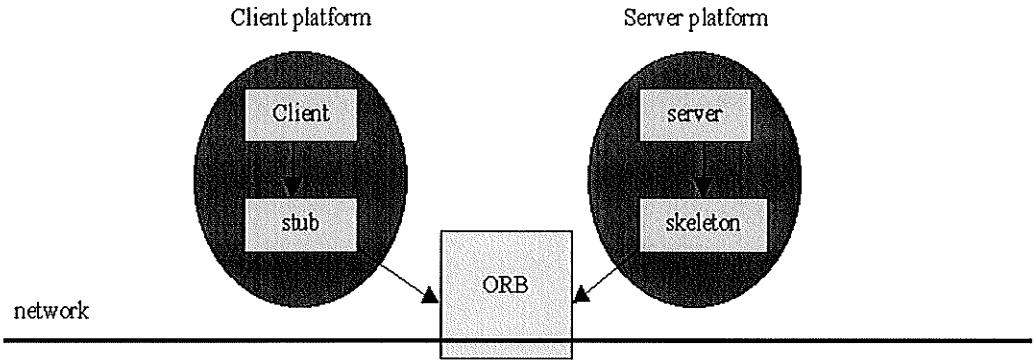


Figure 2-2 Stub, ORB, and Skeleton

Figure 2-3 shows an ORB interface, which contains the static invocation (showed in Figure 2-2), the dynamic invocation and the object adaptor. A static invocation means that the IDL is defined at compile time, and only the operations defined in the IDL interface can be invoked. The CORBA specification defines a *Dynamic Invocation Interface* (DII), which allows client to build requests dynamically for any operation. In response to the DII, a *Dynamic Skeleton Interface* (DSI) is defined for arbitrary requests. The DII defines the form of a request which is sent to an object denoted by an object reference (handle of an object) to request the invocation of a particular operation with particular arguments. By using the DII, clients that know the object by reference can determine its interface type. The DII enables clients build a request without using the stub code generated by the IDL compiler. The DSI deals with

requests in a generic manner. The DSI looks at the requested operations and their arguments, and interprets the semantics dynamically. An object adaptor connects the object implementation with an ORB. The ORB uses the object adaptor to manage the run-time environment of the object implementation.

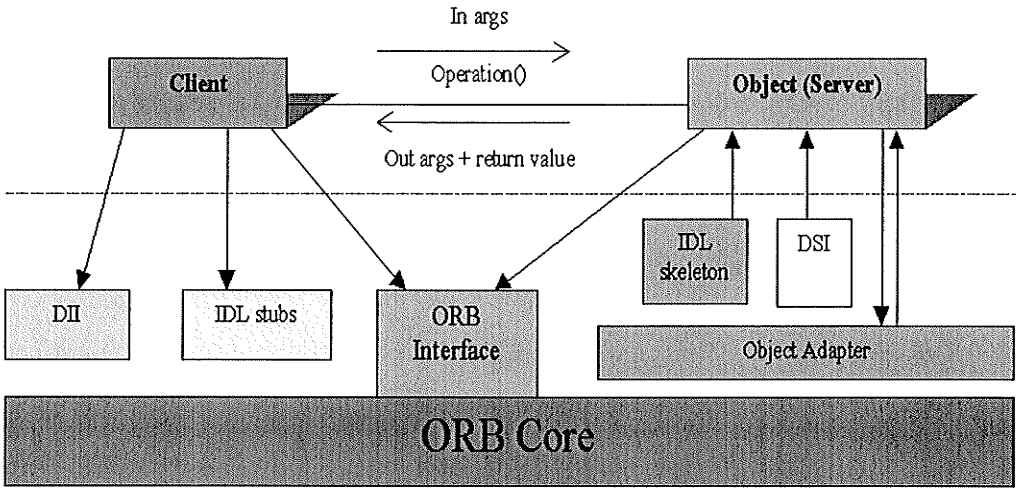


Figure 2-3 ORB Interfaces

The OMG IDL is not a programming language but a declarative language. It does not contain features such as control constructs. So, it cannot be used directly to implement distributed applications. Instead, the IDL is mapped to the facilities of a given programming language via language mapping. Currently, the OMG has standardized several language bindings for CORBA. The current supported language mappings are C, C++, Smalltalk, Ada-95, COBOL, LISP, Python, and Java.

In order to solve the interoperability of different commercial ORB products, CORBA introduced general ORB interoperability architecture that provides direct ORB-to-ORB interoperability. The general ORB interoperability architecture is based on General Inter-ORB protocol (GIOP), which defines a set of message formats and

transfer syntax for ORB interoperation without requiring a particular network transport protocol. The Internet Inter-ORB Protocol (IIOP) specifies how the GIOP is built over TCP/IP; it defines some primitives to assist in establishing TCP connections.

2.8 Borland Visibroker

Borland Visibroker is a leading deployed and adopted ORB. Borland Enterprise Server Visibroker edition is a complete CORBA environment for developing, deploying, and managing distributed applications. Development under Visibroker is supported in both Java and C++, and all product features of Visibroker are CORBA 2.6-compliant. Objects built with Visibroker can be accessed by Web-based applications that communicate using the Internet Inter-ORB Protocol (IIOP). In this thesis, I use Visibroker for Java to implement the prototype system. Steps to create an application using Visibroker are as follows:

1. Specify all objects interfaces using CORBA's Interface Definition Language (IDL).
2. Compile the IDL file with the Visibroker IDL compiler idl2java to generate stub routines for the client program and a skeleton code for the object implementation.
3. Write the client application code, which uses stub routines for method invocations on server objects.

4. Write the server object code, along with the skeleton code to implement server objects.

5. The code for client and server objects is used as input to a Java compiler to compile a Java applet or application, and object server.

These steps are shown in Figure 2-4.

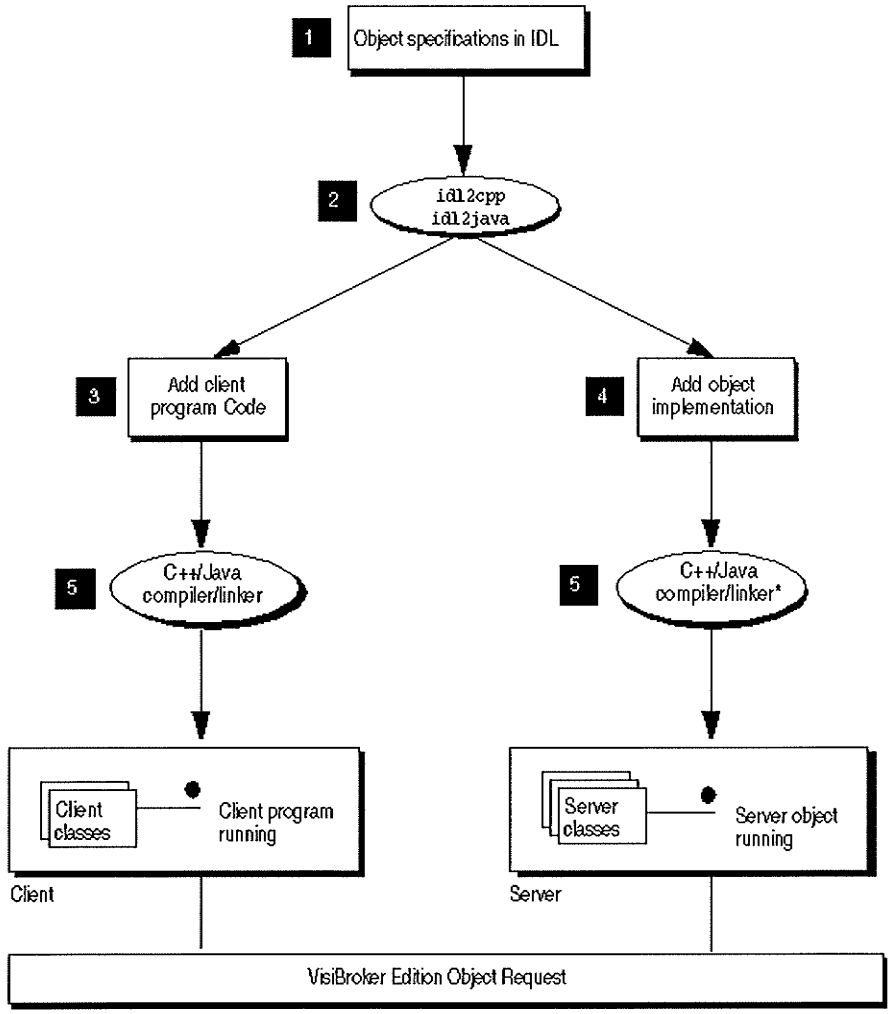


Figure 2-4 Steps to Create an Application Using Visibroker

In addition to providing the features defined in the CORBA specification, Visibroker also offers enhancements that increase application performance and reliability. Some of these enhancements, which are used in this thesis, are explained below.

- Smart binding: When a client application tries to bind to an object, Visibroker enhances performance by using the most efficient communication mechanism. The bind may be established through the shared memory, a pointer reference or a TCP/IP socket, depending on the location of the requested object and the platform.
- Smart agent: The Smart Agent is a dynamic distributed directory service, and it uses simplified APIs that eliminate the complexity of discovering objects. The Smart Agent also provides other facilities (such as load balancing/fault tolerance and location services) for both client programs and object implementations. When a client program tries to invoke an object, the Smart Agent is automatically consulted. Then, the Smart Agent locates the required object implementation, and lets the client make a connection to the object implementation. The communication between the client and the Smart Agent is transparent to the client program.
- Gatekeeper: The Gatekeeper acts as a gateway between applets and server objects. The Gatekeeper enables Java applets to communicate with CORBA server object implementations without compromising network security restrictions imposed by Web browsers.

2.9 Summary

In this chapter, I provided background information on e-commerce transactions (e.g., search, compare, payment, and order transactions), transaction systems (e.g., ACID

properties), parallel processing techniques (e.g., shared nothing systems, intra-operation parallelism), and the workflow. In addition, I also introduced the middleware framework used in this thesis (i.e., CORBA).

Chapter 3 System Design and Algorithms

The three-tiered architecture is usually used to design an e-commerce system because such architecture provides great flexibility and scalability; it separates a system into three distinct tiers: the presentation service tier, the business logic tier, and the database tier.

- The *presentation service tier* provides user interfaces that present information to a user, and collect information from the user.
- The *business logic tier*, often referred to the "middle tier", contains most of the application logic. This tier coordinates the application, and processes commands; it also moves and processes data between to surrounding tiers.
- The *database tier* is the DBMS access tier. Information is stored and retrieved from databases. In this tier, the information is passed back to the business logic tier for processing, and eventually back to the user. Databases in the database tier can only be accessed by the business logic tier and cannot be accessed by the client directly.

3.1 Opportunities that Can Apply Parallel Processing Techniques

The three-tiered architecture separates an application into different blocks and makes the application easier to maintain and upgrade. A three-tiered architecture system can be easily deployed on a distributed environment, which provides opportunities for applying parallel processing.

In following sections, I will explore some opportunities where we can apply parallel processing techniques. Most of these opportunities are in the business logic tier and the database tier. An e-commerce system contains some typical operations such as product search, product comparison, payment processing, and order processing. All these operations can benefit from applying parallel processing techniques.

3.1.1 Parallel Search and Comparison

The search operation can be divided into several steps. First, in the presentation service tier, a customer provides some search criteria and then submits the search criteria to an e-commerce system. The business logic tier receives the search criteria from the customer and converts the search criteria into database queries, and then submits the queries to the database tier. The database tier processes the queries and returns the result to the presentation service tier. Then, the business logic tier may apply some business logic to the data and returns the result to the customer.

In an e-commerce system, it is quite common to divide a large database into several partitions and distribute these partitions to different database servers. When the database is divided and distributed to different database servers, it is possible to perform *parallel search* on these partitions. We can perform search in different data partitions concurrently and merge the search result, and then return the result to customers.

To provide broader selection of products, some e-commerce systems provide

services to let customers search for products from the e-commerce systems of their partners. In these cases, when an e-commerce system receives a search request, it will perform a search on its own local database servers. At the same time, the system will forward the search criteria to its partner systems, and will let partner systems perform their searches. The search in the local e-commerce system and the searches in the partner systems can be executed concurrently. When all searches are completed, the e-commerce system will collect and merge search results from different systems. The merged search result will then be returned to the customer. Figure 3-1 shows the activity view of a parallel search.

A customer sometimes needs the search result in a certain order. The e-commerce system needs to compare the search result and return the search result in that specific order. It is also possible for an e-commerce system to apply parallel processing techniques for comparison. The *parallel comparison* is very similar to the parallel search. The difference is when the system merges search result from different partitions or from different systems. The parallel comparison must compare the search result from different partitions or from different systems and return the search result in order. Figure 3-2 shows an activity diagram of parallel comparison.

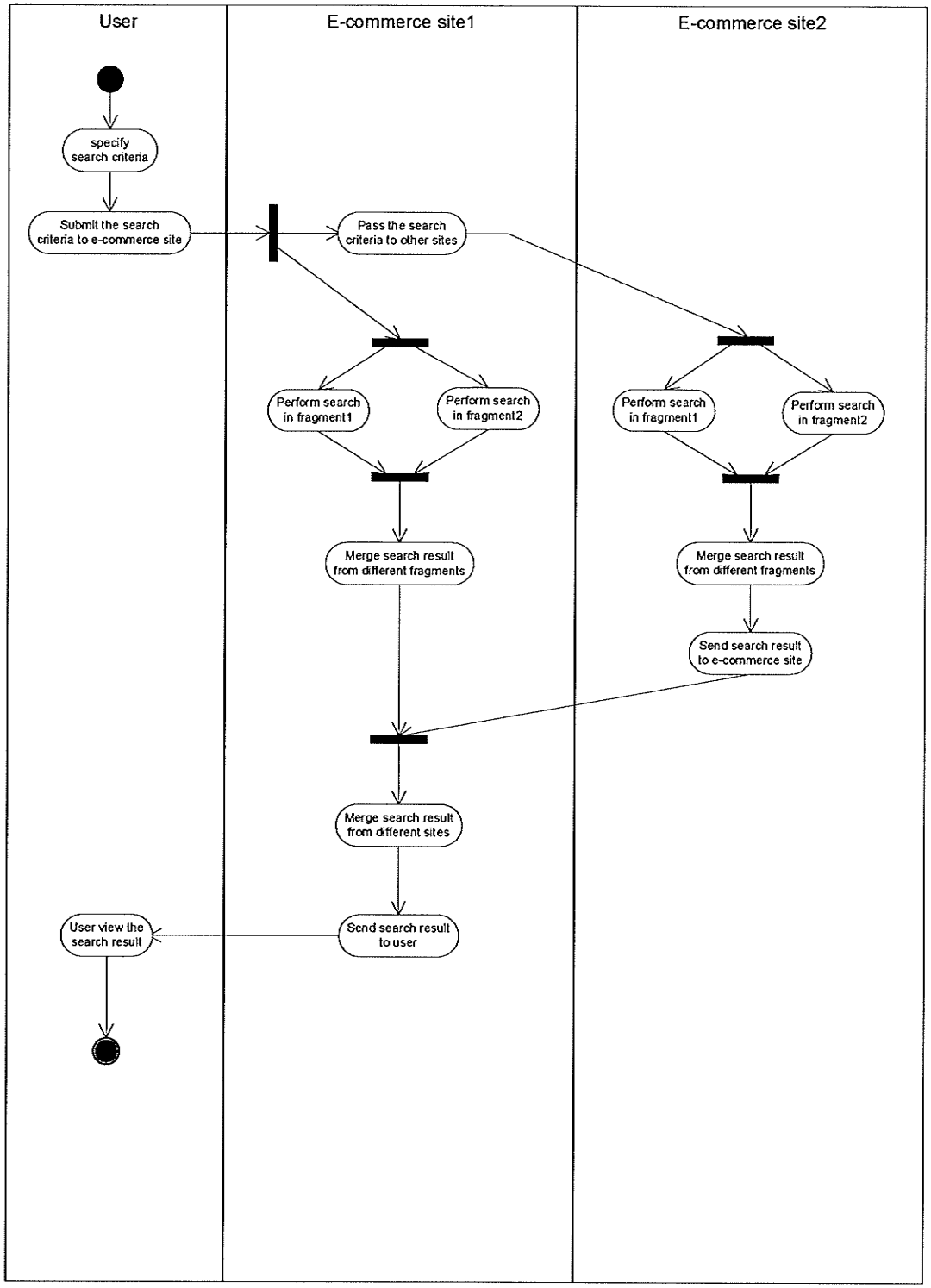


Figure 3-1 Activity View of Parallel Search

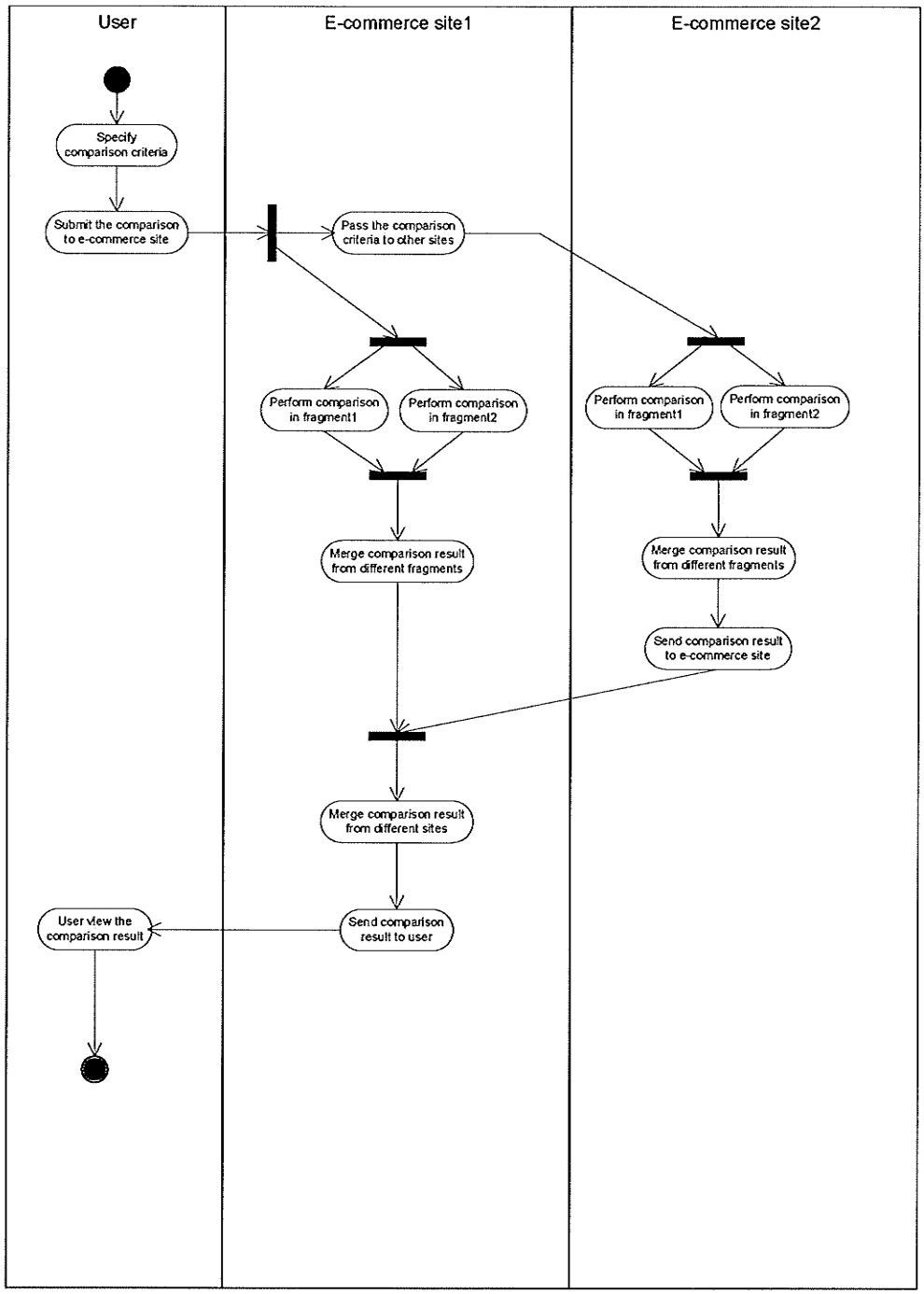


Figure 3-2 Activity View of Parallel Comparison

3.1.2 Parallel Payment Processing

The payment process consists of processes in at least two different accounts: the first process withdraws money from the bank account of the customer, and the second

process deposits money to the bank account of the e-commerce system. In some cases, an e-commerce system may allow a customer to withdraw money from multiple bank accounts. In those cases, the payment process consists of more than two processes.

The *withdrawal process* consists of three sub-steps. The bank will first validate the customer's bank account, then check the available credit of that account, and finally withdraw money from that account. The *deposit process* consists of two sub-steps. The bank will first validate the merchant account, and deposit money to that account.

Because the withdrawal and the deposit processes can be operated in different accounts or different banks, it is possible for us to apply parallel processing techniques in the business logic tier. When the e-commerce system receives the customer payment information, the system initiates two threads. The first thread deposits money to the bank account of the e-commerce system, and the second thread withdraws money from the bank account of the customer. These two threads are executed concurrently.

In the context of parallel processing, it is important to keep the ACID properties of a transaction. If one thread fails in one of its sub-steps, all threads should roll back their changes. If all threads are executed successfully, all their changes should be committed. In either payment processing, if one of the sub-steps fails in the withdrawal thread or the deposit thread, both threads should roll back their changes. Only when both threads are executed successfully, all their changes will be committed. Figure 3-3 shows an activity view of the process of parallel payment. If the

e-commerce system allows a customer to pay from multiple accounts, multiple withdrawal threads should be created.

