Center for Information Systems Security Studies and Research (CISR) Faculty and Researcher Publications

1998-06-02

A Comparison of C++ Sockets and Corba in a Distributed Matrix Multiply Application

Schnaidt, Matt

http://hdl.handle.net/10945/35378

NPS-CS-99-001

NAVAL POSTGRADUATE SCHOOL
Monterey, California

A Comparison of C++ Sockets and Corba in a Distributed Matrix Multiply Application

by

M. Schnaidt
A. Duman
T. Lewis

June 2, 1998

Approved for public release; distribution is unlimited.
Prepared for: Naval Postgraduate School

REPORT DOCUMENTATION PAGE

Form approved

OMB No 0704-0188

1. AGENCY USE ONLY (Leave blank)

2. REPORT DATE
2 JUNE 98

3. REPORT TYPE AND DATES COVERED
TECHNICAL REPORT

4. TITLE AND SUBTITLE
A COMPARISON OF C++ SOCKETS AND CORBA IN A DISTRIBUTED MATRIX MULTIPLY APPLICATION

5. FUNDING

6. AUTHOR(S)
MATT SCHNAIDT, ALPAY DUMAN, and TED LEWIS

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
Naval Postgraduate School, 833 Dyer Road, Monterey, CA 93943

8. PERFORMING ORGANIZATION REPORT NUMBER
NPS-CS-99-001

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)

10. SPONSORING/MONITORING AGENCY REPORT NUMBER

11. SUPPLEMENTARY NOTES

12a. DISTRIBUTION/AVAILABILITY STATEMENT

Approved for public release; distribution is unlimited

12b. DISTRIBUTION CODE

13. ABSTRACT (Maximum 200 words.)

This project has two primary purposes. The first, is to implement a distributed matrix multiply algorithm using C++ sockets, and Corba objects with the objective of discovering what additional overhead, if any, exists in a Corba implementation. Secondly, attempt to improve the speedup through the use of stateful servers in the C++ implementation.

14. SUBJECT TERMS

Buffer Deadlock, Buffer Size Limitations, Client-Server Architecture, C++ Sockets, CORBA, Unix Name Service, Speedup, Efficiency, Superscalar Efficiency.

15. NUMBER OF PAGES 24

16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT
Unclassifed

18. SECURITY CLASSIFICATION OF THIS PAGE
Unclassified

19. SECURITY CLASSIFICATION OF ABSTRACT
Unclassified

20. LIMITATION OF ABSTRACT

NSN 7540-01-280-5800

Standard Form 298 (Rev. 2-89)
Prescribed by ANSI Std 239-18

Enclosure (6)

NAVAL POSTGRADUATE SCHOOL
Monterey, California 93943-5000

RADM Robert C. Chaplin Superintendent	R. Elster Provost

This report was prepared for Naval Postgraduate School.

This report was prepared by:
Matt Schnaidt and Alpay Duman.

Matt Schnaidt Captain US Army	Alpay Duman Lieutenant Jr. Grade Turkish Navy

Ted Lewis
Professor of Computer Science

Reviewed by:	Released by:
Dean D. Boger, Chair Department of Computer Science	D. W.Netzer Associate Provost and Dean of Research

A Comparison of C++ Sockets and Corba in a
Distributed Matrix Multiply Application

2 June 1998

CPT Matt Schnaidt, USA, mcschnai@cs.nps.navy.mil

LTJG Alpay Duman, TN, aduman@cs.nps.navy.mil

ABSTRACT: This project has two primary purposes. The first, is to implement a distributed matrix multiply algorithm using C++ sockets, and Corba objects with the objective of discovering what additional overhead, if any, exists in a Corba implementation. Secondly, attempt to improve the speedup through the use of stateful servers in the C++ implementation.

CONCEPTS:

Buffer Deadlock
Buffer Size Limitations
Client-Server Architecture
C++ Sockets
CORBA
Unix Name Service
CORBA Name Service
Speedup
Efficiency
Superscalar Efficiency

SUMMARY

Matt Schnaidt did the C++ Socket implementation of the matrix multiply. This included writing all of the C++ code (a 2d Array Class, the client and server code for the matrix multiply, and a name server), testing, debugging, and making record runs. I did this in three phases. In phase 1, I implemented a simple matrix multiply using UDP sockets; I used this to debug the multiply algorithm, and the interaction with the name server (which I called memberServer).

