A Tuple-space Based Framework for Solving Large Highly Parallel Problems

Submitted in partial fulfillment of the requirements in CS 5204

Submitted by:
Amit Karnik (akarnik@vt.edu)
Roy Patrick Tan (rtan@vt.edu)

Submitted to:
Dr. Dennis Kafura

December 7, 2000

ABSTRACT

This project implements a tuple-space inspired system for solving parallel problems. The project used matrix multiplication as a specific problem to be solved using this implementation. Java was used as the language for implementation, with RMI and UDP multicasts as the method for communication between distributed processes. In doing this project we wished to learn about issues related to computing in a distributed environment, including data distribution, load balancing, and fault tolerance. We also wished to learn about the underlying technologies such as RMI and UDP multicasting.
I. Introduction

The architecture of our system is inspired by the tuple-space idea. However, it is not exactly the same. For example, parts of the data may be actively distributed through a multicast mechanism, rather than passively stored.

Although the program, as implemented, may be modified and extended to solve many different problems, we have concentrated on solving a specific problem: matrix multiplication.

Our system is composed of three parts: a client, a “tuple-space” server (called a TServer), and one or more computing servers (CServers)

II. Underlying Mechanism

Because our system is not purely a tuple space server, we implemented our own server, using RMI as the basic communication protocol. We use a multicast mechanism, using UDP for data distribution. All the code is written in Java.

The matrix multiplication algorithm used is a straightforward one, each computing server, given a row and a matrix, returns a row back. Thus the matrix multiplication granularity is at the row level.

III. Data Flow Diagram

The flow of data in the system can be visualized as follows:
The client sends two matrices to the tuple space server, and waits for the answer to be returned. The two matrices are encapsulated in an object that is marked as a “matrix multiplication” problem, and this object is sent to the tuple space server through an RMI call.

On the other side, the computing servers register themselves with the tuple space server through RMI, and wait for data to be distributed.

The tuple space server, after getting the data from the client, and upon identifying the problem data as “matrix multiplication”, breaks down the left matrix into several tuples, each tuple representing a row. It then sends the right matrix simultaneously, through a UDP multicast, to all the computing servers currently registered with it.

The computing servers then request for tuples from the tuple space server. The computing server computes a row of the answer matrix, and sends this row back to the tuple space server. The tuple server then compiles all the answers into one matrix, and once all rows have been computed, returns the answer back to the client.

### IV. Control Flow Diagram
The flow of control in the system can be visualized as follows:

1- TupleSpace Server starts up.
2-Computing Server starts up.
3-Computing Server registers with the Tuple-Space server.
4-Tuple-Space server asks Computing Server to wait for data.
5-Client starts up.
6-Client gives problem data to Tuple-Space server.
7-Tuple-Space server gives partial problems to Computing server.
8-Computing Server returns partial result.
9-Tuple-Space server returns the solution to the Client.

V. Classes
The implementation involves numerous classes, including small classes for representing the tuples, and classes with methods for solving matrix multiplication. However, there are five important classes that form the core of the system. These are:

1. **Client**
The Client class basically sends the problem (in our case the matrices to be multiplied) to the ClientAcceptor class, which resides on the TServer. The client also receives the answer matrix back from the ClientAcceptor class.

2. **ClientAcceptor**
The ClientAcceptor class waits for connections from the client. After receiving the connection, the ClientAcceptor accepts the problem (the matrices) from the client. It then passes the data to the TServer class.

3. **CServer**
The CServer class stands for the Computing Server. This class is activated when the computing server comes up. It connects with the ServerAcceptor class, which resides on the TServer and registers itself with it. The CServer class then accepts the problem data from both the ServerAcceptor and the TServer. It then computes the result of the problem assigned to it and returns the result to the ServerAcceptor.

4. **ServerAcceptor**
The ServerAcceptor class waits for connections from the CServer (computing server). After receiving a connection from the CServer, it registers the CServer on the TServer and asks it to wait for the problem data.

5. **TServer**
The TServer is a repository. It stores the data for the registered CServers as well as the data for the problem. The TServer broadcasts the second matrix and breaks down the first matrix into rows. These rows are introduced as tuples in the tuple-space. The TServer also has the responsibility to give out the tuples to the CServer through the ServerAcceptor and get back the partial results from the CServer again through the ServerAcceptor. During the computation of the problem, the TServer maintains a degree of fault tolerance by periodically broadcasting a heartbeat, which in turn is responded by the CServers that are alive. After the computation is finished the TServer gives back the solution to the Client through the ClientAcceptor.
VI. Broad Level Algorithms for the Classes

1. **Client**
   a. Generate test data
   b. Make connection with ClientAcceptor class on TServer.
   c. Send data to ClientAcceptor.
   d. Wait for results.
   e. Terminate

2. **ClientAcceptor**
   a. Waits for connection from a Client.
   b. Accepts problem data from the Client.
   c. Send the data to the TServer.
   d. Receives solution from the TServer.
   e. Returns the solution to the Client.