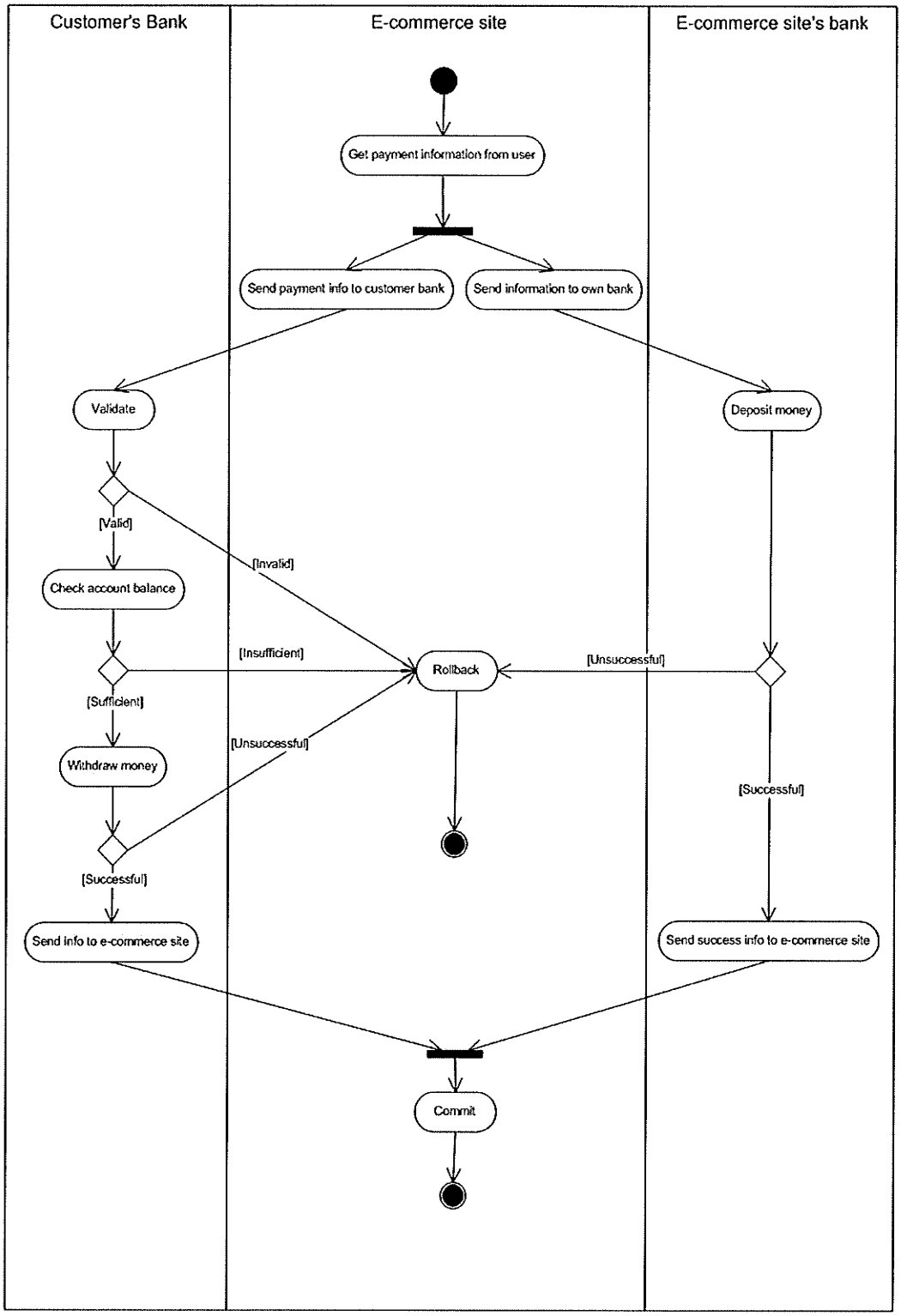


Figure 3-3 Activity View of Parallel Payment

3.1.3 Parallel Order Processing

The order processing is the core service of an e-commerce system. The order process consists of several processes: the inventory check process, the payment process, and shipping process. For an e-commerce system, there are many concurrent users and the inventory is updated dynamically. For each order, the system should perform an inventory check against each product in the order (the *inventory check process*). The *payment process* was described in Section 3.1.2. The *shipping process* consists of shipping information confirmation and the shipping arrangement. For these different processes, we can apply parallel processing techniques. We can initialize three different threads for these processes. The first thread checks and updates the inventory. The second thread processes the payment. The third thread processes the shipping. These threads are then executed concurrently. If any of the threads fails in any of its sub-steps, all the threads should roll back their changes, and an error message will be generated and returned to the customer. If all the threads are executed successfully, all the threads will commit their changes. Finally, the system will record the detailed order information, and will return an invoice to the customer for future reference. Figure 3-4 shows an activity view of order process in parallel.

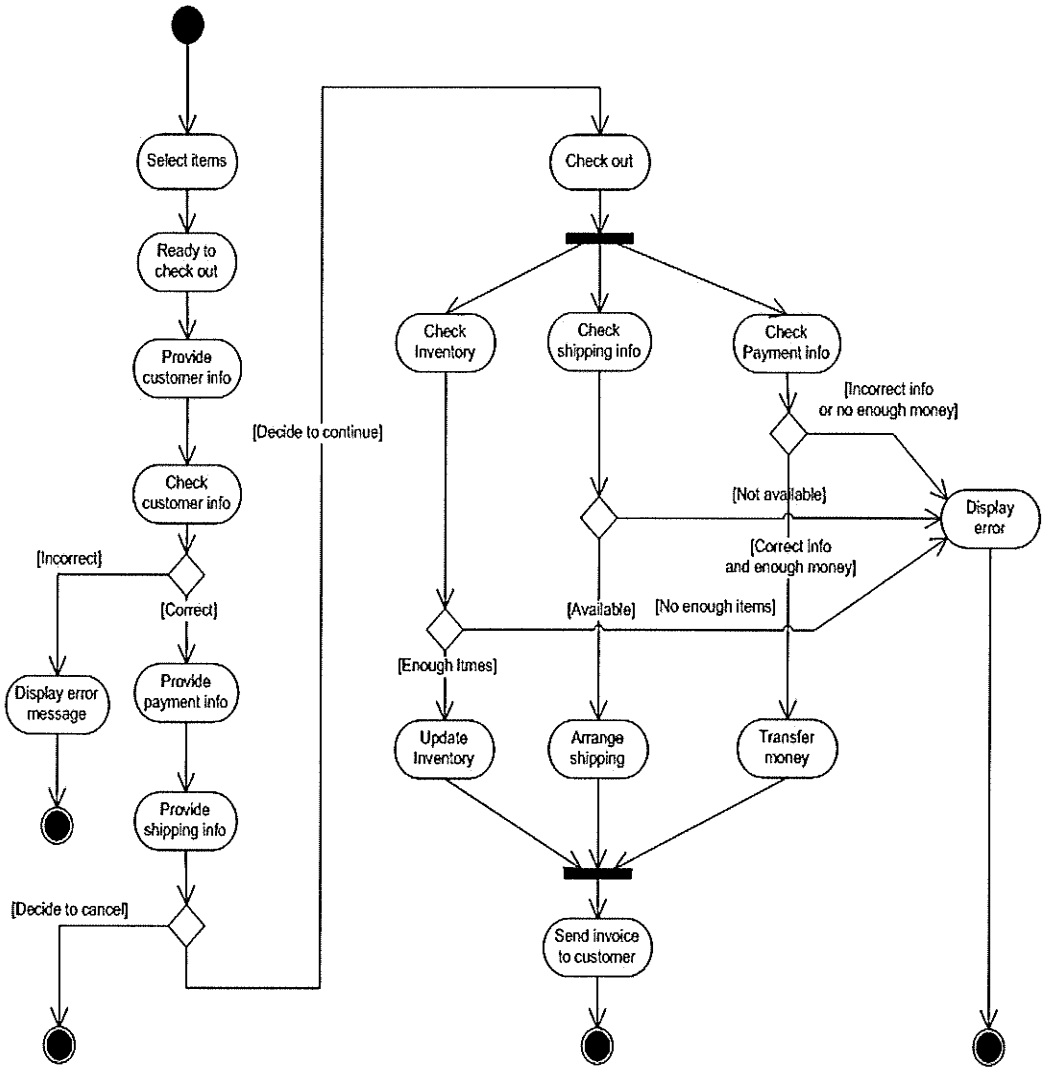


Figure 3-4 Activity View of Parallel Order Process

3.2 Parallel Algorithms

For each opportunity discussed in Section 3.1, a piece of pseudo-code is given to show how parallel processing techniques can be applied.

3.2.1 Parallel Search Algorithm

We can apply parallel processing for search in two different tiers of the system (the database tier and the business logic tier):

1. We can apply parallel processing for the search in the database tier. A database can be fragmented horizontally and distributed to multiple database servers, or simply replicated over a number of database servers. When a user submits a search operation, the operation can be processed in parts across the multiple database servers simultaneously.

2. When the e-commerce system has other partners to provide search service for the system, in the business logic tier, the e-commerce site could transform customer's search criteria into database queries and forward them to partner systems, and execute the queries at the local site and the partner systems simultaneously.

The pseudo-code below shows a parallel search in an e-commerce system.

1) **Input** = search criteria specified by user.
2) **forall** system S_i in S do begin // define threads for different systems
3) q_i = Convert(Input) // convert the user search criteria into a database query
4) Create thread $t_i(S_i, q_i)$ // create a search thread for each system
5) **end**
6) **forall** threads t_i do begin
7) $t_i.start()$ // let all threads be executed simultaneously
8) **end**
9) wait for all threads t_i to stop
10) $output = \bigcup_i output_i$ // collect search result from all systems
11) **Output** = *output* // return the results from different systems to user

The business logic tier of an e-commerce system creates threads for all participating e-commerce systems. Then, the search criteria are transformed into database queries, which are then forwarded to all participating e-commerce systems. The main thread in the business logic tier of the originating e-commerce system starts all search threads to let all threads execute simultaneously. Each participating

e-commerce system performs the search in its own database server and returns the result to the originating e-commerce system. When all threads stop, the originating e-commerce system returns the union of all the results from different systems to the customer. Remarkably, a timeout threshold should be specified for the e-commerce system. If not all of the threads stops (at Step 9) when the specified timeout threshold is reached, the system should stop all the threads and throw a timeout exception.

The pseudo-code below shows how the Convert function converts the user search criteria into database queries (Step 3).

C.1) **Input** *Search Criteria*
C.2) S_1 = *Search Criteria 1*
C.3) S_2 = *Search Criteria 2*
C.4) **...**
C.5) *query string* = select *filed list* from *table list* where $field_1 = S_1$ *and* $filed_2 = S_2$ ...
C.6) **Output** = *query sting*

The Convert function takes the user search criteria as input, and converts the search criteria to the corresponding query string. Because the underlying database structure may be different for different e-commerce systems, different queries may be generated for different participating e-commerce systems.

The pseudo-code below shows the code for a thread t_i to perform search in a participating e-commerce system (Steps 4–10).

S.1) **Input** S_i, q_i
S.2) **forall** all data partitions P_j in S_i do begin
S.3) Create thread $t_j(P_j, q_{i,})$ // create a search thread for each partition
S.4) **end**
S.5) **forall** threads t_j do begin
S.6) $t_j.start()$ // let all threads be executed simultaneously
S.7) **end**
S.8) wait all threads t_j to stop

S.9) $output_i = \bigcup_j output_j$ // collect search result from different data partitions

S.10) **Output** = $output_i$

The pseudo-code above shows how the search is performed in parallel in each participating e-commerce system. The main thread initializes different search threads for different partitions, and lets all the threads execute simultaneously. When all the threads stop, the system collects results from different threads and returns them to the originating e-commerce system. In each participating e-commerce system, if not all the threads stops (at Step S.8) when the specified timeout threshold is reached, the system should stop all the threads and throw a timeout exception.

The pseudo-code below shows the code for a thread t_j to perform a search in a data partition (Steps S.3 – S.9).

P.1) **Input** P_j, q_i
P.2) $output_j$ = result set of perform q_i in data partition P_j
P.3) **Output** = $output_j$

3.2.2 Parallel Comparison Algorithm

Similar to the algorithm describe in Section 3.1.1, parallel processing can be applied to comparison in two different tiers of the system (the database tier and the business logic tier). The differences between parallel comparison and parallel search are as follows. First, when converting the comparison criteria, the query string should include the ordering information. Second, when a thread collects results from other threads, the combined result should be reordered and returned to the user.

To let the query string contain the ordering information, we should add several steps between Step C.4 and Step C.5 in the Convert function in the parallel search.

These steps specify what comparison criteria are used for the comparison.

C.4.1) C_1 = *Comparison Criteria 1*
C.4.2) C_2 = *Comparison Criteria 2*
C.4.2) ...

Step C.5 of the Convert function should also be changed to contain the comparison criteria.

C.5) *query string* = select *filed list* from *table list* where $field_1 = S_1$ *and* $filed_2 = S_2$ ...*ORDER BY* C_1, C_2, ...

When a thread collects search results in a system from different data partitions, it should reorder the combined search result. In the local parallel search algorithm of an e-commerce system, Step 9 could be modified to reorder the combined search result.

S.9) $output_i$ = reorder($\bigcup_j output_j$) // collect search result from different data partitions
// then reorder according the comparison criteria.

When the main thread collects results from different systems, it should reorder the combined search result. In the parallel search algorithm of the main thread, Step 10 could be modified to reorder the combined search result.

10) *output* = reorder($\bigcup_i output_i$) // collect search result from all systems and then
// reorder result according to comparison criteria

3.2.3 Parallel Payment Algorithm

In multithreaded programming, a concept called *barrier* is often used to synchronize threads. When a thread runs to the barrier, it checks if all other threads reach the barrier. If other threads have not reached the barrier, the thread will pause and wait for other threads to reach the barrier. The last thread that reaches the barrier needs to release the barrier and notify all paused threads to run again. The pseudo-code below

describes the barrier class code.

```
1) class Barrier {
2)   int threshold;
3)   int count = 0;
4)   public Barrier(int t) {
5)        threshold = t;
6)        }
7)   public reset() {
8)      count = 0;
9)   }
10)   public synchronized waitForRelease(){
11)      count++;
12)      if (count == threshold) {
13)         reset();
14)         notifyAll();
15)      }
16)      else if (count < threshold) {
17)         wait();
18)      }
19)   }
20)}
```

The Barrier class has two member variables and three functions. The *threshold* is the total number of threads, and the *count* variable reflects how many threads are reaching the barrier and waiting for release. The constructor initializes total number of threads, the *reset* function resets the threads that reach the barrier, and the *waitForRelease* function checks how many threads reach the barrier. The thread will wait if not all threads have reached the barrier. If all threads have reached the barrier, then the *count* will be reset, and all waiting threads will be notified and resume their executions.

The pseudo-code below describes the operation in the business logic tier of an e-commerce system for the parallel payment process described in Section 3.1.2.

```
1) Input = user payment information P
2)   define err              // define error message return to the customer
```

```
3)  create B=Barrier (totalBankThread)// create barrier for threads synchronization
4)  forall customer bank account P_i in P do begin
5)  create thread tw_i(P_i, q_i, err, B) // create a withdraw thread for
                                        // each customer bank account
                                        // with bank account info, withdraw amount
                                        // if some operation fails, then write to err
6)  end
7)  create thread ts(M, err, B)     // create a deposit thread for the merchant bank
                                    // account
8)  forall threads t_j ∈ tw_i ⋃ ts do begin
9)      t_j.start()                 // let all threads be executed simultaneously
10) end
11) wait all thread to stop
12) if err = null then              // check if the payment process is successful
13)     return true                 // if successful return true
14)     else
15)     return err                  // if not successful, return error message
16)end
```

In the code, the business logic tier creates threads to withdraw money from different bank accounts of a customer, and a thread to deposit money to the bank account of the e-commerce system. The business logic tier lets all the threads execute concurrently and waits for them to finish. If any error occurs in these threads, the main thread will return an error message to the customer.

The pseudo-code below describes the operation in a withdrawal thread *tw* or a deposit thread *ts*.

```
T.1) Input bankInfo, err
T.2)    connect to the bank server
T.3)      transform the withdraw or deposit operation to a database query q_i
T.4)    if bankInfo incorrect then      // check the payment information or deposit
                                        // information is correct or not
T.5)        Write error message to err  // if have error than put it in err
T.6)    else
T.7)        if execute q_i unsuccessful  // withdraw money from the account
T.8)            Write error message to err  // if have error than put it in err
T.9)        end
T.10)    end
T.11)    barrier.waitForRelease() // synchronize all threads
```

```
T.12)    if (err == null)
T.13)        commit      // if no error, then commit the change
T.14)    else
T.15)        rollback    // if any error in any thread, then rollback
T.16)    end
T.17) end thread
```

A withdrawal or a deposit thread first connects to the database server of the bank, and transforms the withdrawal or the deposit operation into a query. The thread then validates the bank account. If the account is invalid, the thread logs an error message. If the account information is correct, the thread performs the withdrawal or deposit operation in the account. For the withdrawal thread, if the account does not have sufficient fund, the thread logs an error. For the deposit thread, if for any reason that the deposit is unsuccessful, the thread logs an error. Then, the thread will synchronize with other threads. After synchronization, if the thread finds any error occurs in any of the threads, the thread will roll back all its changes. If the thread finds that all threads are executed successfully, the tread will commit all its changes.

The pseudo-code below describes how to transform withdrawal or deposit operation into database query.

```
1)Input Bank, Account, password, amount, operation
2)  if operation = deposit                  //check deposit or withdrawal
3)      query = "UPDATE BankAccount set Balance=Balance + " amount
            + " FROM Bank.BankAccount WHERE Account =" + account
            + "AND Password =" + password;
4)  else
5)      query = "UPDATE BankAccount set Balance=Balance – " amount
            + " FROM Bank.BankAccount WHERE Account =" + account
            + "AND Password =" + password;
6)Output query
```

3.2.4 Parallel Order Algorithm

The order processing consists of several different processes. An *inventory process* checks if there are sufficient products in the inventory, and updates the inventory. A *payment process* transfers money from customer's bank accounts to the merchant account of the e-commerce site. A *shipping process* checks the customer shipping confirmation and arranges shipping time. An order-detail process records the detailed order information such as the customer information, production information, product price, and quantities.

The pseudo-code below shows the operations of the *main thread* in the business logic tier for the order process described in Section 3.1.3.

```
1) Input =user payment information P, Shipping information U, order information O
2)  define err              // define error message return to the customer
3)  Create B = Barrier (totalThreadNo) // create barrier for threads synchronization
4)  forall customer's bank account P_i in P do begin
5)  Create thread tw_i(P_i, err, B)   // create withdraw thread for each customer's bank
                                      //account
                                      // with bank account info, withdraw amount
                                      // if some operation fails, then write to err
6)  end
7)  create thread ts(M, err, B)       // create a deposit thread for the merchant
                                      //bank account
8)  create thread tin(O, err, B) // create an inventory thread
9)  create thread tsh(S, err, B)// create an shipping thread
10) create thread tor(O, err, B) // create order-detail thread
11) forall threads t_j ∈ tw_i ∪ ts ∪ tin ∪ tsh ∪ tor do begin
12)     t_j.start() // let all threads be executed simultaneously
13) end
14) wait all thread to stop
15) if err = null then          // check if the payment process is successful
16)     return true             // if successful return true
17)     else
18)     return err              // if not successful, return error message
19)end
```

In the code, the business logic tier creates different types of threads. Several threads withdraw money from different bank accounts of a customer. One thread deposits money to the bank account of the e-commerce system. One thread checks and updates the inventory. One thread checks shipping information and arranges shipping; one thread records detailed order information. The business logic tier lets all the threads to be executed concurrently and waits for them to finish. If any error occurs in any of these threads, the main thread will return an error message to the customer.

The payment threads of the withdrawal and deposit thread are described in Section 3.2.3. The pseudo-code below describes the operations in the *inventory thread.*

```
I.1)Input O, qin, err, B
I.2) connect to the inventory server
I.3) foreach productOrder in O
I.4)      transform the inventory operation for productOrder to a database query qin.
I.5)      if execute qin unsuccessful      // update inventory
I.6)              Write error message to err  // if have error than put it in err
I.7)          breaks
I.8)        end
I.9)      end
I.10)     barrier.waitForRelease()  // synchronize all threads
I.11)     if (err == null)
I.12)     commit        // if no error, then commit the change
I.13) else
I.14)         rollback          // if any error in any thread, then rollback
I.15) end
I.16)end thread
```

The inventory thread connects to the database server. For each ordered product in an order, the thread transforms the inventory operation into a database query. Then, the thread performs the inventory operation for this product. If the operation is

unsuccessful (e.g., insufficient stock, or some other reasons), the thread logs an error and stops updating the inventory for unprocessed products in the order; if the operation for the current product is successful, it performs inventory operations for other products in the order. After finishing all inventory operations, the inventory thread then synchronizes with other threads. If the inventory thread finds that an error occurs in any of the threads, the inventory thread rolls back all the changes. If the inventory thread finds that all threads are executed successfully, then it commits all the changes. The pseudo-code below shows how to transform the inventory operation into a database query.

```
1)Input
2)  query = "UPDATE ProductTable set Quantity=Quantity – " quantity
            + " FROM Product.Product WHERE Product.ProductID="
            + productID;
3)Output query
```

The pseudo-code below describes the operations in the *shipping thread.*

```
S.1) Input S, qsh, err, B
S.2)    connect to the shipping server
S.3)    transform the shipping operation to a database query qsh
S.4)    if execute qsh unsuccessful    // arrange shipping date
S.5)            Write error message to err// if have error than put it in err
S.6)        end
S.7)    synchronize all threads and exchange err    // synchronize all threads
S.8)    if (err == null)
S.9)            commit      // if no error, then commit the change
S.10)       else
S.11)           rollback    // if any error in any thread, then rollback
S.12)       end
S.13)   end thread
```

The shipping thread connects to the database server, and transforms a shipping operation into a database query. The shipping thread then executes the query to perform the shipping operation. If the operation is unsuccessful (e.g., incorrect

shipping information), the thread logs an error. Then, the shipping thread synchronizes with other threads. If the shipping thread finds that any error occurs in any of the threads, the shipping thread rolls back all its changes. If the shipping thread finds that all threads are executed successfully, then it commits all the changes. The pseudo-code below shows how to transform the shipping operation to a database query.

```
1)Input customerID, address, shippingMethod, shippingFee, processTime
2)  query = "INSERT INTO SHIPPINGTABLE values (" + customerID+","+
    address+","+ shippingMethod+","+ shippingFee+","+ processTime+")"
3)Output query
```

The pseudo-code below describes the operations in *order-detail thread.*

```
O.1)Input O, qor, err;B
O.2)     connect to the bank server
O.3)     transform the record order information operation to a database query qor
O.4)     if execute qor unsuccessful    // record order information
O.5)         Write error message to err// if have error than put it in err
O.6)     end
O.7)     foreach productOrder in O
O.8)         transform the record order inform operation to a database query qor
O.9)         if execute qor unsuccessful    // record detailed order information
O.10)            Write error message to err// if have error than put it in err
O.11)        end
O.12)    end
O.13)    synchronize all threads and exchange err    // synchronize all threads
O.14)    if (err == null)
O.15)        commit        // if no error, then commit the change
O.16)    else
O.17)        rollback      // if any error in any thread, then rollback
O.18)    end
O.19)    end thread
```

The order-detail thread records the order information and the detailed order information. Then, the order-detail thread synchronizes with other threads. If the order-detail thread finds any error occurs in any of the threads, the order-detail thread

will roll back all its changes. If the order-detail thread finds that all threads are executed successfully, then it commits all the changes. The pseudo-code below shows how to transform the record order information to a database query.

```
1) Input customerID
2)          query = "INSERT INTO orders (customer_ID, Order_Time) Values(
             customerID + "," + currentTime())
3) Output query
```

The pseudo-code below shows how to transform the record detailed order information to a database query.

```
1) Input orderID, productID, productPrice, productQuantity
2)       query = "INSERT INTO orderdetails
                 (order_ID, product_ID, product_Price, product_Quantity)
                 Values(orderID, productID, productPrice, productQuantity)
3) Output query
```

3.3 Summary

In this chapter, I analyzed some typical e-commerce transactions (e.g., search, comparison, payment processing, and order processing) that can benefit from applying parallel processing techniques. In addition, I also provided pseudo-codes of parallel algorithms for handling these transactions.

Chapter 4 Implementation

This chapter describes the implementation of my prototype e-commerce transaction processing system.

4.1 System Description

To discuss how to apply parallel processing techniques in a more specific context, I implemented a prototype e-commerce system, which is a subset of an online bookstore, in this thesis. In the system, a user may log in as a customer, search for favorite books, add books to cart, check out books, and make payment.