In phase 2, I converted the client and server to using TCP sockets so that they could reliably transmit messages which exceeded the maximum packet size without concern for ordering, lost or duplicate packets. I finished phase 2 by measuring the time it took to do matrix multiplies with a varied number of servers and varied size matrices.

In phase 3, I changed the server so that it remembers it state to cut down on message traffic time. I finished phase 3 by measuring the time it took to matrix multiplies, again with a varied number of servers and varied size matrices.

Alpay and I divided the report writing, and slide development for the presentation equally.

Alpay Duman did the CORBA implementation of the matrix multiply. This included writing the IDL, the client code and the interface object implementation defined in the IDL file, testing, debugging and making record runs. I did this in three phases.

In phase one, I implemented the IDL file and the object implementation for this definition. For passing the array I used type sequence in Interface Definition Language, which is a linear dynamic container.

In phase two, I implemented the client invoking the object implementation by using deferred synchronous method, which is a non-blocking dynamic remote procedure call method.

In phase three, I used a special function for CORBA send_multiple_requests_deferred(), which initiates a number of requests in parallel. It does not wait for the requests to finish before returning to the caller.

Table of Contents

1. **Problem Statement**. Develop a distributed matrix multiply algorithm, using C++ sockets and Corba. Compare the performance of the two algorithms in order to discover the overhead associated with the use of Corba.

   Given 2 matrices, A and B, multiply them together to produce a result matrix C. The dimensions of the matrices areA(i X j), B(j X k), C(i X k). Note that the number of columns in A must equal the number of rows in C.

   The general algorithm for solving this problem is:

```
for(int ix = 0; ix <rowInA; ix++){
  for(int jx = 0; jx < j; j++){

    C[i, j] = 0;
    for(int kx = 0; kx < k; k++){
      C[i, j] += A[i, k] * B[k,j];
    }//end for
  }//end for
}//end for
```

2. **Approach**.

A. Distributed Matrix Multiply Algorithm. We agreed, prior to any implementation, on a generic algorithm that we would each implement. One of our primary concerns was that both implementations time the same, or equivalent, events so that we could do a meaningful comparison. Below are listed the steps of our generic algorithm.

- Get the dimensions of the A and B matrices.
- Dynamically allocate space for the A and B matrices.
- Get the filename for the A matrix. Open the file, load the matrix into memory, and close the file.
- Get the filename for the B matrix. Open the file, load the matrix into memory, and close the file.
- Get the number of servers to use.
- START THE TIMER (using gettimeofday() system call).
- Access the name server (must do this explicitly in C++).
- Do the matrix multiply. Send one row fromA, and the entire B matrix to every server, until every row has been distributed.
- Receive the result row of C from each server. Assemble the result matrix.
- STOP THE TIMER.
- Display or write to disk the result matrix.
- Calculate and display the elapsed time.

B. C++ Sockets Implementation.

In the C++ implementation, there is one memberServer that listens at a fixed port and host (declared as MEMBER_SERVER_PORT and MEMBER_SERVER_HOST in file memberServer.h).

Each matrixClient and matrixServer uses three sockets. One is a udp socket to communicate with the memberServer. The other two are tcp sockets; one is a “server” socket at which it listens to receive messages, and the other is a “client” socket that it uses to send messages (see Figure 1, Figure 2, and example at the end of this section).

There can be many matrixServers, but only one permachine as each listens at a fixed port (declared as MATRIX_SERVER_PORT in memberServer.h).

There can be many matrixClients when using the stateless matrixServers, but only one at a time when using the stateful matrixServer (this will be further explained in section 3b). Each matrixClient listens at a fixed port number (declared as CLIENT_RCV_PORT in memberServer.h).