3. **CServer**
   **Main Thread**
   a. Connects to the ServerAcceptor class on TServer on start-up.
   b. Registers itself with the TServer through the ServerAcceptor.
   c. Spawns a different thread (BroadcastListener)
   d. Waits for data form ServerAcceptor.
   e. Computes the partial result for the problem data it has.
   f. Send the partial result to the ServerAcceptor.
   g. Loops to d until no new data from ServerAcceptor.

   **BroadcastListener Thread**
   a. Waits for the data broadcast.
   b. Stores this data locally for future computations.
   c. Waits for ‘Heartbeat’ message from TServer.
   e. Loops to c until it receives a ‘Stop’ message from the TServer.

4. **ServerAcceptor**
   a. Waits for connection from CServer
   b. Registers CServer to the TServer
   c. Requests for TServer to send tuple.
   d. Return tuple to CServer.
   e. Waits for CServer to return partial result.
   f. Returns result to TServer.
   g. Loops to c until program terminates.

5. **TServer**
   **Client Thread**
   a. Accepts data from ClientAcceptor
   b. Stores the data.
   c. Starts Heartbeat thread.
d. Waits for results to be completed

e. Sends result to the Client.

**Heartbeat Thread**

a. Notify Server thread that broadcast is starting
b. Broadcasts problem data to C Servers
c. Notify server thread that broadcast is done
d. Send heartbeat to CServer
e. Loops to b until results are completed.
f. Send stop message to C Servers.

**Server Thread**

a. Register C Servers through ServerAcceptor
b. Do (a) until Broadcast thread notifies that broadcast is starting
c. Wait for Broadcast to be done.
d. Send data to C Servers through ServerAcceptor
e. Wait for partial result.
f. Store result.
g. Loops to d until results are completed

---

**VII. Summary**

**A. Issues we had to deal with**

In the course of implementing the project, we have had to face some rather interesting issues, which we present here.

**1. Reliable broadcast over the LAN**

Broadcasting data was one of the most challenging part of the project. We faced two major hurdles: first, the broadcast protocol, being UDP, is unreliable, and lost packets were common. Second, the broadcast was slower than what we expected.

To resolve the first issue, we implemented a protocol that required the C Servers to acknowledge packets, before the next packet was sent. This is not an extremely robust protocol, but it allowed us to determine that some computers in the lab had a more unreliable network link than the others.

We determined that packet size was a dominant factor in the speed issue. We found that larger packet sizes improve the speed considerably. By trial and error, we found that for our needs a packet size of 4 kilobytes gave good results.

**2. Fault tolerance using heartbeats**

The heartbeat mechanism was successful in that it allowed detection of failed computing servers, and allowed a level of fault tolerance to the system. This was surprisingly easy to implement. An interesting aspect of the heartbeat mechanism is that if the heartbeat interval is too short, it would trigger false alarms, due to delayed responses from
computing servers. From experimentation, we determined that a 1 second interval is an appropriate choice.

3. Limits to the problem size

In doing our experiments, we found that the system, using the graduate lab computers, could not handle the problem when the size of the matrix is 1,500 x 1,500. A matrix of this size requires approximately 18 megabytes of memory. The system in this scenario did not terminate in a reasonable amount of time.

B. What we have learnt

Although both of us have experience in Java programming, we have not had done any programs using RMI before. This project was a good experience in using the RMI technology in designing and implementing distributed systems.

The knowledge we gained in learning how UDP multicast worked in the Java environment was also valuable. We are certainly more appreciative of the complexity of designing network protocols.

We also encountered the problems that are typical of distributed systems, such as synchronization, and network reliability. Tackling these problems has given us good insight into the workings of distributed systems.

For reference, we have timed the results of the matrix multiplication of 1,000 x 1,000 matrices, and here are the results:

<table>
<thead>
<tr>
<th>Computing servers</th>
<th>Time (ms)</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>193076</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>130686</td>
<td>1.47</td>
</tr>
<tr>
<td>3</td>
<td>110718</td>
<td>1.74</td>
</tr>
<tr>
<td>4</td>
<td>96989</td>
<td>1.99</td>
</tr>
<tr>
<td>5</td>
<td>93794</td>
<td>2.05</td>
</tr>
<tr>
<td>6</td>
<td>82548</td>
<td>2.33</td>
</tr>
</tbody>
</table>

C. Future directions

We believe that many more interesting problems can be solved using the model that we used in the project.

The system could be modified to use data distribution techniques other than broadcast. It would be interesting to have a comparative study of the performance of broadcasted, and not broadcasted data distribution.