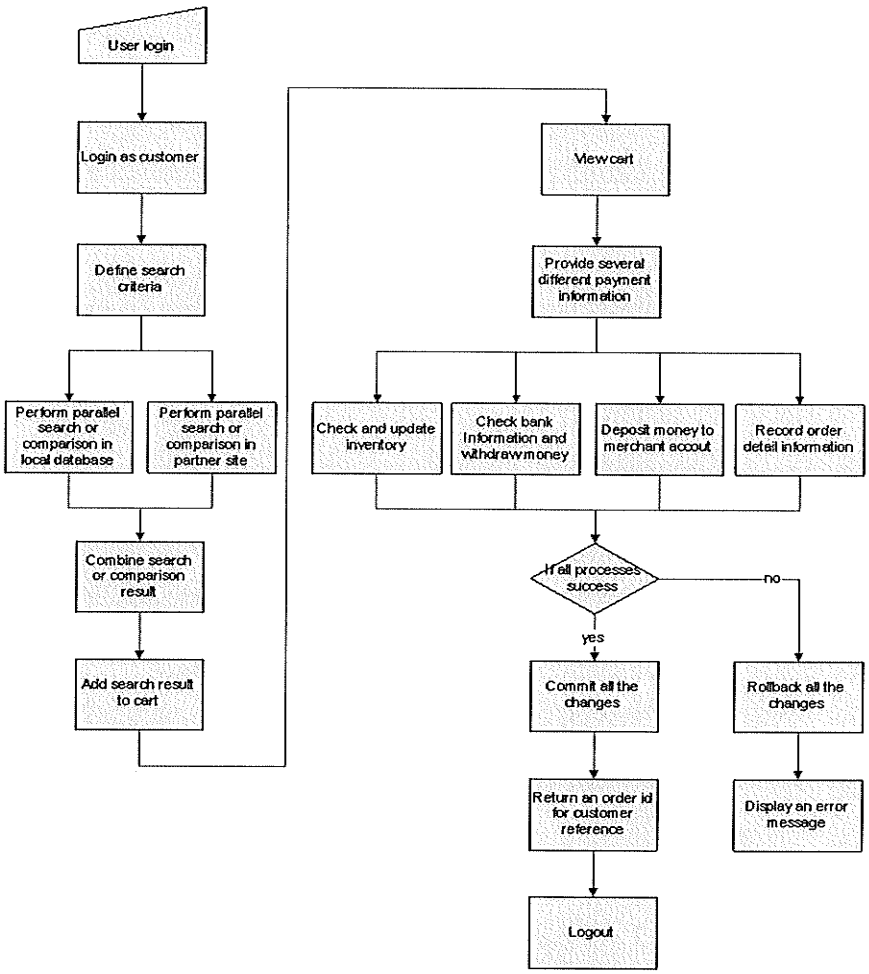


Figure 4-1 System Flowchart

Figure 4-1 shows the flowchart of the prototype e-commerce system implemented in this thesis. As shown in Figure 4-1, a user must log in to the system as a customer to buy books from the online bookstore. If the customer's login is successful, the customer could specify the search criteria to find books. If the customer wants the result in a certain order, he or she could further specify the ordering criteria. Then, the customer could press the search button to perform a search operation. In the search operation, the system initializes several threads to perform concurrent searches in several different sites. One of the threads performs a parallel search in databases of a local system, and other threads perform parallel searches in databases of partner e-commerce systems. After all threads finished their searches, the e-commerce system combines the search results from different systems and returns the combined result to the customer. The customer could then choose books from the combined search result and add the books to a shopping cart. The customer could add more books to the shopping cart, or remove books from the shopping cart. If the customer wants to check out, he or she should first provide the payment information. The payment information may include more than one bank account information. After the payment information is provided, the customer could then press the checkout button to check out. The checkout process uses a two-phase commit mechanism. The two-phase commit splits a commit operation into two parts: the prepare phase and the commit phase. In the *prepare phase* of this system, the system initializes several threads to perform several different jobs. One thread checks and updates the inventory. Several threads check bank accounts of the customer and withdraw money from those

accounts. One thread deposits money to the bank account of the e-commerce system. One thread records the order and detailed order information. All these threads perform their operations concurrently without committing their changes in the prepare phase. In the *commit phase*, the system first checks if operations of all threads have been successful. If any error occurs in any of the threads, the e-commerce system will roll back changes made by all those threads and will display an error message. If operations of all threads are successful, the e-commerce system will commit all the changes and give the customer an invoice for future reference. Finally, the customer could log out by pressing the logout button.

4.2 System Architecture

This system consists of the following components:

- The database server – All persistent data (including customer information, book information, order and detailed order information) are stored in relational databases for permanent storage. Java Database Connectivity (JDBC) is used as an interface for the communication between the server object and databases. Databases can only be accessed by server objects in the business logic tier, and cannot be accessed by the client applet directly.
- The server object – The server object provides services to clients. The server object must be running before the client makes any method invocation. Furthermore, an Internet Inter-ORB Protocol (IIOP) proxy, which will be explained later, is used to transfer invocations of clients across a network.

- The Web server – The Web server allows clients to download Web pages and the embedded client applet.
- The Internet Inter-ORB Protocol (IIOP) proxy – There is a security restriction for Java client applet, which is known as the "sandbox security restriction". The restriction means that applets can only communicate with the host from where the applet is originally downloaded. The restriction is fatal for distributed systems since the nature of distributed systems comes from the cooperation of different computers in networks. To solve this restriction, IIOP proxy is used on the Web server host. The IIOP proxy could route all the IIOP messages on behalf of the applet. From the point of view of an applet, the applet is only communicating with the Web server host. From the point of view of CORBA objects, the applet acts as if it resides on the same network, and these server objects are unaware of the fact that they are communicating with applets. The Visibroker implementation of the IIOP proxy server uses a utility called Gatekeeper. The Gatekeeper also provides extra functionality such as HTTP tunneling, which allows the client to execute in a firewall-protected network. Figure 4-2 shows the IIOP gateway.

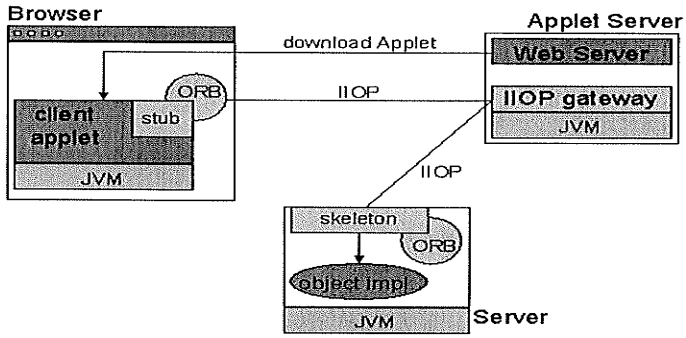


Figure 4-2 IIOP Gateway

- The client applet – The client applet is the front end of the system to users. For this system, the applet is the only visible part for users. The client applet could gather user information, receive user requests, transform user requests into remote method invocation, and display the result returned from the server objects.

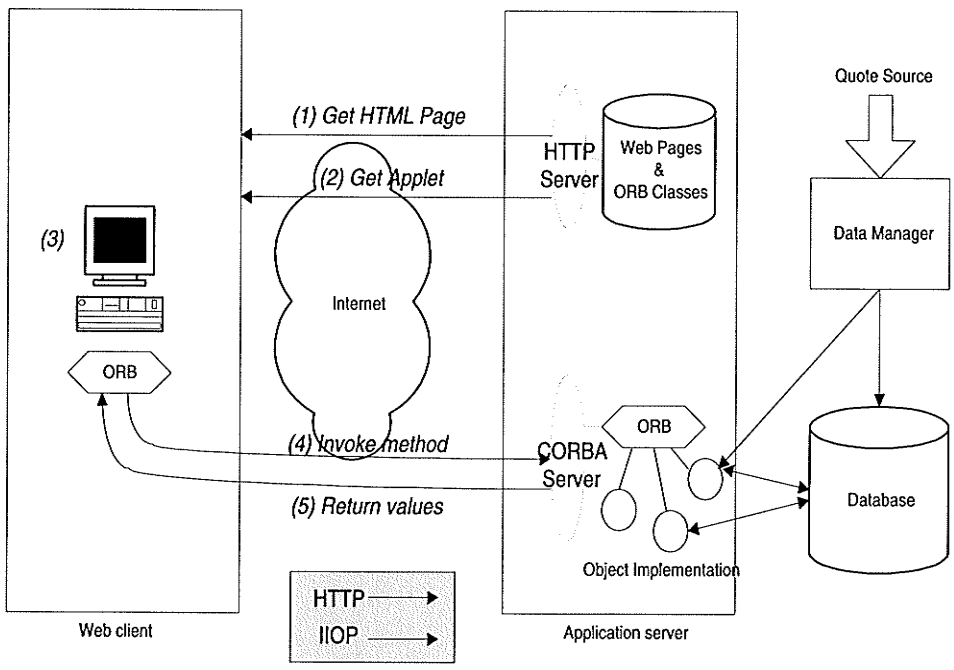


Figure 4-3 System Overview

Figure 4-3 shows the overall system architecture:

1. The Web browser downloads HTML pages from the Web server. Those pages include both HTML contents and references to the embedded client Java applets.
2. The Web browser downloads the Java applets and Object Request Broker (ORB) classes from the Web server. The embedded applets and ORB classes are retrieved in form of the Java byte codes.
3. The Web browser starts the applet and then loads the applet into memory.
4. The applet invokes methods of CORBA server objects. The invocations are sent using IIOP messages. The IIOP proxy (Gatekeeper) wraps IIOP messages into HTTP messages and directs those invocations to server objects in the network.
5. The CORBA server objects return result. The returned result is sent through IIOP messages wrapped in HTTP messages.

This system was developed using Java in both the client side and the server side. Borland Visibroker 5.21 was used as the ORB product. Borland Visibroker for Java provides a pure Java implementation of ORB and a complete interface definition language (IDL) to Java language binding. Moreover, Visibroker provides a useful utility – the Gatekeeper. The Gatekeeper is a proxy utility, which could wrap the IIOP message into HTTP messages. Windows XP was used as the development platform, and Borland JBuilder X was used as the development tool for both the presentation services tier applet and the business logic tier server object. Microsoft SQL Server 2000 was used as the database server in the database tier, and Java Database Connectivity (JDBC) was used to access the databases from the business logic tier.

The system was tested on Windows platform using both the Internet Explorer 6.0 and Netscape Communicator 7.2.

4.3 User Interfaces Design

User interfaces are essential for a user to interact with the system. Figure 4-4 shows the user interface flow used in the system. A user could log in as a customer in the login screen. The customer could perform search operation and add books to a shopping cart in the search screen. In the checkout screen, the customer could provide payment information and check out. Finally, the customer could log out from the logout screen.

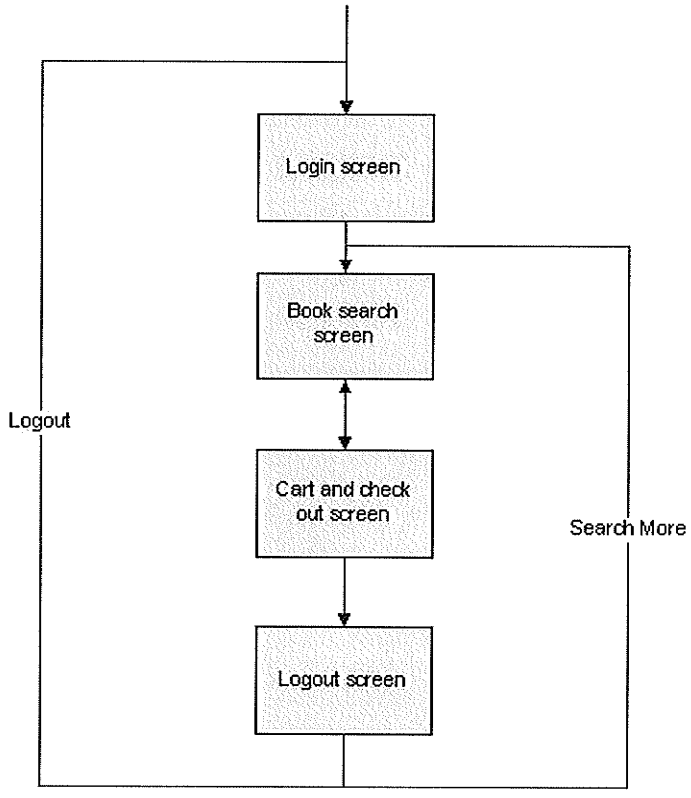


Figure 4-4 Screen Flow

Figure 4-5 shows the login screen of the system. A user must log in as a customer to buy books from this system. The user must provide account and password first, and then press the "Login" button to log in. If the user's login is successful, he or she will see a screen like Figure 4-6; otherwise, the user will receive an error message. The user could press the "Clear" button to clear both inputs in the account and password input boxes.

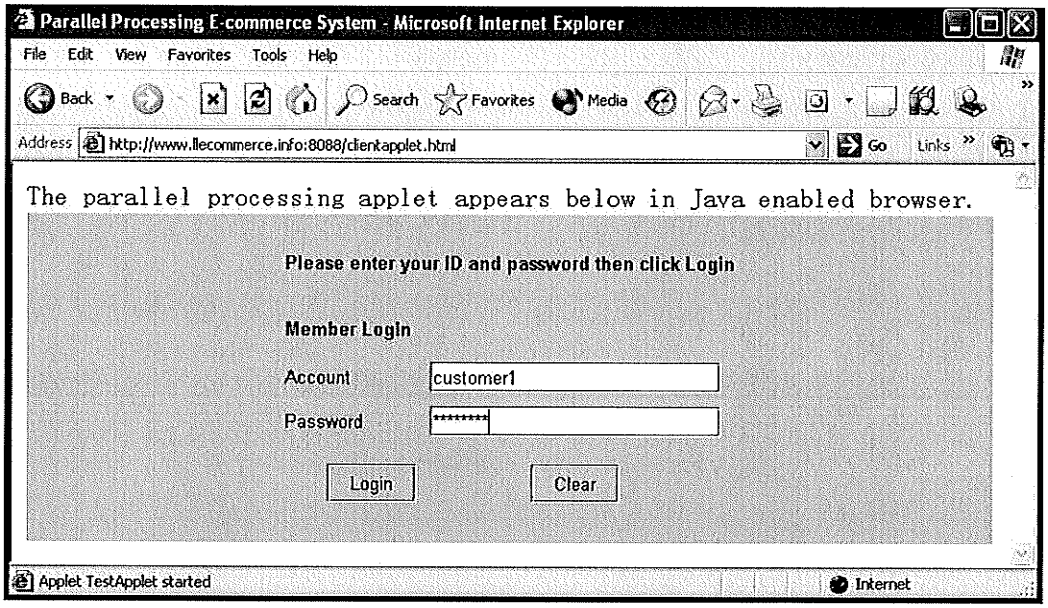


Figure 4-5 The Login Screen

Figure 4-6 shows the initial state of the search screen when a customer logs in. The customer could perform a search operation by specifying the book title, edition, author, publisher, and price range. If the customer wants the search result in a certain order, he or she can also specify on which field and in what way that the search result should be ordered. The customer could press the "Clear" button to clear all the specified search and ordering criteria. After specifying the search and ordering criteria, the customer could press the "Search" button to perform the search operation. If the

search operation is successful, the customer will see a screen like Figure 4-7; otherwise, the customer will receive an error message.

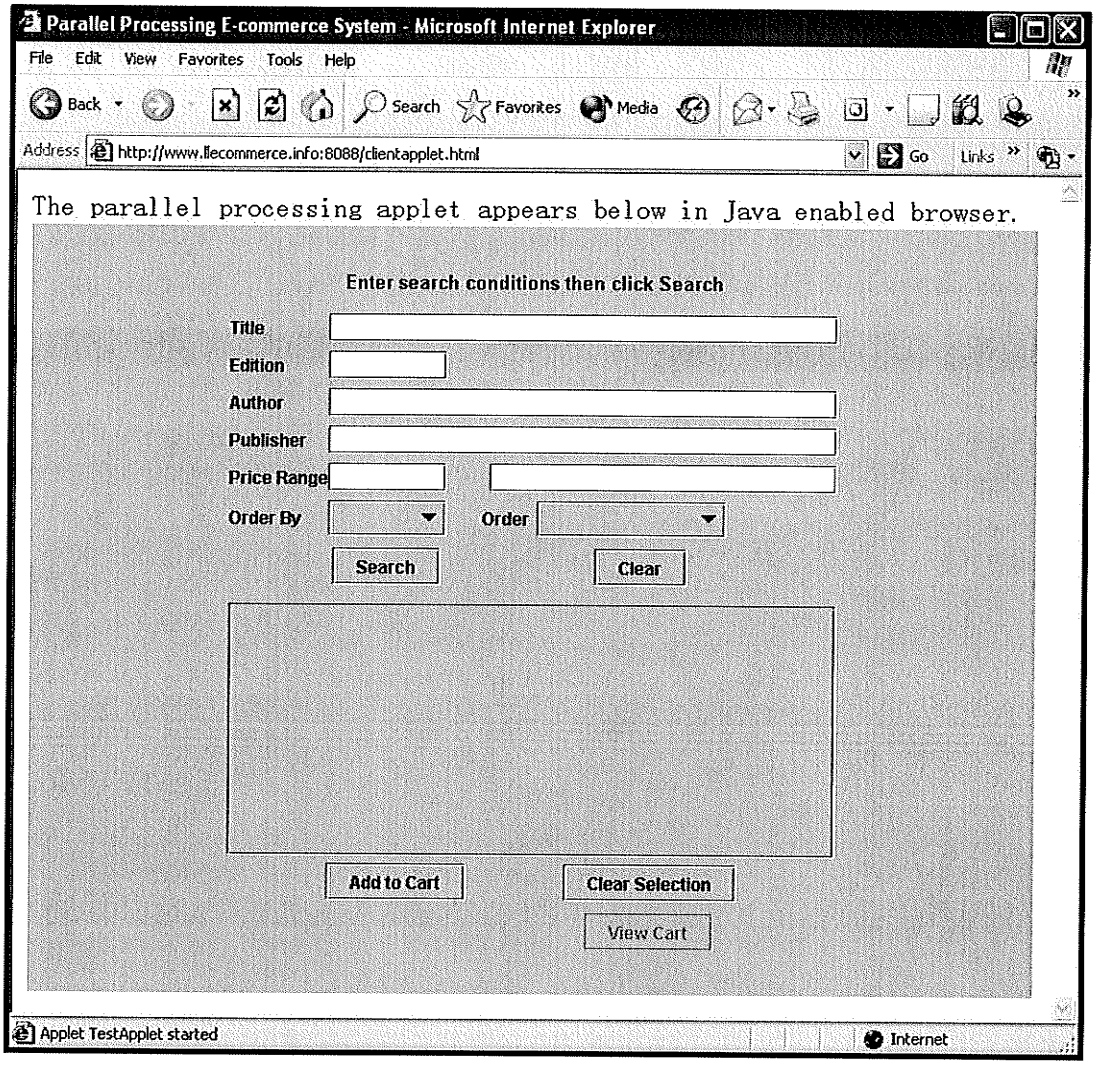


Figure 4-6 Initial State of the Search Screen

Figure 4-7 shows the search-result screen. The customer could choose a book to add into a shopping cart by selecting the book and pressing the "Add to Cart" button. The customer could press the "Clear Selection" button to clear the customer selection. After choosing all the wanted books, the customer could press the "View Cart" button to view the shipping cart shown in Figure 4-8.

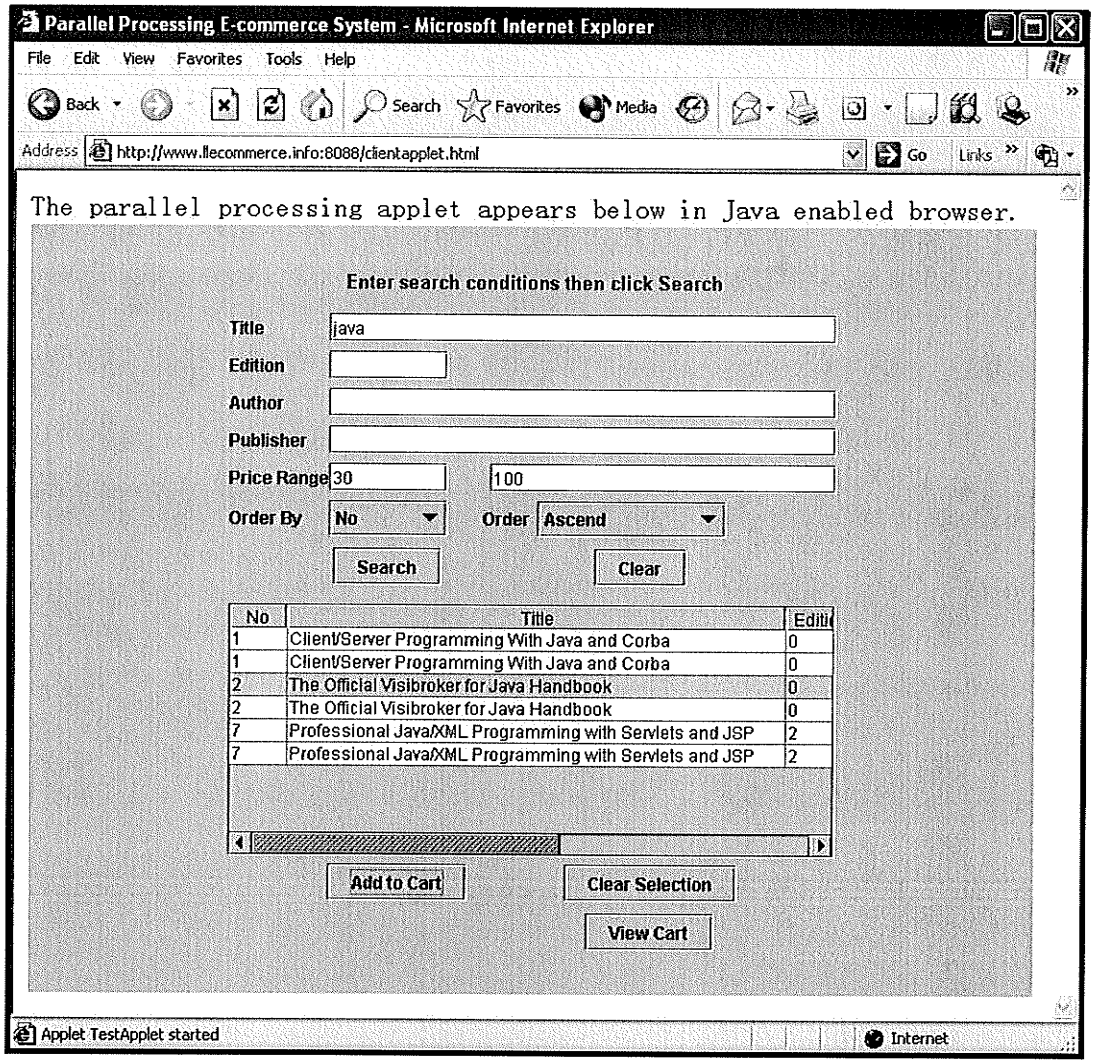


Figure 4-7 The Search Screen after the Customer Performs Search Operation

Figure 4-8 shows the shopping cart screen. In the shopping cart screen, the customer could press the "Select More" button to switch to the search screen to add more books into the cart. The customer could also remove some books from the shopping cart by selecting books from the cart and pressing "Remove from Cart" button. If the customer wants to check out, he or she should provide the bank account information. The system allows money withdrawal from up to three different bank accounts. The customer should specify the name of the bank, the account name, the password that

associates with the account, and the withdrawal amount. The customer could press the "Clear" button to clear all the bank account information. After the payment information is provided, the customer could press the "Check Out" button to check out. If the checkout operation is successful, the customer will see the screen shown in Figure 4-9; otherwise, the customer will receive an error message.