When a matrixServer starts up, it registers with the memberServer. The memberServer keeps a list of the ip address of every matrixServer that has registered with it. The matrixClient then contacts the matrixServer to get the ip addresses of every matrixServer.

The matrixClient uses these ip addresses together with MATRIX_SERVER_PORT to send the following workRequest to the matrixServer:
Index of result into C matrix, number of columns in A, number of columns in B, a row from matrix A, B matrix.

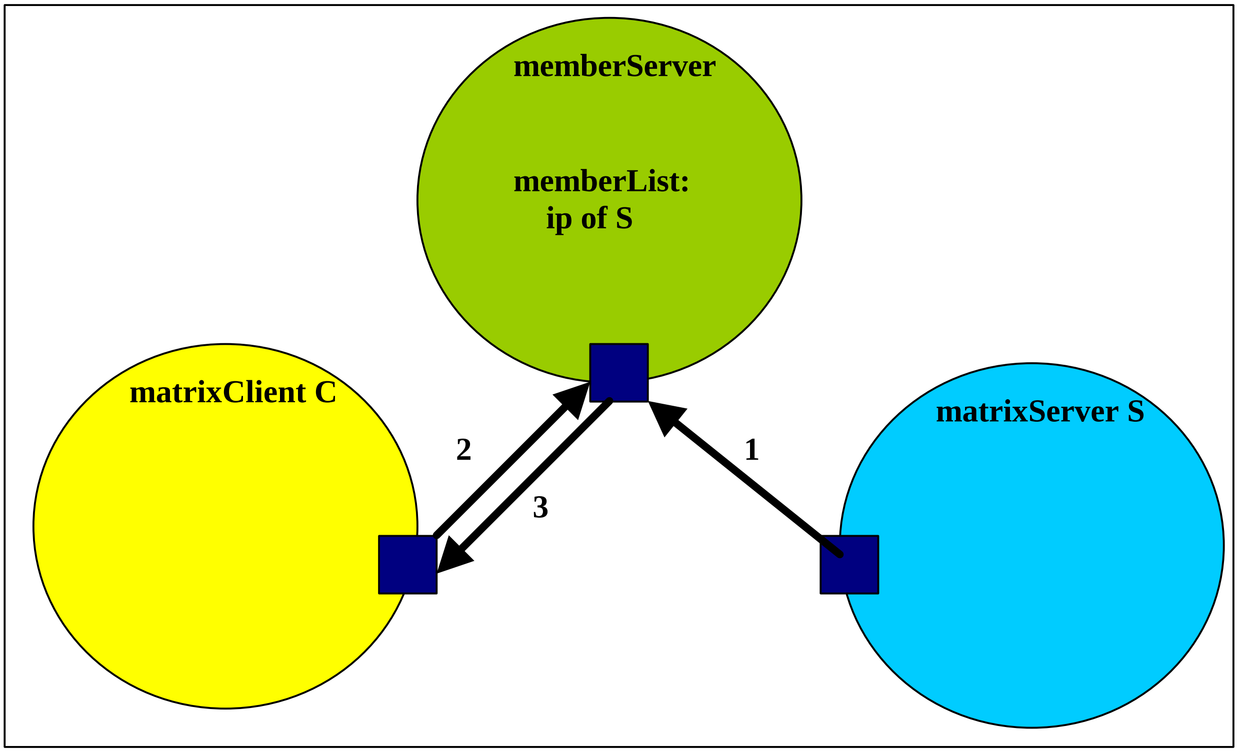

Figure 1: Interaction with memberServer

(1. MatrixServer S registers with memberServer. memberServer records ip of server. 2. matrixClient queries member Server for all matrixServers. 3. memberServer sends ip address of matrixServer S.)

To demonstrate how the distributed algorithm works, consider this example scenario(refer to figure 1 for steps 1 – 6, figure2 for steps 7 – 9):

1) memberServer starts up, listens at udp socket.
2) matrixServer S starts up, registers with memberServer, and listens at its server tcp socket.
3) memberServer adds matrixServer S's ip address to memberList. Steps 2 and 3 are repeated for matrixServer T and U.