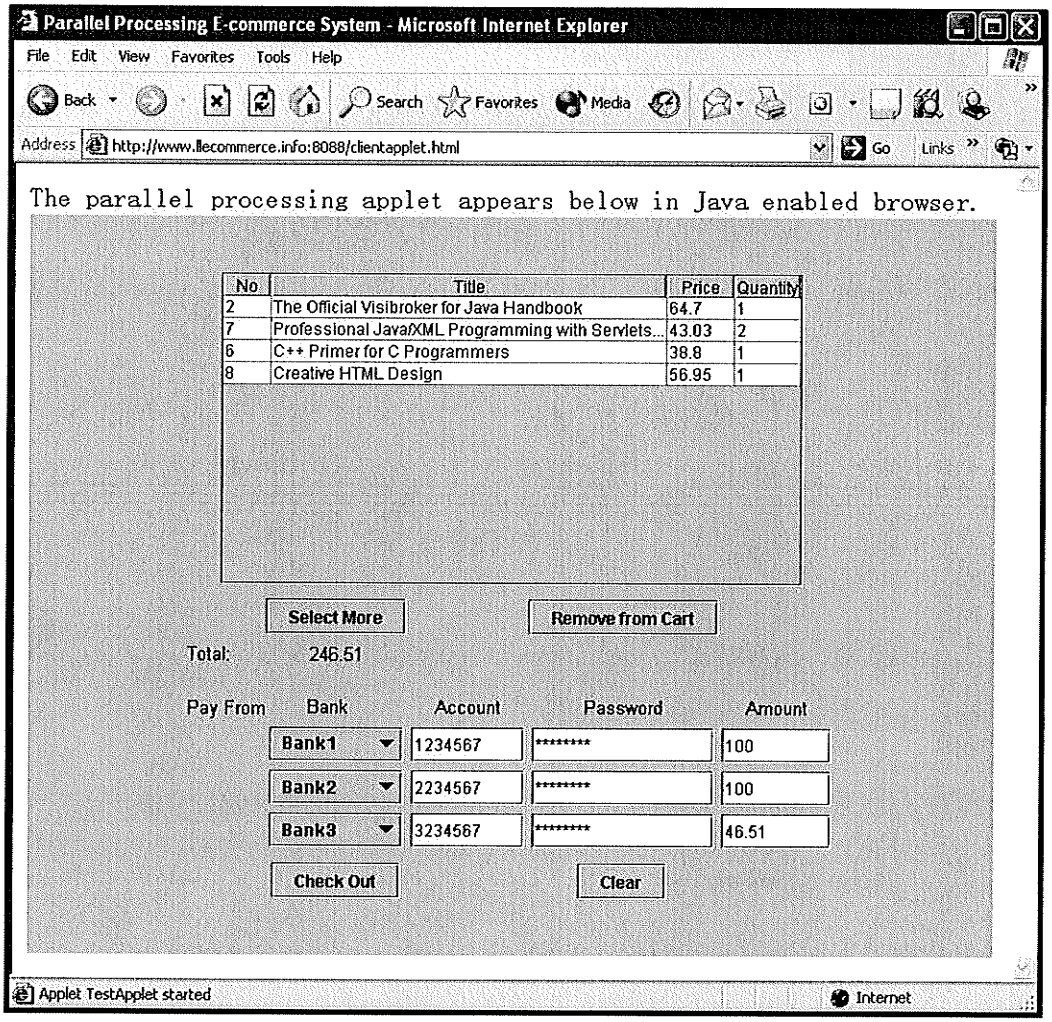


Figure 4-8 The Shopping Cart Screen

Figure 4-9 shows the logout screen. After the customer has checked out, the system processes the order, and returns an invoice number for customer reference. The customer could choose to log out by pressing the "Logout" button. The logout

operation lets the customer log out, and switches the screen back to the login screen as shown in Figure 4-5. Alternatively, the customer could press the "Select More" button to switch back to the search screen as shown in Figure 4-6 without logout.

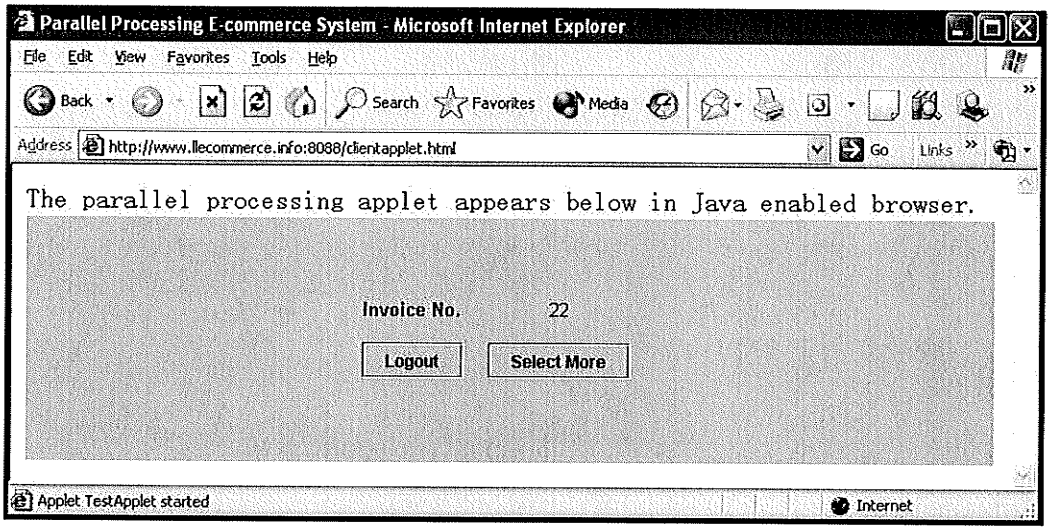


Figure 4-9 The Logout screen

4.4 IDL Design and description

In CORBA, the communication interface between client applet and server object is defined using an interface language, the Interface Definition Language (IDL). The communication interface (an IDL file) of this prototype system is described below.

```
module BookStoreModule {
    // Structure for book information
    struct book_stru {
        long    bookNo;
        string bookTitle;
        short     bookEdition;
        string bookAuthor;
        string bookPublisher;
        float    bookPrice;
    };
```

```
// Structure for order information
struct order_stru {
    long bookNo;
    float bookPrice;
    long bookQuantity;
};

// Structure for customer account information
struct withdraw_stru {
    string bankName;
    string bankAcc;
    string bankPwd;
    float withdrawAmount;
};

// Array of book information
typedef sequence<book_stru> bookSeq;

// Array of order information
typedef sequence<order_stru> orderSeq;

// Array of customer account information
typedef sequence<withdraw_stru> withdrawSeq;


//Exception declaration
exception LoginException {
    string reason;
};

exception SearchException {
    string reason;
};

exception OrderException {
    string reason;
};

exception DbException {
    string reason;
};
```

```
    // Interface definition and signatures of methods of
    // the bookstore server object.
    interface BookStore {

        boolean login(in string account, in string password) raises (LoginException,
                DbException);

        boolean search(in string title, in short edition, in string author,
                in string publisher, in float priceLow, in float priceHigh,
                in boolean ordering, in long orderingField, in boolean ascend,
                out bookSeq searchResult)    raises (SearchException);

        boolean order(in string customerID, in orderSeq orderInfo, in withdrawSeq
                bankInfo, out long invoiceNo) raises (OrderException);
    };
};
```

- **book_stru** defines a structure that holds the information of a book. This structure allows the client to retrieve the information of the book from the server object.
- **order_stru** defines a structure that holds the order information of a book. This structure allows the client to pass the order information of the book to the server object for order processing.
- **withdraw_stru** defines a structure that holds customer's bank account information. This structure allows the client to pass the bank account information to the server object for payment processing.
- **bookSeq** defines an array of *book_stru* that allows the client to retrieve information of multiple books from the server object.
- **orderSeq** defines an array of *order_stru* that allows the client to pass the order information on multiple books to the server object for order processing.
- **withdrawSeq** defines an array of *withdraw_stru* that allows the client to pass information of multiple bank accounts of a customer to the server object for

payment processing

- **Exception declarations:** Four kinds of exceptions are defined in this system: ***LoginException, SearchException, OrderException, DbException.*** The server object throws *LoginException* when a user provides incorrect customer account or password when the user tries to log in. The server object throws *SearchException* when an error occurs in a search operation in one of the participating e-commerce systems. The server object throws *OrderException* in several circumstances: insufficient inventory, incorrect bank information, insufficient credit or cash in bank accounts. The server object throws *DbException* when a database cannot establish a connection or other database errors occur.

- A **BookStore** server object defines all the methods that a client applet can invoke. These methods are described as follows:

  1. The ***login method*** passes a customer account and a password to the server object for authentication. If both the customer account and the password are correct, the method returns true to the client; otherwise, it returns false. If any error occurs in this method, the server object raises a *LoginException*.

  2. The ***search method*** passes search criteria and ordering criteria to the server object for a parallel search in all participating e-commerce systems. The search criteria contains at least one of the fields such as the book title, book edition, book author, book publisher, and price range.

The ordering criteria specify what kind of ordering is used for the search result. If the search operation is successful, the method returns true; otherwise, it returns false. If any error occurs in this method, the method raises a *SearchException*. In the implementation of this method, the system initializes different threads for each participating e-commerce system. All threads run concurrently. After all threads have completed, the originating e-commerce system combines the results from different threads and returns the combined result to the client.

3. The ***order method*** passes the customer ID, the information of shopping cart, and the customer payment information to the server object for order processing. If the order is processed successfully, the method returns true to the client; otherwise, the method returns false. If any error occurs in this method, the method raises an *OrderException*. In the implementation of this method, the system creates several threads for different operations. One thread checks and updates the inventory. Several threads check different customer bank accounts and withdraw money from those accounts. One thread deposits money to the merchant bank account; one thread records the detailed order information. All the threads perform their operations concurrently without committing their changes to database. After all the threads have finished, the originating e-commerce system checks whether all the operations are successful. If so, all the changes will be committed and an invoice number is generated

and returned to the customer; otherwise, all the changes are rolled back and an *OrderException* is raised.

4.5 Server Object

This system is implemented as a three-tiered client/server application. Java applet clients invoke operations of the CORBA server objects in the business logic tier via an IIOP ORB. The server objects contain business logic, and are able to access persistent data in the database tier. The applet clients can only communicate with CORBA server objects. CORBA server objects can communicate with DBMS via JDBC. The server object consists of several main classes:

- **BookStoreModuleServerApp:** A class starts and manages server objects. This class includes the main method to start the server object. The main method provides the following functionalities: Initialize an ORB; register the server object implementation to that ORB; notify a name server of the ORB that has been initialized. Once the ORB has been registered, CORBA clients are able to look for the interface, and sends request through the ORB to the server object.
- **BookStoreImpl**: A server object class that provides the server side implementation of the CORBA interface. This class implements the IDL defined interface and all the following methods defined in that interface.

  1. The ***login method*** takes the customer name and password as input parameters, creates new DbAccess objects, and then passes the customer account and password to the *customerLogin method* of DbAccess for

customer authentication.

2. The ***search method*** takes user search criteria and order criteria as input parameters. The method then creates several *BookSearchThread thread* objects according the search criteria and order criteria, and allows all the search threads to run concurrently. Each search thread is responsible for searching in one site. After all search threads have finished their search, the search method combines the results from different threads and returns the combined result to the customer.

3. The ***order method*** takes the user's payment and shopping cart information as input parameters. The method creates several different kinds of threads for different operations using the input parameters. An *InventoryThread thread* is created to check and update the inventory of books in the shopping cart. Several withdraw threads are created to withdraw money from different customer's bank accounts. One save thread is created to save money to the merchant bank account. One order thread is created to record all the order information and the detailed order information, and generate invoice. All those threads are executed concurrently. If operations of all the threads are successful, the order method returns an invoice number for the customer's future reference; otherwise, an error message will be displayed.

- **Different kinds of threads:** In this system, several different kinds of threads are used for different kinds of operations. These threads are listed below.

1. The ***BookSearchThread*** is responsible for searching books in one site. The thread creates a DbAccess object, and then passes the search criteria and order criteria to the underlying DbAccess object's *bookSearch method* to perform the search.

2. The ***InventoryThread*** is responsible for inventory check and update. The thread first creates a DbAccess object, performs inventory check and update by calling DbAccess object's *updateInventory method*, and then waits for other threads initialized by the *order method* to finish. If all threads have finished successfully, the InventoryThread will commit all its changes; otherwise, the InventoryThread will roll back all its changes.

3. The ***BankThread*** is responsible for withdrawing or saving money to the bank. The thread creates a DbAccess object and performs a withdrawal or a deposit operation by calling the *saveOrWithdrawMoney method* of the DbAccess object, and waits for other threads initialized by the *order method* to finish. If all threads have finished successfully, the BankThread will commit all its changes; otherwise, the BankThread will roll back all its changes.

4. The ***OrderThread*** is responsible for recording all the order information and detailed order information, and generating an invoice. The thread creates a DbAccess object and records order and detailed order information, generates an invoice by calling the *generateOrder method* of DbAccess, and waits for other threads initialized by the *order method*

to finish. If all threads have finished successfully, the OrderThread will commit all its changes, otherwise, the OrderThread will roll back all its changes.

- **DbAccess:** A helper class provides a JDBC connection to the persistent data in databases. DbAccess encapsulates all database operations and handles all interaction with database using JDBC. It includes ten methods: ***connect***, ***closeConnection***, ***setAutoCommit***, ***commit***, ***rollback***, ***customerLogin***, ***bookSearch***, ***updateInventory***, ***saveOrWithdrawMoney***, and ***generateOrder***.

4.6 Database Design

In the implementation of the system, five databases are used. ***BookStore*** store customer information, book information, and order information. The books table of BookStore database is partitioned and distributed to different servers. ***Bank1***, ***Bank2***, ***Bank3***, and ***SaveBank*** databases are used to simulate databases of different banks.

BookStore Database:

Table: Customer table

The customer table holds customer information.

```
CREATE TABLE [dbo].[customers] (
    [Customer_ID] [nchar](10) NOT NULL ,
    [Customer_Password] [nchar](10) NOT NULL ,
    [FirstName] [nvarchar](20) NOT NULL ,
    [LastName] [nvarchar](20) NOT NULL ,
    [Address] [nvarchar](100) NOT NULL ,
    [City] [nvarchar](20) NOT NULL ,
    [Province] [nvarchar](20) NOT NULL ,
    [Country] [nvarchar](20) NOT NULL ,
    [PostCode] [nchar](10) NOT NULL ,
```

```
    [Email] [nvarchar](60) NULL
)
ALTER TABLE [customers]
ADD PRIMARY KEY([Customer_ID])
```

Table: publisher

The publisher table holds book publisher information

```
CREATE TABLE [dbo].[publishers] (
    [publisherID] [int] NOT NULL ,
    [publisherName] [nvarchar] (40) NOT NULL ,
    [publisherAddress] [nvarchar] (100) NOT NULL
)
ALTER TABLE publishers
ADD PRIMARY KEY([publisherID])
```

Table: books1

The books1 table holds the books information stored in server1

```
CREATE TABLE [dbo].[books1] (
    [bookNo] [bigint] NOT NULL
    CHECK ([bookNo]BETWEEN 1 AND 100000),
    [bookTitle] [nvarchar](100) NOT NULL,
    [bookEdition] [smallint] NULL ,
    [bookAuthor] [nvarchar](60) NOT NULL ,
    [bookPrice] [float] NOT NULL ,
    [bookPublisher] [int] NOT NULL ,
    [bookQuantity] [int] NOT NULL
)
ALTER TABLE [books1]
ADD PRIMARY KEY ([bookNo])
ALTER TABLE [books1] ADD
    CONSTRAINT [FK_books1_publisher] FOREIGN KEY
    ([bookPublisher]) REFERENCES [dbo].[publishers] ([publisherID])
```

In this system, the books table is horizontally partitioned, and distributed into different servers. The table books1 is a smaller member table of the original books table, and it stores a horizontal slice of the original books data. The value range for each member table is enforced by a CHECK constraint on the partitioning column, and the value of each member's partitioning column cannot overlap. In the prototype

e-commerce system, the partitioning column is bookNo, the book1 table stores books information ranging from bookNo 1 to bookNo 1000000. In another server, we can create a similar table book2 with bookNo range from 1000001 to 2000000.

Table: Order

The orders table holds order information.

```
CREATE TABLE [dbo].[orders] (
    [Order_ID] [bigint] IDENTITY (1, 1) NOT NULL ,
    [Customer_ID] [nchar] (10) NOT NULL ,
    [Order_Date] [datetime] NOT NULL
)
ALTER TABLE orders
ADD PRIMARY KEY([Order_ID])
```

Table: orderdetails

This table holds detailed order information.

```
CREATE TABLE [dbo].[orderdetails] (
    [Order_ID] [bigint] NOT NULL ,
    [BookNo] [bigint] NOT NULL ,
    [BookPrice] [float] NOT NULL ,
    [BookQuantity] [int] NOT NULL
)

ALTER TABLE [orderdetails]
ADD PRIMARY KEY([Order_ID],[BookNo])
```

View: books_publishers_view

The books_publishers_view is created in each data partition.

```
CREATE VIEW [dbo].[books_publishers_view] AS
    select a.bookNo, a.bookTitle, a.bookEdition,
    a.bookAuthor, b.publisherName, a.bookPrice
    FROM [dbo].[books1] a inner join
    [dbo].[publishers] b on a.bookPublisher = b.publisherID
```

Bank databases:

To perform payment processing, four banks databases that represent data stored in banks are used. These bank databases are almost identical in term of size and structure.

Databases: Bank1, Bank2, Bank3, SaveBank

Table: BankAccount

BankAccount table stores bank information of credit cards.

```
CREATE TABLE [dbo].[BankAccount] (
    [Account] [nchar] (10) NOT NULL ,
    [Password] [nchar] (10) NOT NULL ,
    [Balance] [float] NOT NULL
)
ALTER TABLE [dbo].[BankAccount] ADD
     PRIMARY KEY   CLUSTERED ([Account])
```

4.7 Summary

In this chapter, I discussed the implementation of my prototype e-commerce transaction processing system for an online bookstore. In particular, I described the system architecture, including user interfaces (which allow users to interact with my system) & interface definition language, server objects (which provide users services), and databases (which store all data).

Chapter 5 Performance Analysis

In this chapter, I describe and analyze experimental results of my prototype e-commerce system.

5.1 Metrics Used to Evaluate Performance.

Metrics such as run-time, speedup, and scale-up are often used to gauge the performance of a parallel implementation. I used these three metrics to analyze the performance of the parallel implementation in this thesis.

- **Run-time:** Run-time is the most primitive metric to gauge the performance of a parallel application. I compare the best sequential algorithm run-time t_s with the parallel run-time t_p.
- **Speedup:** Speedup is a metric that captures the relative benefit of solving a problem in parallel over using a single processor system for the same problem. Speedup is defined as:

  $$S(p)=T_1/T_p$$

  where T_1 is the time required by the algorithm on one processor, and T_p is the time required on P processors.
- **Scaleup:** Scaleup is the ability of an application to retain response time as the job size or the transaction volume increases by adding additional resources.

5.2 The Performance Analysis Environment

5.2.1 Hardware and Software

The performance study in this thesis was conducted on six nodes of PCs in the Cargill Lab in Department of Computer Science at the University of Manitoba. One node acts as a Web server, as an application server, and as a database server. All other nodes act as database servers, which store parts of the database. All the nodes are 866 MHz Intel Pentium III systems with 256 MB memory, 20 GB local disk storage, and a fast Ethernet network card. Nodes are linked with 100M fast Ethernet. The Microsoft Windows 2003 Enterprise is the operating system for all nodes, while Microsoft SQL Server 2000 Enterprise is used as DBMS for all the nodes. Borland Enterprise Server 5.21 Visibroker is used as an application server and Borland Enterprise Server 5.21 Gatekeeper is used as a Web server.

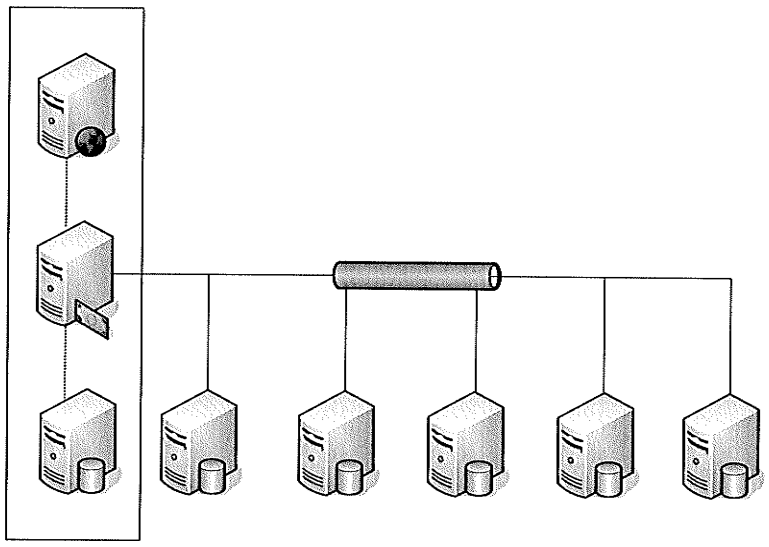


Figure 5-1 System Organization Overview

Figure 5-1 shows the system organization of the network and arrangement of the databases. Each node has its own database instance. The first node gets requests from

clients and spawns query processes on each of the nodes. After each query process finishes executing on each node, the results are returned to the first node, which in turn assembles the results to provide the final output for client request.

5.2.2 Database Description

The ***BookStore*** database contains 6 million records (approximately 380 MB in size). Each of the ***Bank1***, ***Bank2***, ***Bank3***, ***SaveBank*** databases contains 1 million records (approximately 306MB in size).

The performance test was based on three kinds of frequently used e-commerce transactions: the search, comparison, and order processing (Because payment processing was included in the order processing of the prototype system, there was no performance test made for payment processing.) For each kind of transactions, the speedup and the scale-up analyses were performed. For the speedup test, the ***BookStore*** database was partitioned using a range partitioning algorithm on the *BookNo* field and evenly distributed to each node. Table 5-1 shows how data in the ***BookStore*** database were partitioned and distributed.

Table 5-1 BookStore Database for the Speedup & Transaction Volume Scaleup Tests

Test name	Nodes involved	Records in each node	Data size in each node	Total records	Total data size
Test1	1	6M	380MB	6M	380MB
Test2	2	3M	190MB	6M	380MB
Test3	3	2M	126.7MB	6M	380MB
Test4	4	1.5M	95MB	6M	380MB
Test5	5	1.2M	76MB	6M	380MB
Test6	6	1M	63.3MB	6M	380MB

For the scaleup test, two different tests were performed: (1) the transaction volume test and (2) the response time test. The *transaction volume test* measured how many transactions the system could process per minute when the number of processor nodes increased while the amount of data in the database remained constant. The ***BookStore*** database used in the transaction volume scaleup test is shown in Table 5-1. The *response time test* measured the response time for a transaction when the number of processor nodes increased in proportion to the amount of data in the database. The ***BookStore*** database used in the response time scaleup test is shown in Table 5-2.

Table 5-2 BookStore Database for the Response Time Scaleup Test

Test name	Nodes involved	Records in each node	Bookstore data in each node	Total records	Total data size
Test1	1	1M	63.3MB	1M	63.3MB
Test2	2	1M	63.3MB	2M	126.7MB
Test3	3	1M	63.3MB	3M	190MB
Test4	4	1M	63.3MB	4M	253.3MB
Test5	5	1M	63.3MB	5M	316.7MB
Test6	6	1M	63.3MB	6M	380MB

These two scaleup tests are related. Because the transaction volume could be computed using *transaction volume = one minute / response time per transaction*, the transaction volume test could be considered as the response time test when the number of processor nodes increased while the amount of data in the database remained constant.

5.3 Test Results on Searches and Comparisons

The search and comparison transactions are similar. Comparison transactions first perform a *search* operation, compare the result from the *search* operation, and then

return the reordered result to the client. For both search and comparison transactions, different numbers of database servers and different criteria are used for the speedup and the scaleup tests.

5.3.1 Searches Based on One Criterion

Table 5-3 shows the search criterion and result set size of Search1 and Comapre1 for the speedup and the scaleup tests. Figure 5-2 shows the run-time for different queries in the speedup test for Search1.

Table 5-3 Search and Compare Criterion for Search1 and Compare1

Query Name	Criteria	Result set size	Ordering field (for compare only)
Search1-a Compare1-a	Title contains "java"	19	Title
Search1-b Compare1-b	Author contains "dan"	4	Title
Search1-c Compare1-c	Publisher = "Microsoft"	6	Title
Search1-d Compare1-d	Price > 10	60	Title

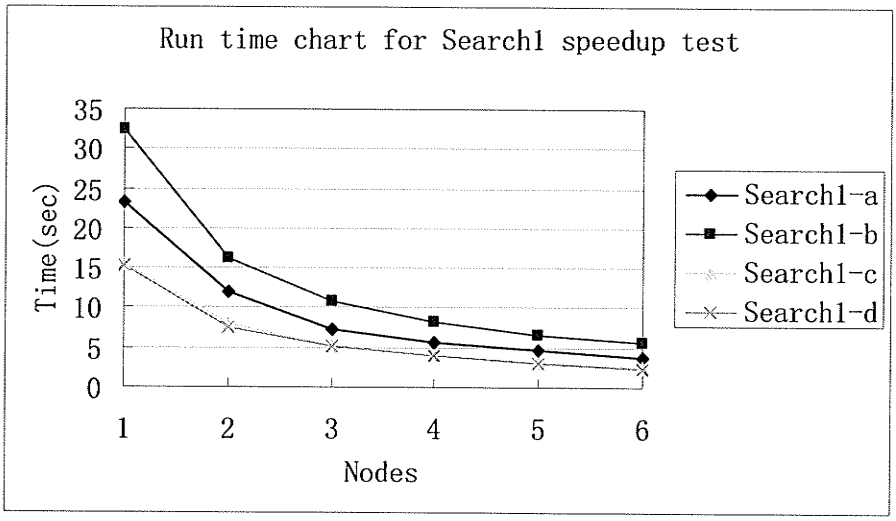


Figure 5-2 Run-time Chart for the Search1 Speedup Test

The run-time of a sequential search transaction (i.e., when the number of nodes = 1) can be computed using:

$$T_{seq\text{-}search} = t_{ss}$$

where $T_{seq\text{-}search}$ stands for the run-time of the sequential search, and t_{ss} stands for the time that used for the sequential search. Similarly, the run-time of a parallel search transaction (i.e., when the number of nodes > 1) can be computed using:

$$T_{par\text{-}search} = t_{start} + t_{ps} + t_{comm}$$

where $T_{par\text{-}search}$ stands for the run-time of the parallel search, t_{start} stands for the time used for initializing the parallel search in different nodes, t_{ps} stands for the time used for the parallel search, and t_{comm} refers to the communication time between nodes (e.g., the time for each node to send result to the originating node). As the search time in each node may be different, t_{ps} is the longest time among all the nodes performing the searches.

Figure 5-3 shows the relative speedup for different queries in the speedup test for Search1; it shows that linear speedups (i.e., *the time taken for searching transactions decreased in proportion to the increase in the number of processor nodes*) were achieved for all searches using one criterion. In our test, since the data for search was large and the result set of the search was small, the communication overhead was small. So,

$$T_{par\text{-}search} = t_{start} + t_{ps} + t_{comm} \approx t_{ps}.$$

Moreover, the number of records in each node was $1/n$ of the original data (where n = the number of nodes involved in the test). Therefore, the scan time of a parallel search

was $1/n$ of a sequential search (i.e., $t_{ps} \approx t_{ss} / n$). Hence,

$$T_{par\text{-}search} = t_{start} + t_{ps} + t_{comm} \approx \ t_{ps} \approx T_{seq\text{-}search} / n$$

and

$$Speedup_{search} = T_{seq\text{-}search} / T_{par\text{-}search} \approx n.$$

This explains why linear speedup was achieved (when result set was small and the data for the search was large). However, when the result set is large, the communication cost t_{comm} will increase, which will then cause a sub-linear speedup.

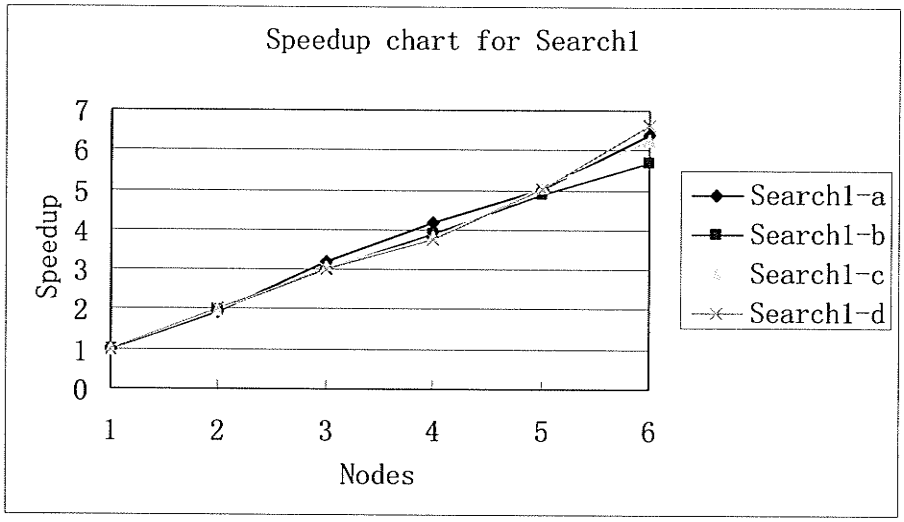


Figure 5-3 Relative Speedup Chart for the Search1 Speedup Test

Figure 5-4 shows how many search transactions can be processed per minute for different queries in the transaction volume scaleup test in Search1. The transaction volume of a sequential search and a parallel search can be computed, respectively, using:

$$Vol_{tran\text{-}seq} = 1 / T_{seq\text{-}search}$$

and

$$Vol_{tran\text{-}par} = 1 / T_{par\text{-}search}.$$

So, the scaleup can be computed using

$$Scaleup_{search} = Vol_{tran\text{-}par} / Vol_{tran\text{-}seq} = T_{seq\text{-}search} / T_{par\text{-}search} \approx n.$$

This explains why in Figure 5-5 linear scaleups (i.e., *the transaction volume was increased in proportion to the number of processor nodes was increased*) were achieved for all searches using different one-criterion queries in various tests for Search1.

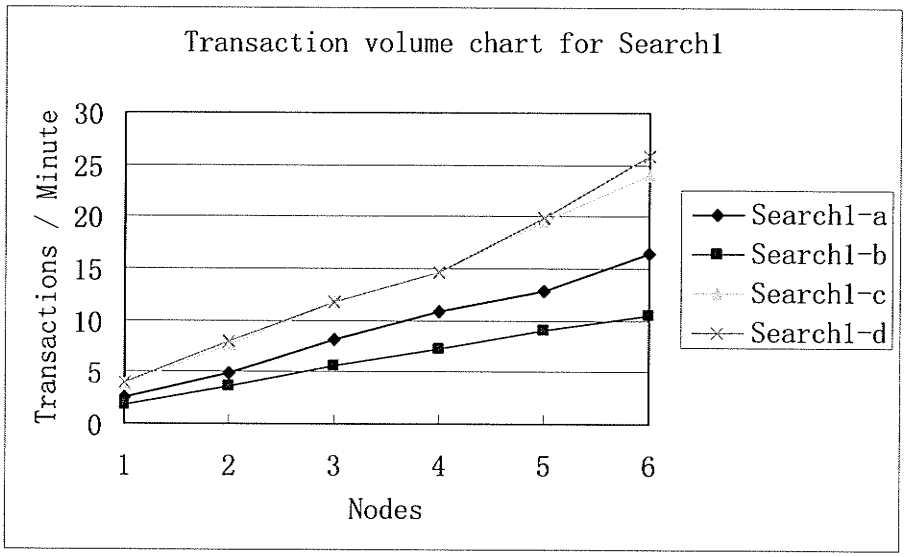


Figure 5-4 Transaction Volume Chart for the Search1 Scaleup Test

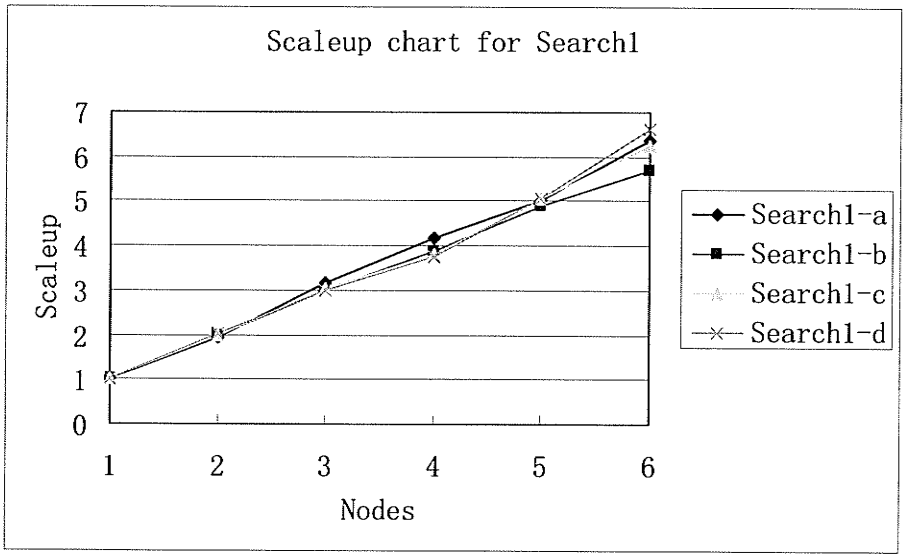


Figure 5-5 Scaleup Chart for the Search1 Scaleup Test

Figure 5-6 shows the response times for the response time scaleup test. In the response time scaleup test, because the number of records in each node was constant, the scan time of parallel search was almost the same as the sequential search in one node ($t_{ss} \approx t_{ps}$). So,

$$T_{seq\text{-}search} \approx T_{par\text{-}search}.$$

This explains why linear scaleups (i.e., *search times were sustained when the number of processor nodes was increased in proportion to the amount of data in the database*) were achieved for all searches using different one-criterion queries in various tests for Search1.

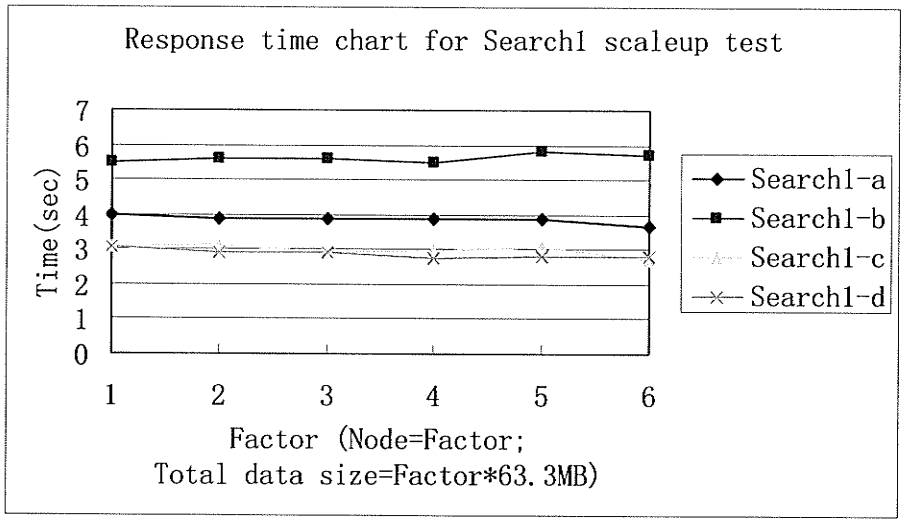


Figure 5-6 Response Time Chart for the Search1 Scaleup Test

For the scaleup tests, linear scaleups were achieved when result set was small and the data for search was large. However, when the result set was large, the communication cost t_{comm} increased. The increased communication cost then caused a sub-linear scaleup.

5.3.2 Comparisons Based on One Criterion

The run-time for comparing transaction can be computed based on the run-time for searching transaction. The run-time for handling sequential comparison transactions (i.e., when the number of nodes = 1) can be computed using:

$$T_{seq\text{-}compare} = T_{seq\text{-}search} + t_{compare} = t_{ss} + t_{compare}$$

where $T_{seq\text{-}compare}$ stands for the sequential compare transaction run-time, and $t_{compare}$ refers to the time used to compare and order the searched result. Similarly, the run-time for handling parallel comparison transactions (i.e., when the number of nodes > 1) can be computed using:

$$T_{par\text{-}compare} = T_{par\text{-}search} + t_{compare} = (t_{start} + t_{ps} + t_{comm}) + t_{compare}$$

where the $T_{par\text{-}compare}$ stands for the parallel compare transaction run-time, and $t_{compare}$ refers to the time used to compare and order the searched result. Figure 5-7 shows the run-time for different queries in the speedup test for Compare1.

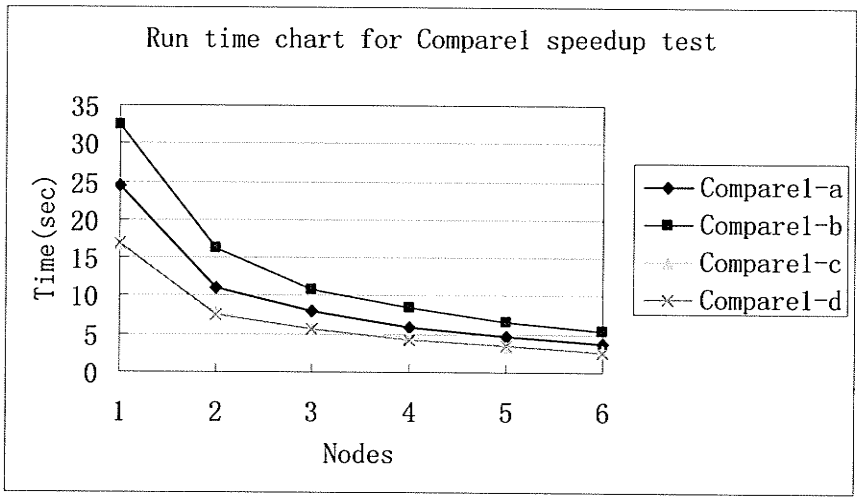


Figure 5-7 Run-time Chart for the Compare1 Speedup Test

Figure 5-8 shows the relative speedup for different queries in speedup tests for Compare1. The experimental test result in Figure 5-8 shows that linear speedups (i.e., the time taken for comparing transactions decreased in proportion to the increase in the number of processor nodes) were achieved for all comparisons using one criterion. Here, the data for search was large, and the result set of the search was small. Therefore, the time used to initialize the search, the time used to communicate, and the time used to compare and order the result set were all short. Hence,

$$T_{seq\text{-}compare} = T_{seq\text{-}search} + t_{compare} = t_{ss} + t_{compare} \approx t_{ss}$$

and

$$T_{par\text{-}compare} = T_{par\text{-}search} + t_{compare} = (t_{start} + t_{ps} + t_{comm}) + t_{compare} \approx t_{ps}.$$

The number of records in each node was $1/n$ of the original data (where n stands for the number of node involved in the speedup test), and the runtime was $1/n$ of the sequential search (i.e., $t_{ps} \approx t_{ss} / n$). Hence,

$$T_{par\text{-}compare} \approx t_{ps} \approx t_{ss} / n \approx T_{seq\text{-}compare} / n$$

and

$$Speedup_{compare} = T_{seq\text{-}compare} / T_{par\text{-}compare} \approx n.$$

This explains why linear speedup was achieved (when the result set was small and the data for the search was large). However, when the result set was large, the communication cost t_{comm} increased, and the time used to compare and reorder the search result $t_{compare}$ increased. These two increased-costs then caused a sub-linear speedup.

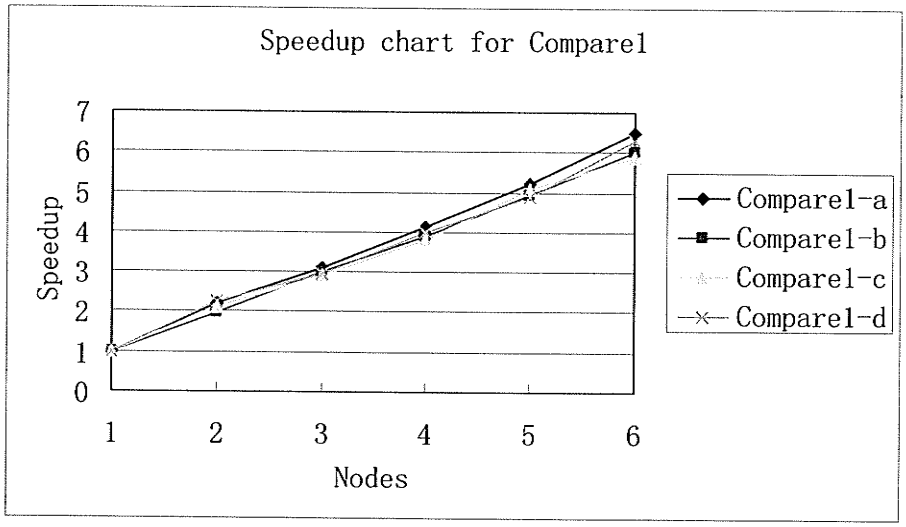


Figure 5-8 Relative Speedup Chart for the Compare1 Speedup Test

Figure 5-9 show how many comparison transactions can be processed per minute for the different queries used in the scaleup test for Compare1. The transaction volume of the sequential comparison and the parallel comparison can be computed using:

$$Vol_{tran\text{-}seq} = 1 / T_{seq\text{-}compare}$$

and

$$Vol_{tran\text{-}par} = 1 / T_{par\text{-}compare}.$$

So, the scaleup can be computed using:

$$Scaleup_{compare} = Vol_{tran\text{-}par} / Vol_{tran\text{-}seq} = T_{seq\text{-}compare} / T_{par\text{-}compare} \approx n.$$

This explains why in Figure 5-10 linear scaleups (i.e., the transaction volume was increased in proportion to the number of processor nodes was increased) were achieved for all comparisons using different one-criterion queries in various tests for Compare1.

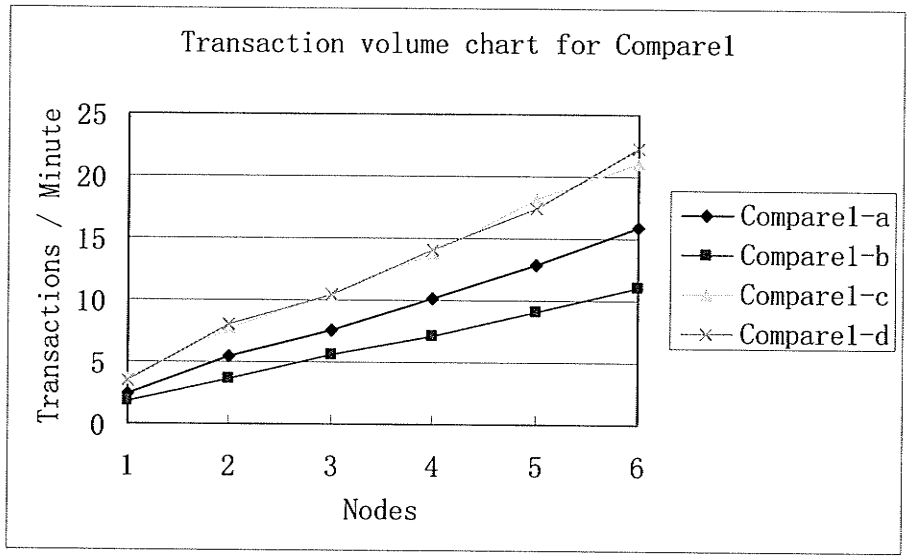


Figure 5-9 Transaction Volume Chart for the Compare1 Scaleup Test

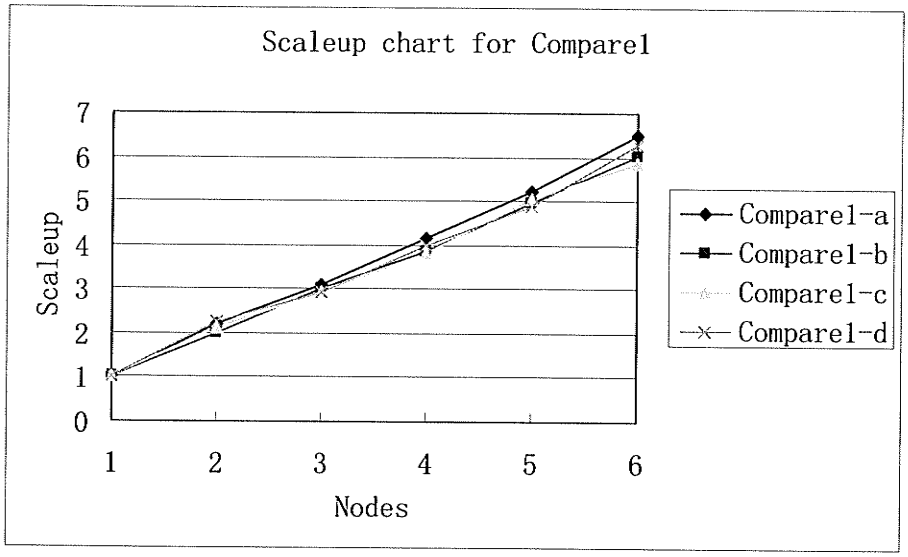


Figure 5-10 Scaleup Chart for the Compare1 Scaleup Test

Figure 5-11 shows response times for the response time scaleup test. In the response time scaleup test, because the number of records in each node was constant, the scan time of parallel search was almost the same as the sequential search in one node ($t_{ss} \approx t_{ps}$). So,

$$T_{seq\text{-}compare} \approx T_{par\text{-}compare}.$$

This explains why linear scaleups (comparison times were sustained when the number of processor nodes was increased in proportion to the amount of data in the database) were achieved for all searches using different one-criterion queries in various tests for Compare1.