4) matrixClient starts up, prompts user for matrix sizes and filenames, loads matrices.
5) matrixClient asks user number of servers to use.
6) matrixClient contacts memberServer. memberServer responds with all ip addresses of matrixServers; memberServer puts this information into a list.
7) matrixClient forks off a child process which will cycle through the list, up to the max number of servers (input by user), connecting frommatrixClient’s client tcp socket and sending work request (format given in preceding paragraph) to each matrixServer. Meanwhile, main process closes the client tcp socket, and listens at matrixClient’s server tcp socket for replies.
8) matrixServers receive work requests from matrixClient on client tcp sockets. Each uses number of columns in A and B to allocate space for the A matrix row and the B matrix, calculates result row C, and connects with matrixClient from matrixServer’s client port and sends result rows and indexes of result row to matrixClient.
9) matrixClient receives all result rows, and fills the C matrix.matrixClient’s child process exits when all requests sent; parent process stops timer once C matrix is fully constructed.

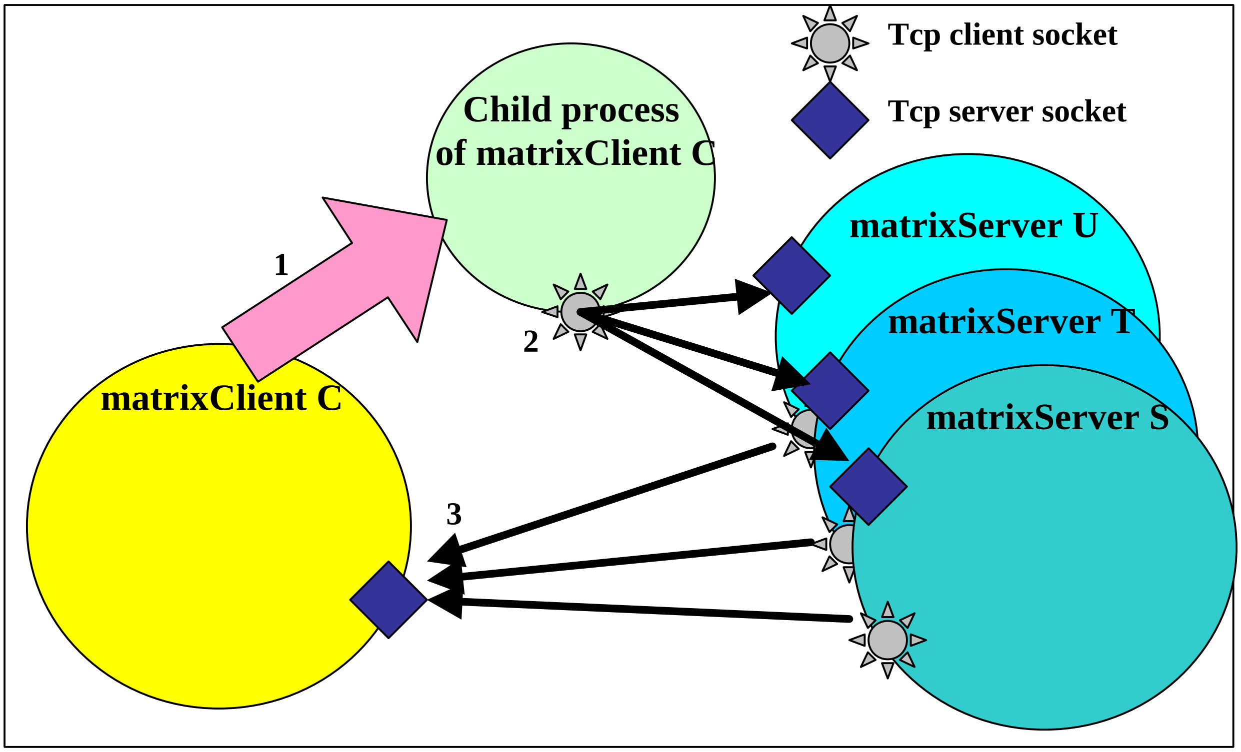

Figure 2: Distributed Matrix Multiply

(1. matrixClient forks child process. 2. Child sends all requests for work to servers. 3. matrixServers reply to the main process.