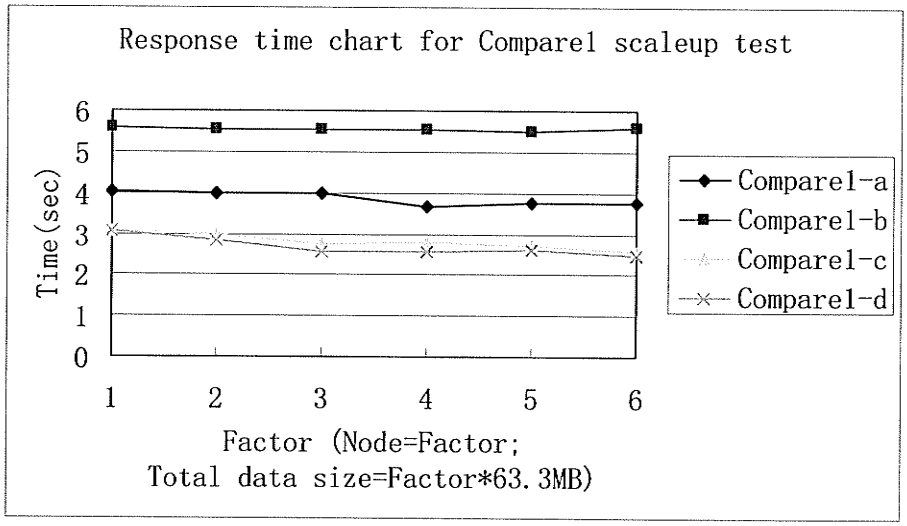


Figure 5-11 Response Time Chart for the Compare1 Scaleup Test

For the scaleup tests, linear scaleups were achieved when result set was small and the data for search was large. However, when the result set was large, the communication cost t_{comm} increased, and the time used to compare and reorder the search result $t_{compare}$ also increased. These two increased-costs then caused a sub-linear scaleup.

5.3.3 Searches and Comparisons Based on Two and Three Criteria

Table 5-4 shows the search criteria and result set size for Search2 and Compare2 tests; Table 5-5 shows the search criteria and result set size for Search3 and Compare3 tests.

Table 5-4 Search and Compare Criteria for Search2 and Compare2

Query Name	Criteria	Result set size	Ordering field (for Compare only)
Search2-a Compare2-a	Title contains "java" Author contains "dan"	2	Publisher
Search2-b Compare2-b	Title contains "java" Publisher = "Microsoft"	2	Publisher
Search2-c Compare2-c	Publisher = "Microsoft" Price > 10	6	Publisher
Search2-d Compare2-d	Price > 10 Price <80	43	Publisher

Table 5-5 Search and Compare Criteria for Search3 and Compare3

Query Name	Criteria	Result set size	Ordering field (for Compare only)
Search3-a Compare3-a	Title contains "java" Author contains "dan" Publisher = "Microsoft"	1	Author
Search3-b Compare3-b	Title contains "java" Edition = 2 Publisher = "John willy and sons"	0	Author
Search3-c Compare3-c	Title contains "java" Publisher = "Microsoft" Price > 10	2	Author
Search3-d Compare3-d	Publisher = "Microsoft" Price > 10 Price <80	43	Author

Experimental results can be found in Appendix A. From these test results, sub-linear speedups and sub-linear scaleups were achieved for both the search and comparisons using both two criteria and three criteria.

5.4 Test Results on Order Processing

For the order transactions, two sets of order tests were performed. In the first set, each transaction includes *small* number of distinct books; in the second set, each

transaction includes *large* number of distinct books. For each set of order transaction, I recorded the run-time, and computed relative speedups and scaleups.

5.4.1 Small Orders

For small orders, I used three different transaction queries with one deposit bank (i.e., one bank for customers to deposit money) and varying number of withdrawal banks (i.e., banks for customers to withdraw money). Query Smallorder-a uses one withdrawal bank for the e-commerce system, query Smallorder-b uses two withdrawal banks, and query Smallorder-c uses three withdrawal banks. Table 5-6 shows distinct banks involved in each transaction. In each transaction in the test, 60 different books are ordered. All distinct books are evenly distributed into different partitions in different nodes.

Table 5-6 Banks Involved in Different Queries in Smallorder

Query	Withdraw bank No.	Deposit bank No
Smallorder-a	1	1
Smallorder-b	2	1
Smallorder-c	3	1

The run-time of a sequential order transaction can be computed using:

$$T_{seq} = T_{seq\text{-}update} + T_{b1} + \ldots + T_{bn} + T_{order}$$

where T_{seq} stands for the run-time of a sequential order transaction, $T_{seq\text{-}update}$ stands for the time used for inventory check and update, $T_{b1}...T_{bn}$ stands for the time used for communicating with different banks, and T_{order} stands for the time used for recording the order and detailed order information. The run-time of a parallel order transaction

can be computed using:

$$T_{par} = \text{Max}(T_{par\text{-}update}, T_{b1},\ldots T_{bn}, T_{order})$$

where T_{par} stands for the run-time of a parallel order transaction, $T_{par\text{-}update}$ stands for the time used for performing inventory check in different data nodes, $T_{b1},\ldots T_{bn}$ and T_{order} are same as above. Since all the operations (inventory update, different bank processing, and order generation) in parallel order transaction are concurrently performed, the processing time T_{par} is the longest processing time of all operations. Figure 5-12 shows the run-time for different queries in a speedup test for Smallorder.

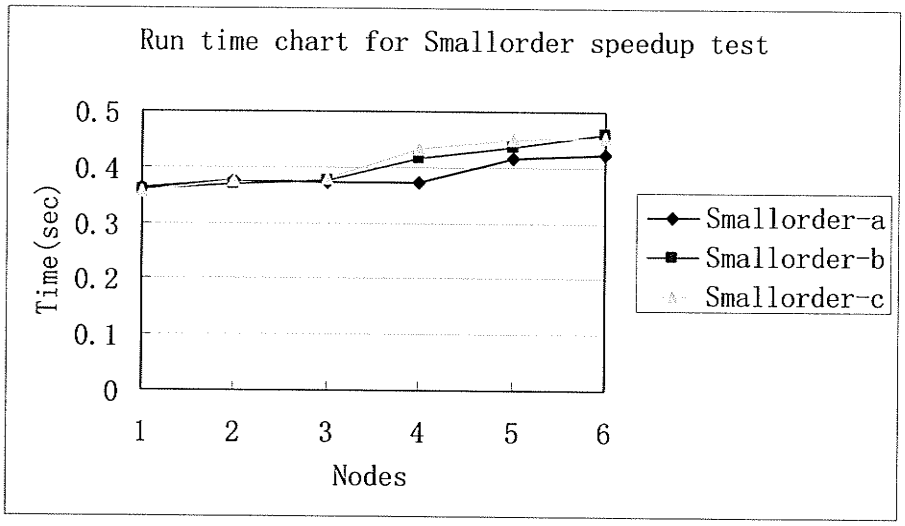


Figure 5-12 Run-time Chart for the Smallorder Speedup Test

Because the number of records in one node was $1/n$ of original data, the update time was $1/n$ of original data. Therefore,

$$T_{par\text{-}update} = T_{seq\text{-}update}/n + T_{comm} + T_{startup}$$

where $T_{startup}$ stands for the time that the originating node to start up the update operation in different nodes, and T_{comm} stands for the time for time that the originating node sends the ordered books information to other nodes for the update operation.

Hence, the run-time of a parallel order transaction can be computed using:

$$T_{par} = \text{Max}(T_{seq\text{-}update} / n + T_{comm} + T_{startup}, T_{b1}, \ldots T_{bn}, T_{order}).$$

For the test for small orders (shown in Figure 5-13), when the involving nodes increased, the communication cost increased and $T_{seq\text{-}update} / n$ decreased. Since $T_{seq\text{-}update}$ was very small, the increase in the communication cost was more than the decrease in $T_{seq\text{-}update} / n$. Therefore, $T_{par\text{-}update}$ increased. Moreover, the speedup test was performed using a local network, the $T_{b1}, \ldots T_{bn}$ were very small, and can be ignored. Thus, when the order was small, $T_{par\text{-}update}$ took longer than $T_{seq\text{-}update}$ when node increased. This means that the performance of parallel order processing was worse than sequential order processing. However, in reality, the communication cost with banks $T_{b1}, \ldots T_{bn}$ was high, and was much greater than $T_{par\text{-}update}$ or $T_{seq\text{-}update}$ for small orders. So,

$$T_{seq} = T_{seq\text{-}update} + (T_{b1} + \ldots + T_{bn}) + T_{order} \approx T_{b1} + \ldots + T_{bn}$$

and

$$T_{par} = \text{Max}(T_{par\text{-}update}, T_{b1}, \ldots T_{bn}, T_{order}) \approx \text{Max}(T_{b1}, \ldots T_{bn}).$$

Comparing to the sequential order transaction, the parallel order transaction took less time ($T_{par} < T_{seq}$ because $\text{Max}(T_{b1}, \ldots T_{bn}) < T_{b1} + \ldots + T_{bn,}$). However, using more nodes did *not* lead to high speedup for small orders, because using more nodes did not affect the communication cost with banks.

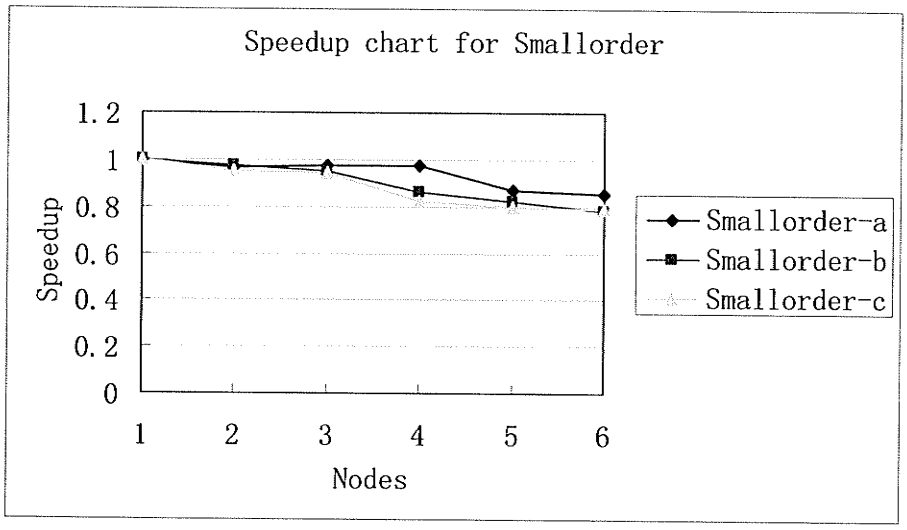


Figure 5-13 Relative Speedup Chart for the Smallorder Speedup Test

Figure 5-14 shows how many order transactions can be processed per minute for different queries in the Smallorder's transaction volume scaleup test. Figure 5-15 shows the response times for different queries in the Smallorder's response time scaleup test.

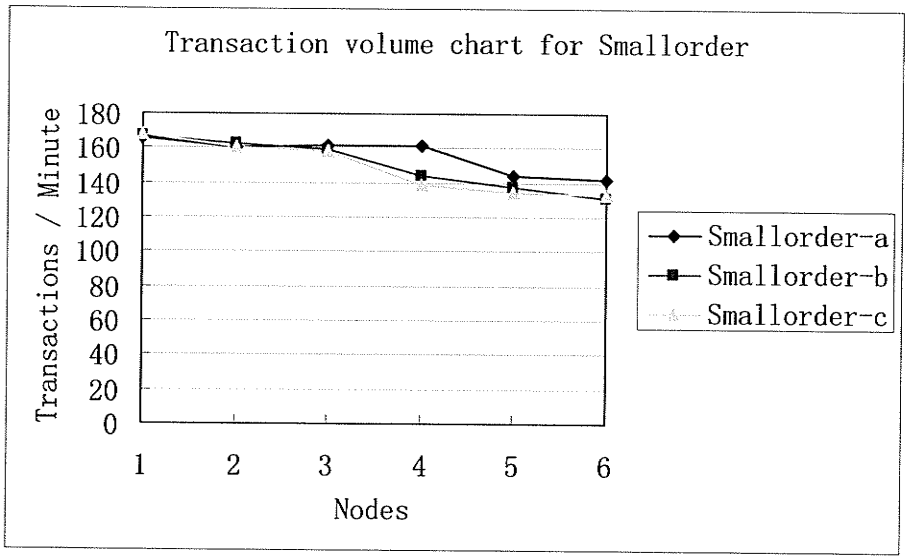


Figure 5-14 Transaction Volume Chart for the Smallorder Scaleup Test

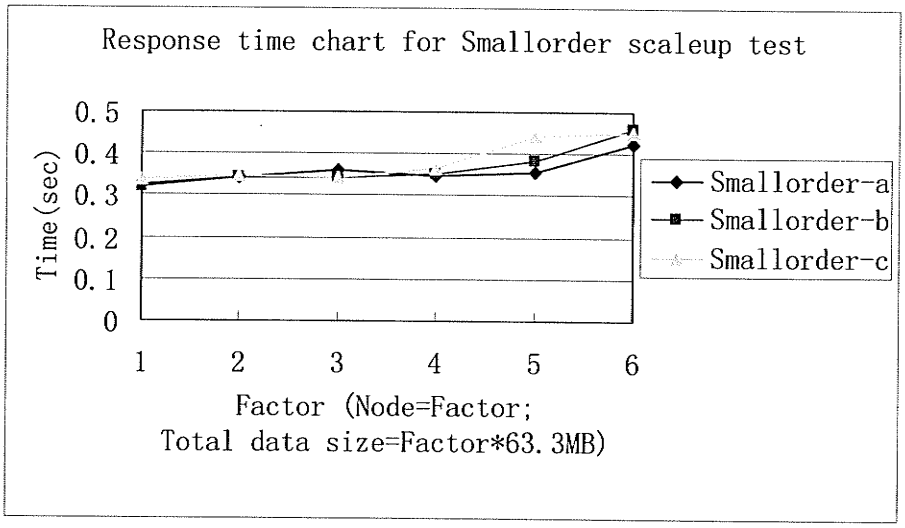


Figure 5-15 Response Time Chart for the Smallorder Scaleup Test

Figure 5-16 shows that sub-linear scaleups were achieved for orders have small number of distinct books.

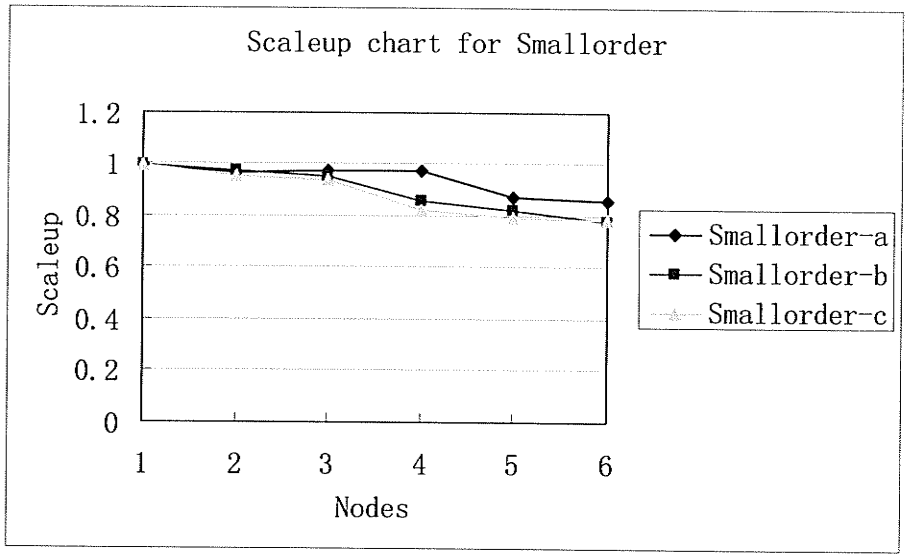


Figure 5-16 Scaleup Chart for the Smallorder Scaleup Test

When the involving nodes increased, the communication cost increased. Hence, $T_{par\text{-}update}$ increased as well. Because I used a local network, $T_{b1},\ldots T_{bn}$ were very small, and were ignored. Moreover, when the order was small, $T_{par\text{-}update}$ took longer than

$T_{seq\text{-}update}$. (i.e., *not* lead to high scaleup). Figure 5-16 shows such a result. However, in reality, the communication with banks $T_{b1},...T_{bn}$ was long, and was much greater than $T_{par\text{-}update}$ or $T_{seq\text{-}update}$ for small orders. So,

$$T_{seq} = T_{seq\text{-}update} + (T_{b1}+...+T_{bn}) + T_{order} \approx T_{b1}+...+T_{bn}$$

and

$$T_{par} = \text{Max}(T_{par\text{-}update}, T_{b1},...T_{bn}, T_{order}) \approx \text{Max}(T_{b1},...T_{bn}).$$

Compared to the sequential processed order, the parallel processed order used less time. However, the scaleup was *not* high because using more nodes did not affect the processing time. Thus, the number of transactions processed per minute did not increase when the involving nodes increased.

5.4.2 Large Orders

Similar to the queries used for small orders, I also used three transaction queries with one deposit bank and varied number of withdrawal banks (where Queries Largeorder-a, -b, & -c uses one, two, & three withdrawal banks respectively) for large orders. Table 5-7 shows different banks involved in each transaction. In each transaction in the test, 6060 distinct books were ordered.

Table 5-7 Banks Involved in Different Queries in Largeorder

Query	Withdraw bank No.	Deposit bank No
Largeorder-a	1	1
Largeorder-b	2	1
Largeorder-c	3	1

Figure 5-17 shows the run-time for different queries in the speedup test for Largeorder.

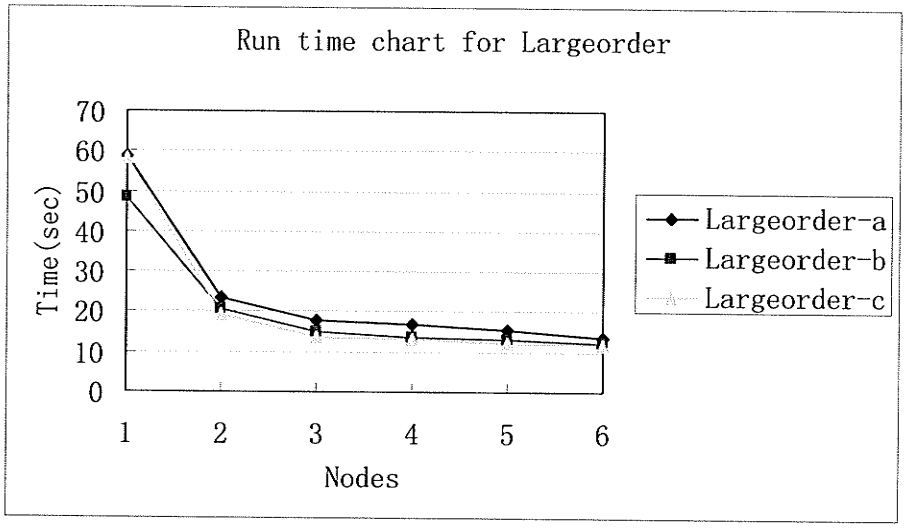


Figure 5-17 Run-time Chart for the Largeorder Speedup Test

For large order, the communication cost with banks can be ignored. Hence,

$$T_{seq} = T_{seq\text{-}update} + (T_{b1}+\ldots+T_{bn}) + T_{order} \approx T_{seq\text{-}update} + T_{order}$$

and

$$T_{par} = \text{Max}(T_{par\text{-}update}, T_{b1},\ldots T_{bn}, T_{order}) \approx \text{Max}(T_{par\text{-}update}, T_{order}).$$

Because in my implementation, the Order table was not partitioned and distributed, T_{order} remained the same and no communication was involved. Moreover, in most cases, $T_{par\text{-}update} > T_{order}$. So,

$$T_{par} = \text{Max}(T_{par\text{-}update}, T_{b1},\ldots T_{bn}, T_{order}) \approx \text{Max}(T_{par\text{-}update}, T_{order}) \approx T_{par\text{-}update}.$$

In the large-order speedup test, when the number of nodes increased, T_{comm} increased and $T_{seq\text{-}update} / n$ decreased. Since the decrease in $T_{seq\text{-}update} / n$ was more than the increase in T_{comm}, a sub-linear speedup was achieved (as shown in Figure 5-18) for orders having large number of distinct books.

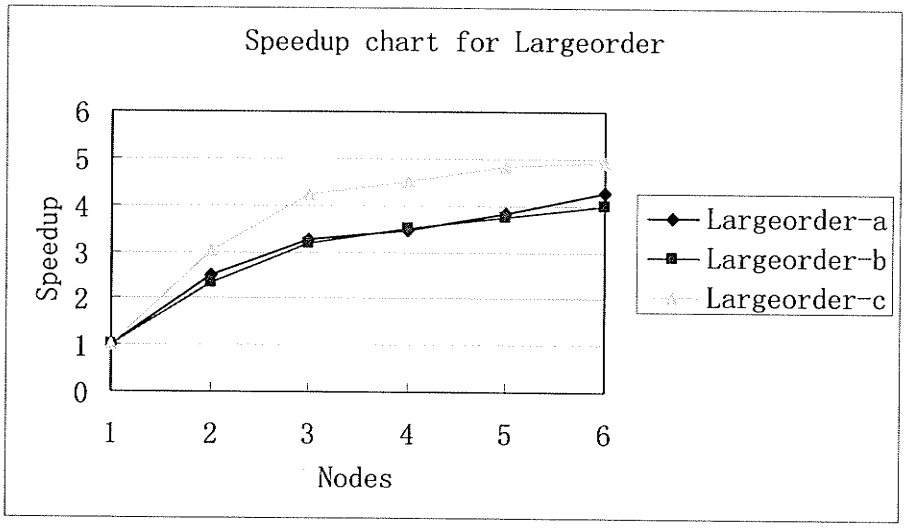


Figure 5-18 Relative Speedup Chart for the Largeorder Speedup Test

Figure 5-19 shows how many order transactions can be processed per minute for different queries in the Largeorder's transaction volume scaleup test. Figure 5-20 shows the response times for the response time scaleup for different queries in the Largeorder's response time scaleup test.

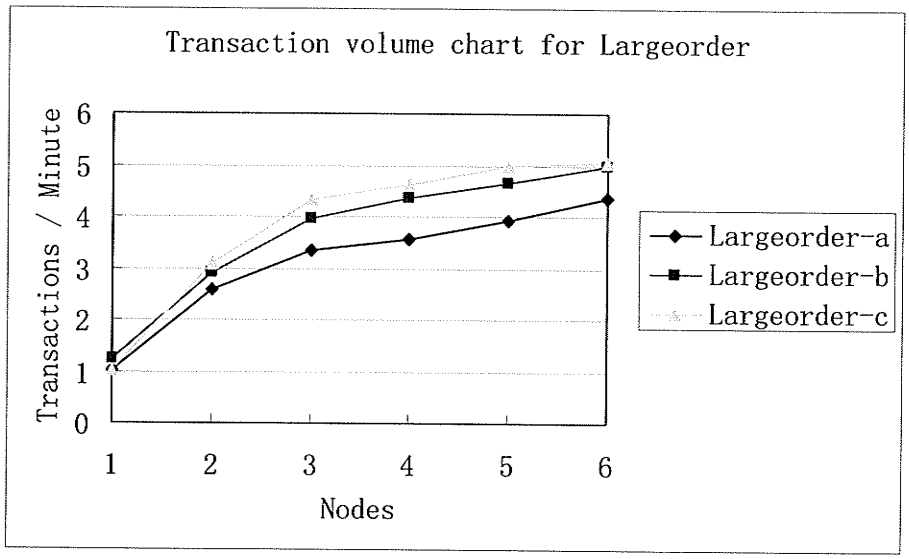


Figure 5-19 Transaction Volume Chart for the Largeorder Scaleup Test

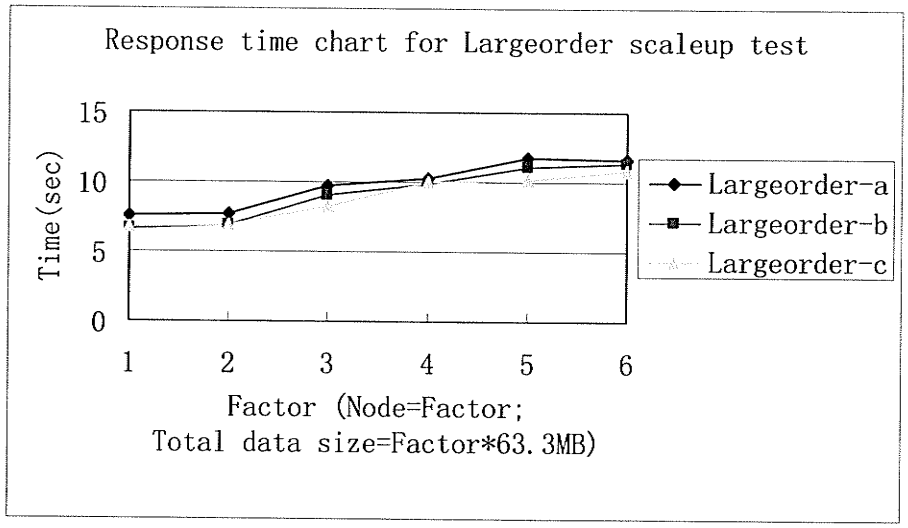


Figure 5-20 Response Time Chart for the Largeorder Scaleup Test

Figure 5-21 shows the scaleup for different queries in the scaleup test for Largeorder. We can see that sub-linear scaleup has been achieved for orders have large number of distinct books.

Here, when the number of nodes increased, T_{comm} increased and $T_{seq\text{-}update} / n$ decreased. Since the decrease in $T_{seq\text{-}update} / n$ was more than the increase in T_{comm}, the number of order transactions processed per minute increased. As shown in Figure 5-21, a sub-linear speedup was achieved.

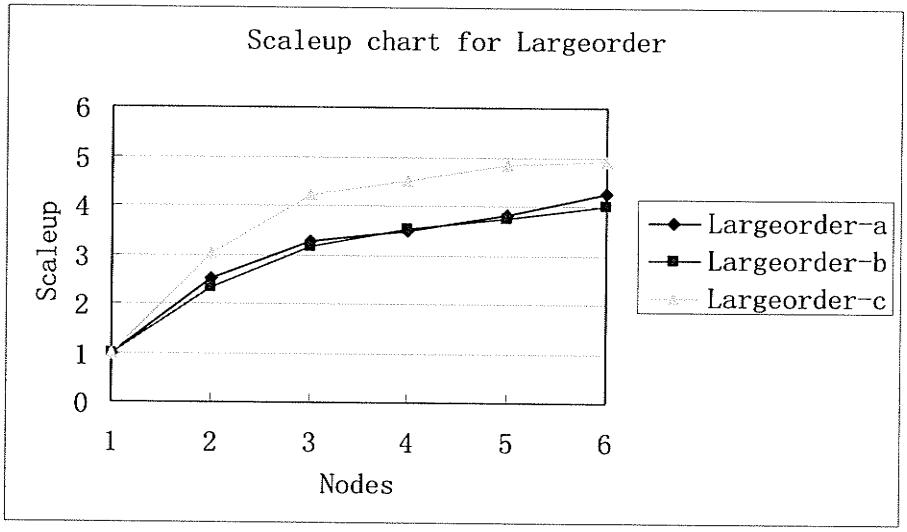


Figure 5-21 Scaleup Chart for the Largeorder Scaleup Test

5.5 Summary

In this chapter, I described and analyzed experimental results of my prototype e-commerce system. The results showed that my system provided good speed-up and scale-up. In other words, the time taken for processing transactions decreased when the number of processors increased. Moreover, the system sustained the runtime per transaction.

Chapter 6 Conclusions

The wide acceptance and use of the World Wide Web has significantly expanded the horizons of commerce, and has changed the face of commercial transactions we used to know to a new form of transactions (namely, e-commerce transactions). An e-commerce transaction processing system, which processes e-commerce transactions, requires high throughput and high performance. Traditional sequential transaction processing techniques fails to meet these e-commerce system requirements. However, parallel processing techniques could be used to deal with the demands of e-commerce system. In this thesis, our goal of this thesis was to (a) investigate typical e-commerce transactions, (b) identify the aspects in those transactions that could benefit from parallel processing, (c) apply parallel processing techniques suitable for those transactions to e-commerce system, and (d) provide a reliable, flexible, and scalable e-commerce transaction system design.

6.1 Summary of Contributions

In this thesis, I analyzed some typical e-commerce transactions such as the search, compare, payment, and order transactions. Each of these transactions can be benefited by applying parallel processing techniques in their implementations. Each of the transactions was analyzed using the UML activity diagram to isolate opportunities that are amenable to parallel processing techniques.

Furthermore, a detail design of a prototype e-commerce transaction processing system using parallel techniques in the processing of transactions was provided. My

system design used typical three-tiered architecture, which is very reliable and flexible. In the database tier of the system, parallel databases are used to store persistent data, which could provide good scalability. In the business logic tier, multithread techniques are used to access data stored in different e-commerce systems. The use of parallel databases and multithread techniques ensures higher performance and higher throughput.

Finally, I implemented a prototype e-commerce system that applies parallel processing techniques designed in this thesis. The system was implemented based on CORBA architecture. Performance test results of the prototype e-commerce system showed that applying parallel processing techniques results in better performance and higher throughput than using the sequential processing techniques.

6.2 Future work

In this thesis, I discussed how to apply parallel processing techniques in e-commerce systems to achieve higher performance. In the future, several things need to be investigated in detail in an e-commerce system design. For example, how to balance the workload? How to make a system fault tolerant? How to ensure Quality of Service (QoS)? *Load balance* is a measure of how to evenly distributed work among a set of parallel processors. Parallel programs are the most efficient when the load is perfectly balanced. If the workload of an e-commerce system is well balanced among processors, higher performance can be achieved. *Fault tolerance* refers to the ability to continue non-stop when a hardware failure occurs. A fault tolerant system is designed from the ground up for reliability by building multiples of all critical

components such as CPUs, memories, disks and power supplies into a system. If an e-commerce system is fault tolerant, the availability and reliability of the system is ensured. The *QoS* refers to ensure a guaranteed level of performance delivered to the customer; it can be characterized by several basic performance criteria including availability, performance, response time and throughput. In order to keep the QoS of the system high, an e-commerce system that uses parallel processing techniques must be designed with great extensibility.

References

[AFH+99] Gustavo Alonso, Ulrich Fiedler, Claus Hagen, Amaia Lazcano, Heiko Schuldt, and Nathalie Weiler. WISE: Business to Business E-Commerce. In *Proceedings of the 9th RIDE Workshop*, pages 132–139, 1999.

[BFV96] Luc Bouganim, Daniela Florescu, and Patrick Valduriez, Dynamic Load Balancing in Hierarchical Parallel Database Systems. In *Proceedings of the 22nd VLDB Conference*, pages 436-447, 1996.

[BLWW95] Roger W. H. Bons, Ronald M. Lee, René W. Wagenaar, and Clive D. Wrigley. Modelling Inter-Organization Trade Procedures Using Documentary Petri Nets. In *Proceedings of 28th Annual Hawaii International Conference on System Sciences*, pages 189–198, 1995.

[CDTA99] Ibrahim Cingil, Asuman Dogac, Nesime Tatbul, and Sena Arpinar. An Adaptable Workflow System Architecture on the Internet for Electronic Commerce Applications. In *Proceedings of Distributed Objects and Applications (DOA),* pages 242–251, 1999.

[DG92] David J. DeWitt and Jim Gray. Parallel database systems: The Future of High Performance Database Systems. *Communications of the ACM*, 35(6):85–98, 1992.

[DDA+98] Asuman Dogac, Ilker Durusoy, Sena Nural Arpinar, Nesime Tatbul, Pinar Koskal, Ibrahim Cingil, and Nazife Dimililer. A Workflow-Based Electronic Marketplace on the Web. *SIGMOD Record*, 27(4):25–31, 1998.

[Ehi97] Sylvanus A. Ehikioya. *Specification of Transaction Systems Protocols.* PhD thesis, Department of Computer Science, University of Manitoba, Winnipeg, Manitoba, Canada, September 1997.

[Ehi01] Sylvanus A. Ehikioya. A Formal Characterization of Electronic Commerce Transactions. *International Journal of Computer & Information Science*, 2(3):97–117, 2001.

[Fur04] Pedro Furtado. Workload-based Placement and Join Processing in Node-partitioned Data Warehouses. In *Proceedings of the 6th DaWaK Conference*, pages 38-47, 2004.

[GC97] Sanjay Goil and Alok N. Choudhary. High Performance OLAP and Data Mining on Parallel Computers. *Data Mining and Knowledge Discovery*, 1(4):391–417, 1997.

[GC99] Sanjay Goil and Alok N. Choudhary. A Parallel Scalable Infrastructure for OLAP and Data Mining. In *Proceedings of International Database Engineering and Applications Symposium*, pages 178–186, 1999.

[GR92] Jim Gray and Andreas Reuter. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann Publishers, San Francisco, California, 1992.

[HL00] Chen Hao and Chengwen Liu. An Efficient Algorithm for Processing Distributed Queries Using Partition Dependency. In *Proceedings of International Conference on Parallel and Distributed Systems*, pages 339–346, 2000.

[KPAG99] Farrukh Khan, Ray Paul, Ishfaq Ahmad, and Arif Ghafoor. Intensive Data Management in Parallel Systems: A Survey. *Distributed and Parallel Databases*, 7(4):383-414, October 1999.

[Law97] Peter Lawrence. *Workflow Handbook 1997*. Wiley Europe, Chichester, England, 1997.

[LCK96] Chengwen Liu, Hao Chen, and Warren Krueger. A Distributed Query Processing Strategy Using Placement Dependency. In *Proceedings of 12th ICDE Conference*, pages 477–484, 1996.

[MD97] Manish Mehta and David J. DeWitt. Data Placement in Shared-nothing Parallel Database Systems. *VLDB Journal*, 6(1):53–72, 1997.

[MOW97] Stefan Manegold, Johann K. Obermaier, and Florian Waas. Load Balanced Query Evaluation in Shared-everything Environments. In *Proceedings of European Conference on Parallel Processing*, pages 1117-1124, 1997.

[Obj98] Object Management Group. *The Common Object Request Broker: Architecture and Specification*, 1998.

[RHN05] Vijayshankar Raman, Wei Han, and Inderpal Narang. Parallel Querying with Non-dedicated Computers. In *Proceedings of the 31st VLDB Conference*, pages 61–72, 2005.

[RM95] Erhard Rahm and Robert Marek. Dynamic Multi-resource Load Balancing in Parallel Database Systems. In *Proceedings of the 21st VLDB conference*, pages 395-406, 1995.

[RNS96] Michael Rys, Moira C. Norrie, and Hans-Jörg Schek. Intra-transaction Parallelism in the Mapping of an Object Model to a Relational Multi-processor System. In *Proceedings of the 22nd VLDB Conference,* pages 460–471, 1996.

[Saa99] Mark Saaltink. *The Z/EVES 2.0 User's Guide*. ORA Canada Technical Report TR-99-5493-06a, October 1999.

[SABS02] Heiko Schuldt, Gustavo Alonso, Catriel Beeri, and Hans-Jörg Schek. Atomicity and Isolation for Transactional Processes. *ACM Transactions on Database Systems*, 27(1):63–116, 2002.

[Spi92] J. Michael Spivey. *The Z Notation: A Reference Manual, 2nd Edition.* Prentice Hall International, Upper Saddle River, New Jersey, 1992.

[TV02] Andrew S. Tanenbuam and Maarten van Steen. *Distributed Systems: Principles and Paradigms*. Prentice Hall International, Upper Saddle River, New Jersey, 2002.

[WB97] Jeffrey L. Whitten and Lonnie D. Bentley. *Systems Analysis and Design Methods, 4th Edition*. Irwin McGraw-Hill, Columbus, OH, 1997.

[WVD98] Hans Weigand, Willem-Jan van den Heuvel, and Frank Dignum. Modelling Electronic Commerce Transactions - A Layered Approach. In *Proceedings of the 3rd International LAP Workshop — The Language-Action Perspective on Communication Modelling*, pages 47–58, 1998.

[WTCY94] Joel L. Wolf, John Turek, Ming-Syan Chen, and Philip S. Yu. Scheduling Multiple Queries on a Parallel Machine. In *Proceedings of the 1994 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, pages 45–55, 1994.

[YM98] Clement T. Yu and Weiyi Meng. *Principles of Database Query Processing for Advanced Applications*. Morgan Kaufmann, 1998.

[ZCRS05] Jingren Zhou, John Cieslewicz, Kenneth A. Ross, and Mihir Shah. Improving Database Performance on Simultaneous Multithreading Processors. In *Proceedings of the 31st VLDB Conference*, pages 49–60, 2005.

[Zom96] Albert Y. Zomaya (Editor). *Parallel and Distributed Computing Handbook*. McGraw-Hill, New York, USA, 1996.

Appendix A Performance Test Result for Search and Comparison Based on Two and Three Criteria