Note that this implementation of the matrixServer is stateless; that is, every time the matrixClient sends a request for work to the matrixServer, it must send the complete B matrix.

The advantage of this implementation is that requests for work to a single matrixServer may be interleaved from several matrixClients. Since the matrixServer has no state, it does not care from which client it received the current request; previous results have no impact on current calculations.

However, message sizes can be very large for every request to a server since we include both the row from the A matrix as well as the entire B matrix – even if we are sending a subsequent row computation to the same server. In order to speed up the calculations, we implemented a stateful matrixServer. On the first request sent to each matrixServer, the matrixClient sends the full message: index, #cols in A, # cols in B, A row, B matrix. On subsequent sends to the servers, the matrixClient setscols in B equal to zero and sends no B matrix. On the matrixServer side, if cols in B is nonzero, the server deletes the old B matrix, allocates space for and receives the B matrix. But if cols in B is zero, the server uses the B matrix it currently has in memory. We found that this drastically improved speedup. Again, the drawback is that a stateful matrixServer cannot gracefully handle interleaved requests.

A work around for the interleaved work request problem, that we did not implement, is to send some identification to the matrixServer along with the work requests (eg the ip of the client concatenated with the process id of the current process). Once a matrix server begins to serve a client, it will only accept requests for service from that client until the client sends a termination, or end of job, flag. It would send a request denied message to any other processes requesting service until its current client released it with the end of job flag. This would prevent errors from occurring due to interleaved calculations of unrelated matrix multiplies, but may not be the best use of resources as each server would be dedicated to one client until the client released it. Also, if the end of job flag was not received by the matrix server (if the client crashes, or the message is lost), the server could wait forever and be unavailable to all other processes.

C. Corba Implementation

In CORBA implementation we already had the Naming Service available for us. We used OrbixNames as Naming Service, which had a Load Balancing feature with round robin scheduling. The Naming Service was responding to the clients with an object reference of an available server at the head of the queue.

Each host machine, where client, Naming Service and servers were residing, had an Orbix Daemon running. The communication between client and servers were handled by the daemons, as well-known contact points.

When an object implementations server starts up, it registers its object implementation to the naming server and reports that it is ready to receive calls, where it is added to the list of available servers for a specific group.(i.e. MULTgrp the available object implementations for matrix multiplication.)

When the client contacts the Naming Service, it uses the name MULTgrp to get the object reference for the first available server in the queue for the group, because round robin algorithm is used for scheduling. In this case this causes a Load Balancing throughout the object implementations just like a SPMD machine.