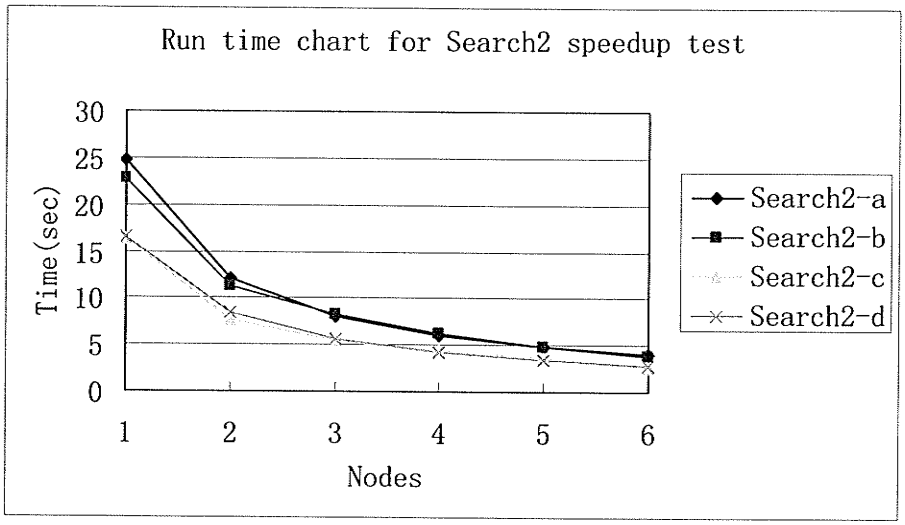


Figure A-1 Run-time Chart for the Search2 Speedup Test

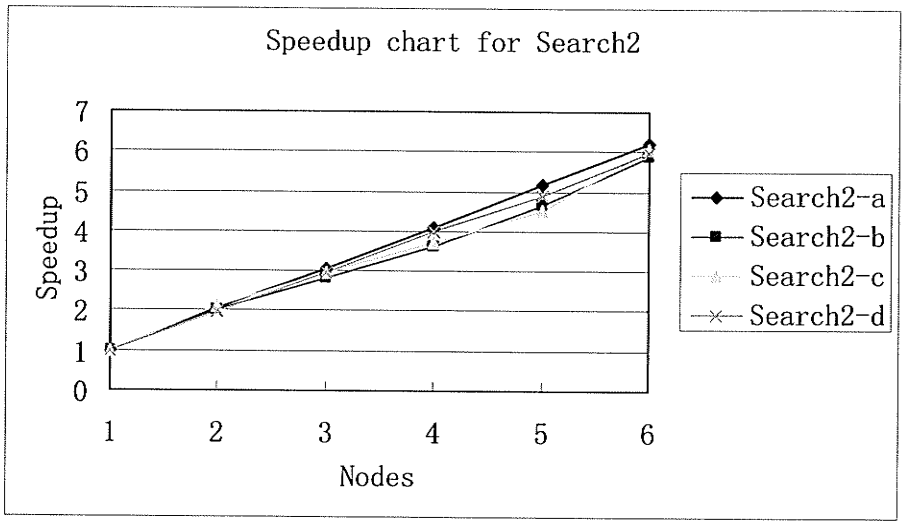


Figure A-2 Relative Speedup Chart for the Search2 Speedup Test

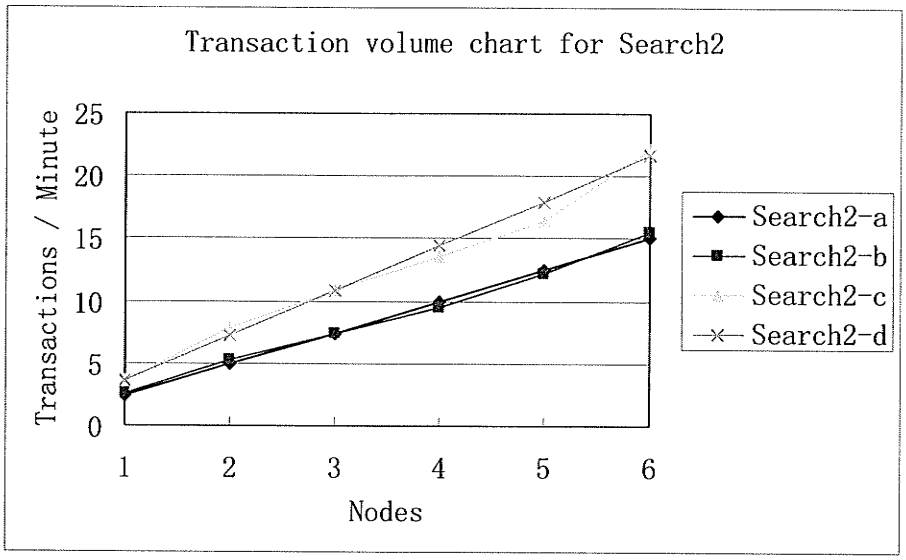


Figure A-3 Transaction Volume Chart for the Search2 Scaleup Test

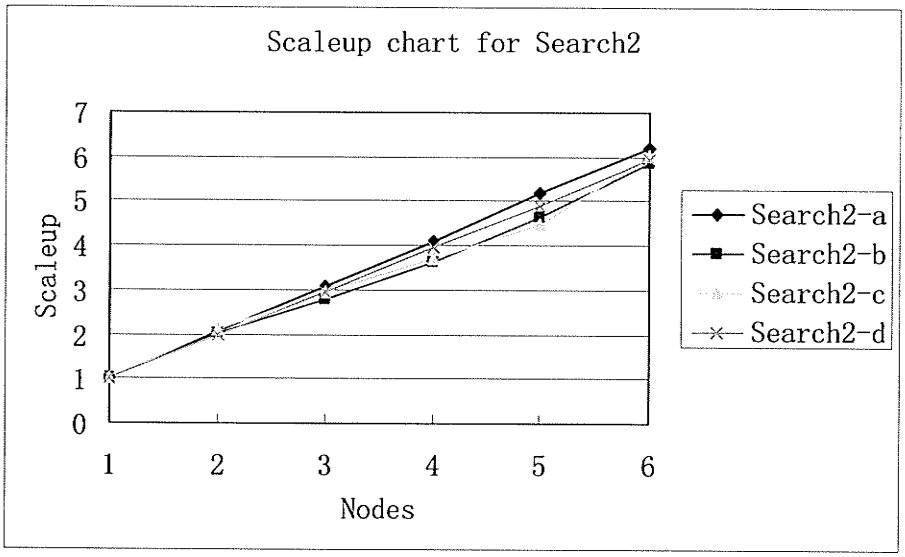


Figure A-4 Scaleup Chart for the Search2 Scaleup Test

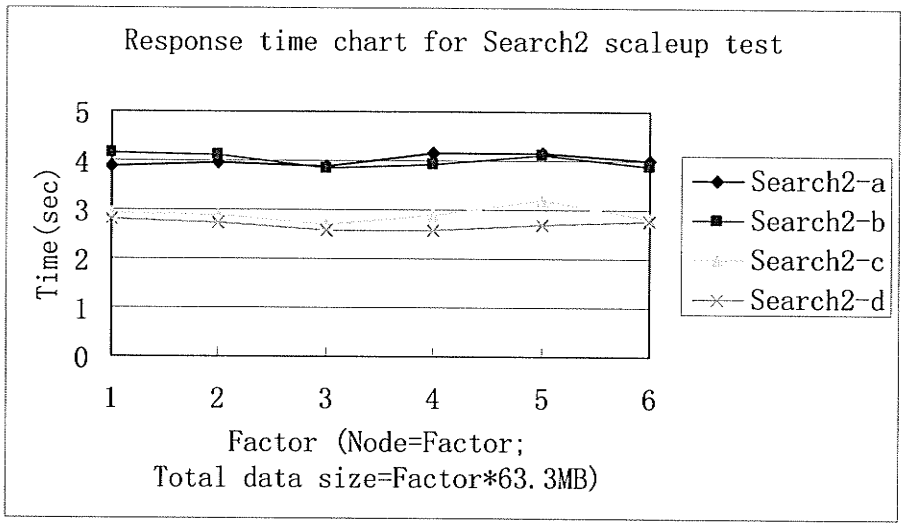


Figure A-5 Response Time Chart for the Search2 Scaleup Test

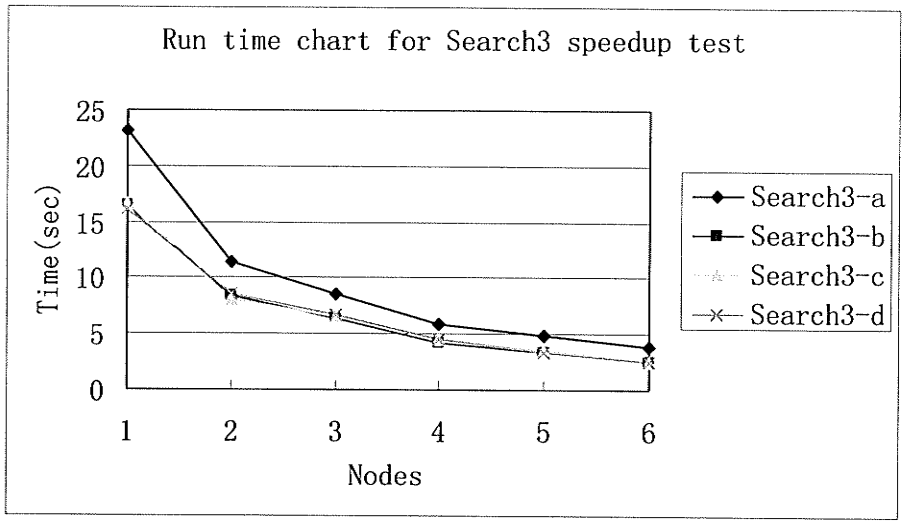


Figure A-6 Run-time Chart for the Search3 Speedup Test

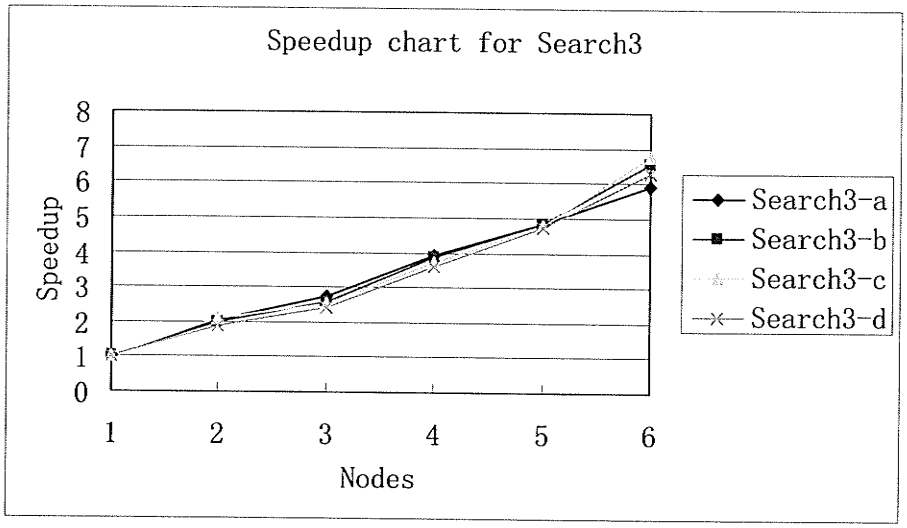


Figure A-7 Relative Speedup Chart for the Search3 Speedup Test

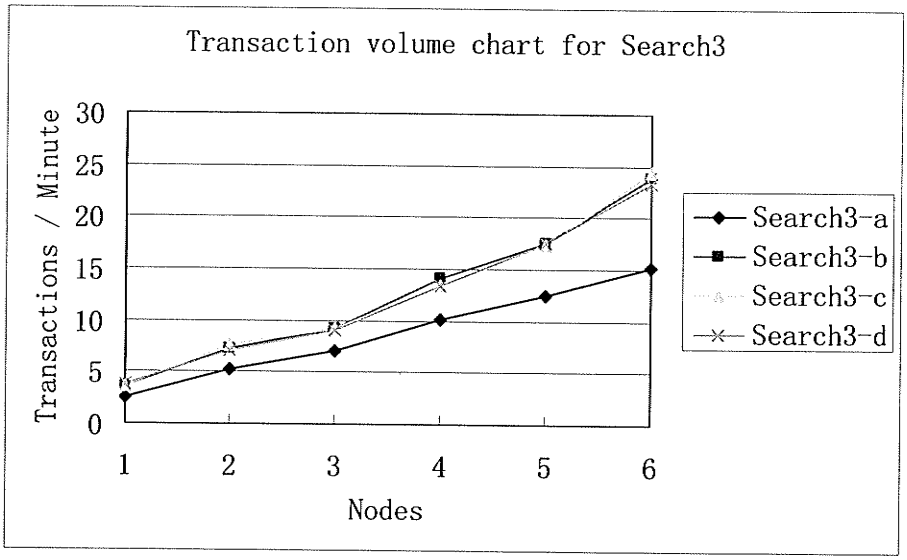


Figure A-8 Transaction Volume Chart for the Search3 Scaleup Test

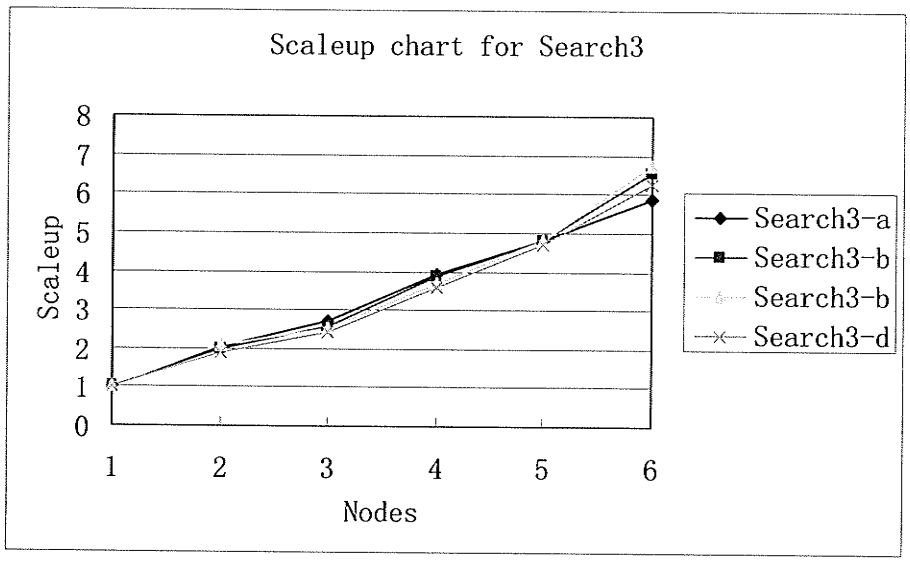


Figure A-9 Scaleup Chart for the Search3 Scaleup Test

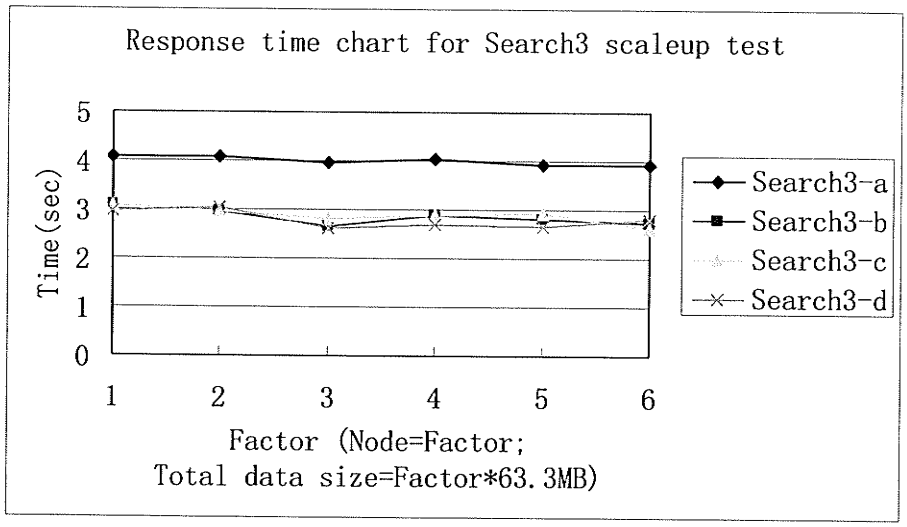


Figure A-10 Response Time Chart for the Search3 Scaleup Test

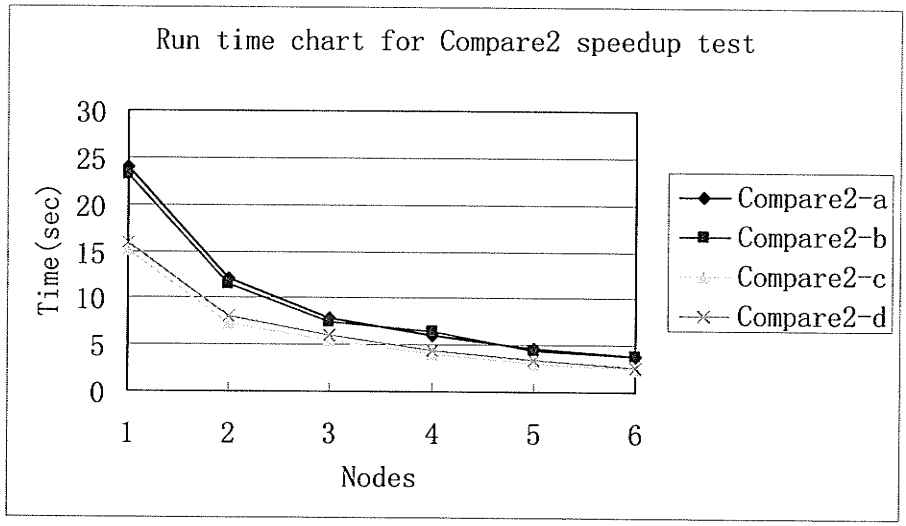


Figure A-11 Run-time Chart for the Compare2 Speedup Test

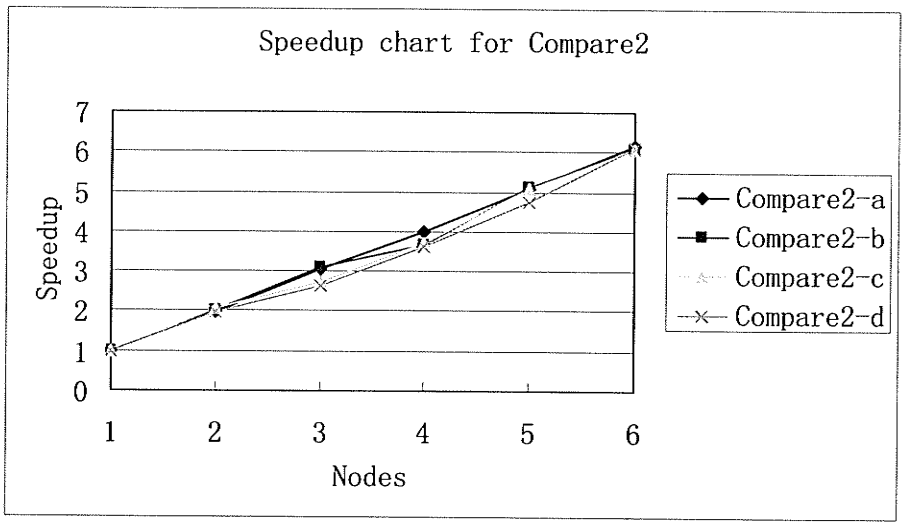


Figure A-12 Relative Speedup Chart for the Compare2 Speedup Test

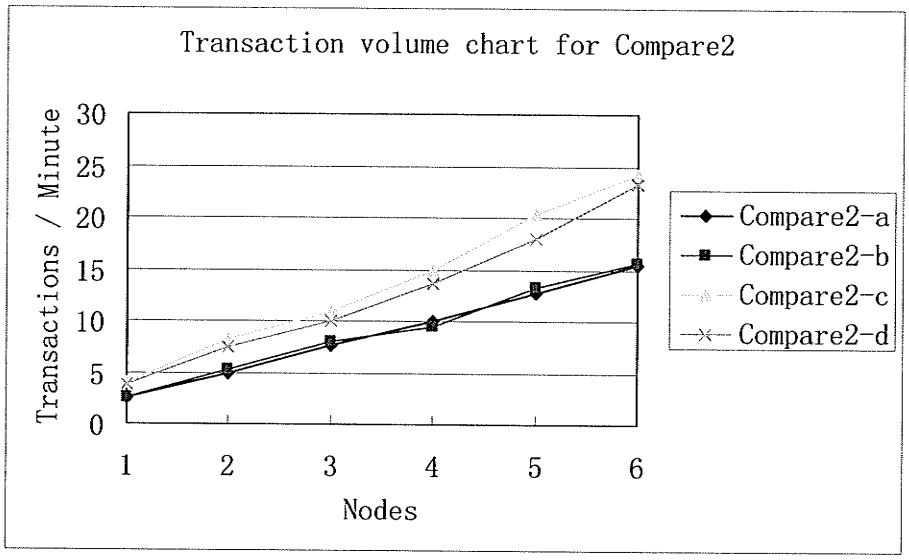


Figure A-13 Transaction Volume Chart for the Compare2 Scaleup Test

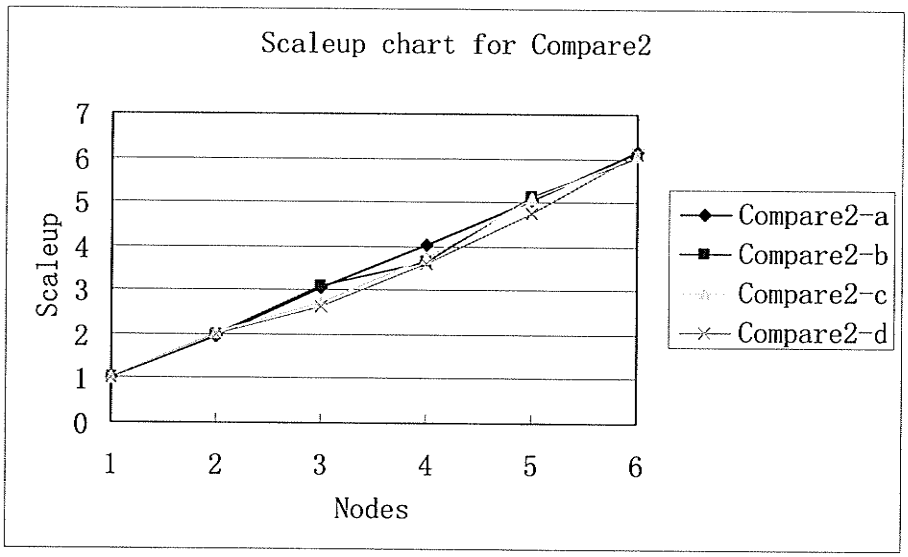


Figure A-14 Scaleup Chart for the Compare2 Scaleup Test

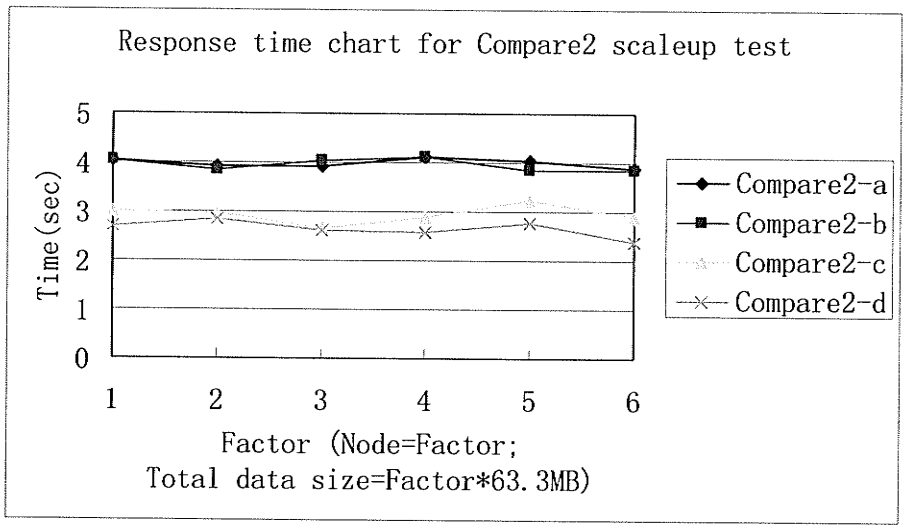


Figure A-15 Response Time Chart for the Compare2 Scaleup Test

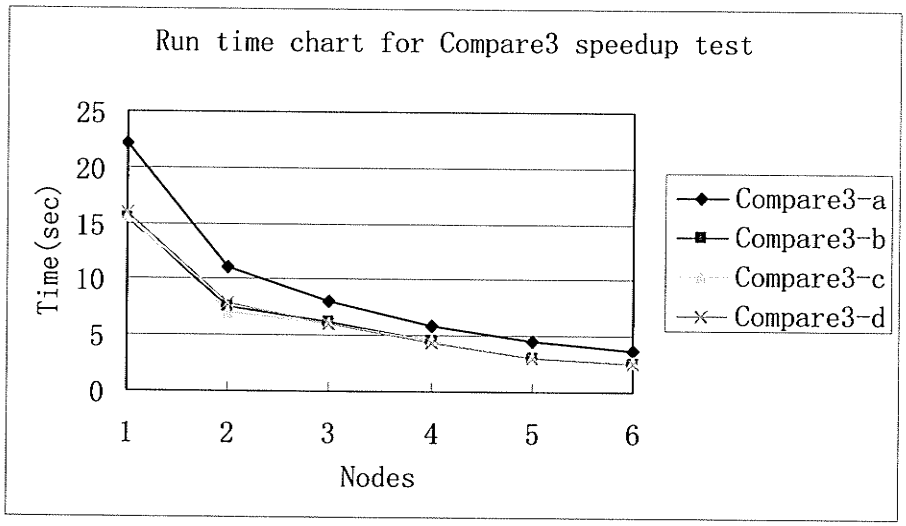


Figure A-16 Run-time Chart for the Compare3 Speedup Test

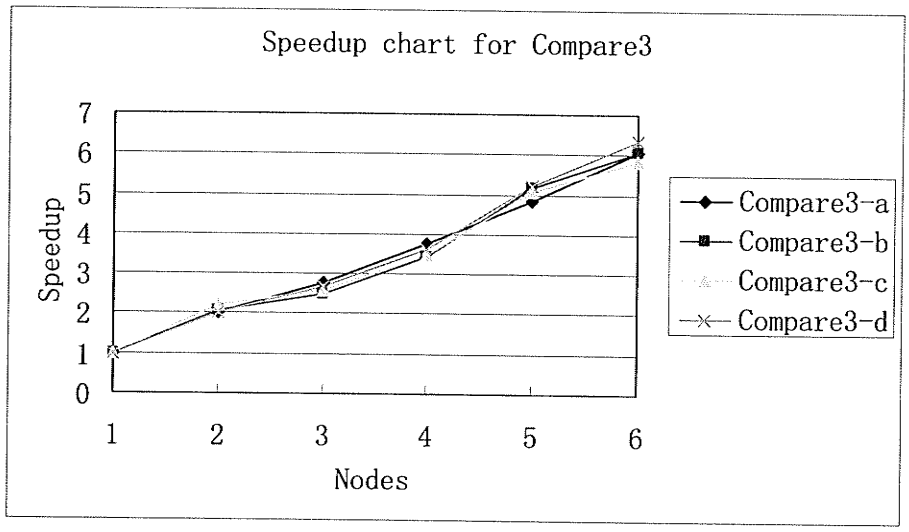


Figure A-17 Relative Speedup Chart for the Compare3 Speedup Test

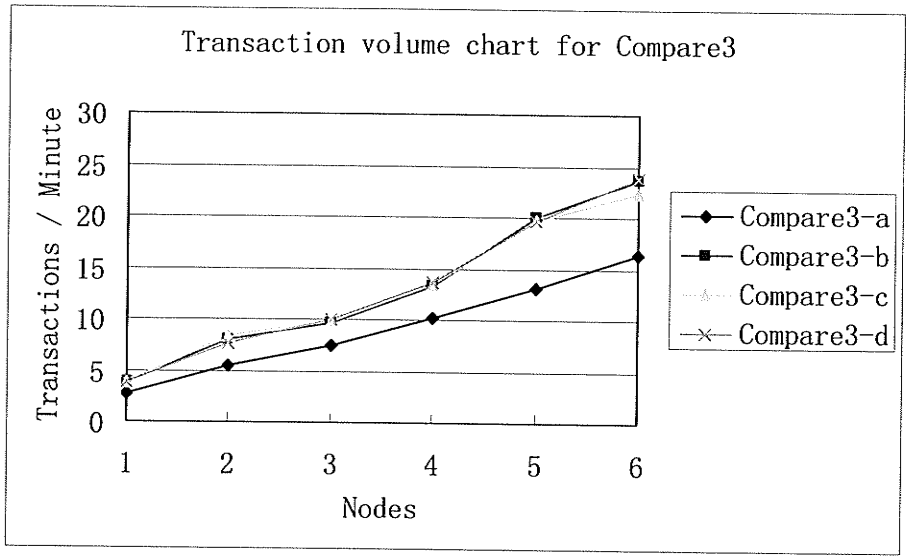


Figure A-18 Transaction Volume Chart for the Compare3 Scaleup Test

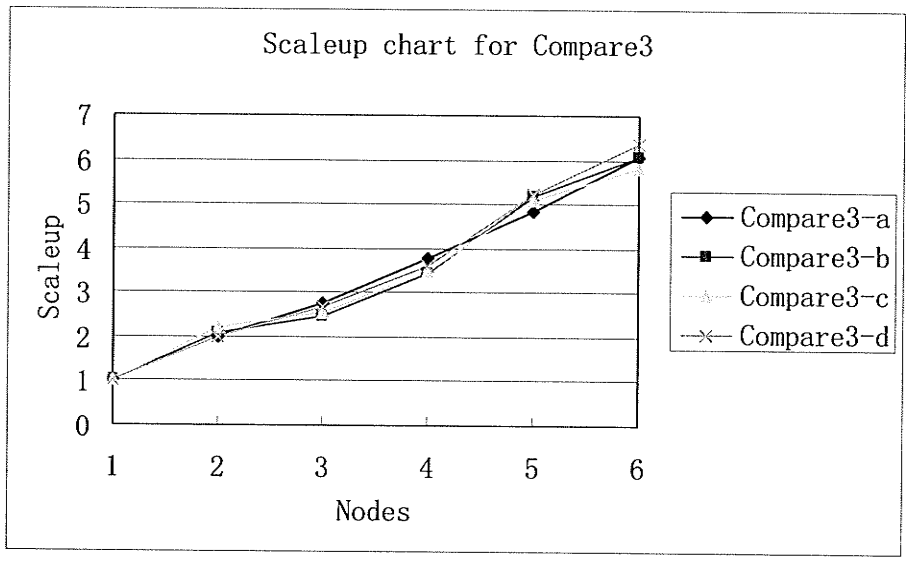
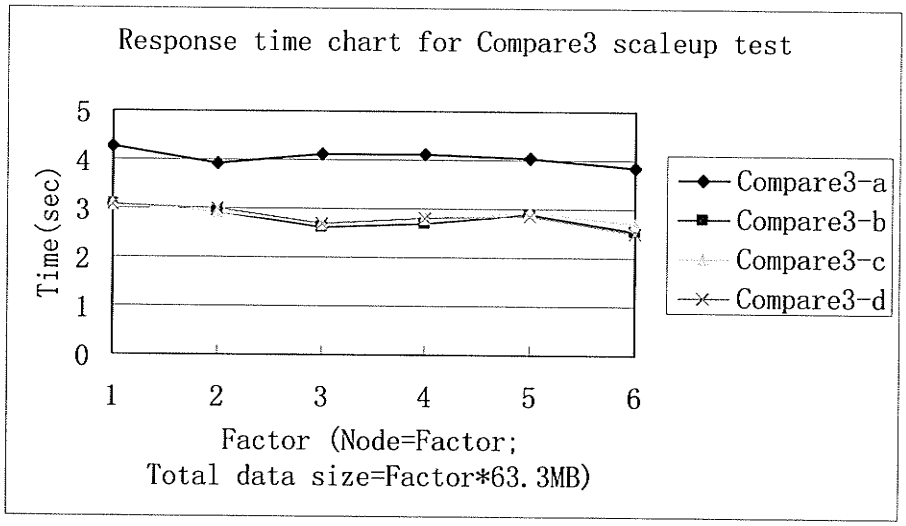


Figure A-19 Scaleup Chart for the Compare3 Scaleup Test



A-20 Response Time Chart for the Compare3 Scaleup Test