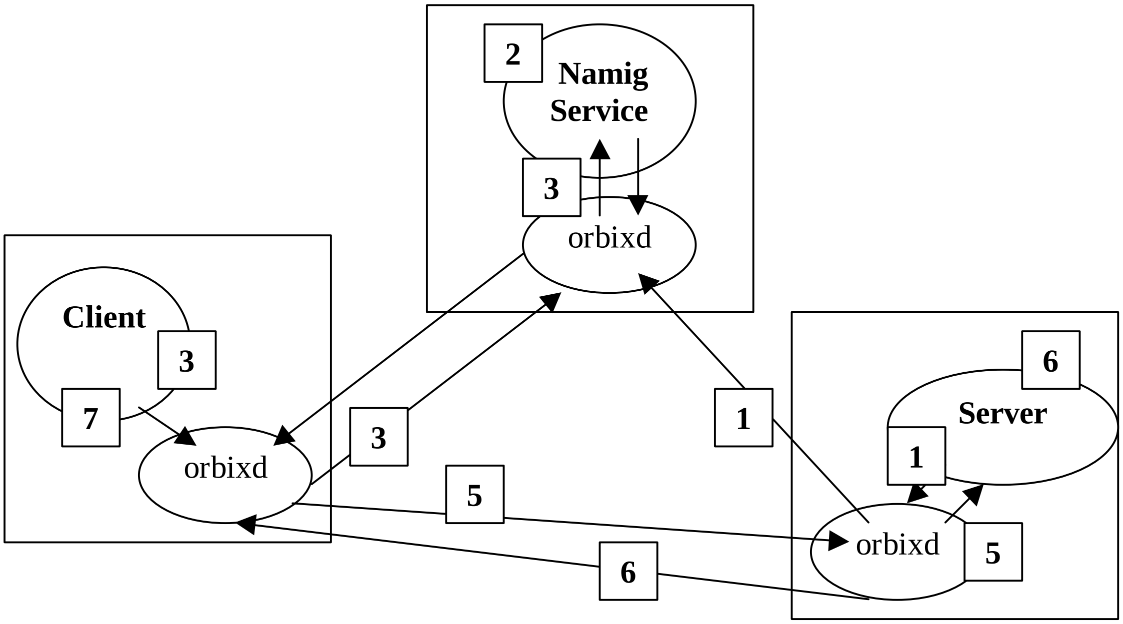

Figure 3: Distributed Matrix Multiply

To demonstrate how the distributed algorithm works, consider this example scenario (refer to Figure 3)

1. Object implementations server starts up, and registers the object implementation to the naming server.(through local daemon)
2. The Naming service adds the ref to the end of the queue of that group.(MULTgrp)
3. Client starts up, prompts user for matrix sizes and filenames, loads matrices. Client tries to resolve the name MULTgrp through Naming Service. Naming Service looks up for that group and returns the object reference at the head of the queue.
4. Client gets a reference to Naming Service and resolves name MULTgrp to get a reference for the first available object implementation.
5. Client creates the request objects and populates them with a row and the second matrix. Finally it invokes send_multiple_requests_deferred() on the orbixd and sends all the requests parallel. Starts waiting for the results. It poll() the request objects to collect the results.
6. Servers receive work requests from client through daemons. They compute the result for each row and return them.
7. Client collects the results available in the request objects and constructs the result matrix.

3. **Results.**

A. Corba versus C++ Sockets.

The comparison experiment consisted of 5 trials each of 10 X 10, 100 X 100 and 200 X 200 matrix multiplies on: 1) no servers (establishes a base), 2) 1 server on same machine, 3) 1 server on different machine, 4) 5 servers on different machines, and 5) 10 servers on different machines from client. The machines were 300 MHz Sun Ultra 10's connected via 10Mbit ethernet in Spanagel 506; the tests were done at times when there were no other users using the machines. The machines used are listed in Appendix 6.d.

Upon receipt of the results, we found that the time measurements of the 10 X 10 matrix multiply were too variable to provide useful insight, so we focused on the 100 X 100 and 200 X 200 matrices. By referring to Table 1, we see that if we exclude the 1st result row, the use of CORBA adds between 23 and 41% to the completion time that you would expect using C++ sockets. If we restrict our review to the larger, and less variable, 200 X 200 matrix multiply we see that the range of added overhead is further restricted to between 26 and 29%.

The exception to our general observation occurs when using a server on the same machine as the client. In this situation, Corba takes almost as long as it would take to use a server on another machine. Contrast this to the C++ socket implementation that takes slightly longer than the baseline test of the non-distributed algorithm. This is because, even though the client and server reside on the same host, the ORB must repeatedly open and close a socket to itself for each request for service.

Comparison Between C++ Sockets and CORBA Matrix Multiply						
	C++ Sockets		**CORBA**		**Socket/CORBA**	
server configuration	**100X100**	**200X200**	**100X100**	**200X200**	**100X100**	**200X200**
no servers	0.2141	1.7874	0.0000	0.0000	N/A	N/A
1 server, same host	0.5571	3.2124	3.2891	29.6662	0.1694	0.1083
1 server, different host	3.6977	29.0047	4.5242	36.4906	0.8173	0.7949
5 servers, different host	3.7413	28.4360	4.5755	36.3644	0.8177	0.7820
10 servers, different host	3.7202	28.5166	5.2512	36.6698	0.7085	0.7777

Table 1: C++ Socket vs. Corba Matrix Multiply Performance

The detailed tables of results are in appendix 6A. Figure 4 below further depicts the overhead of Corba vs C++ sockets.

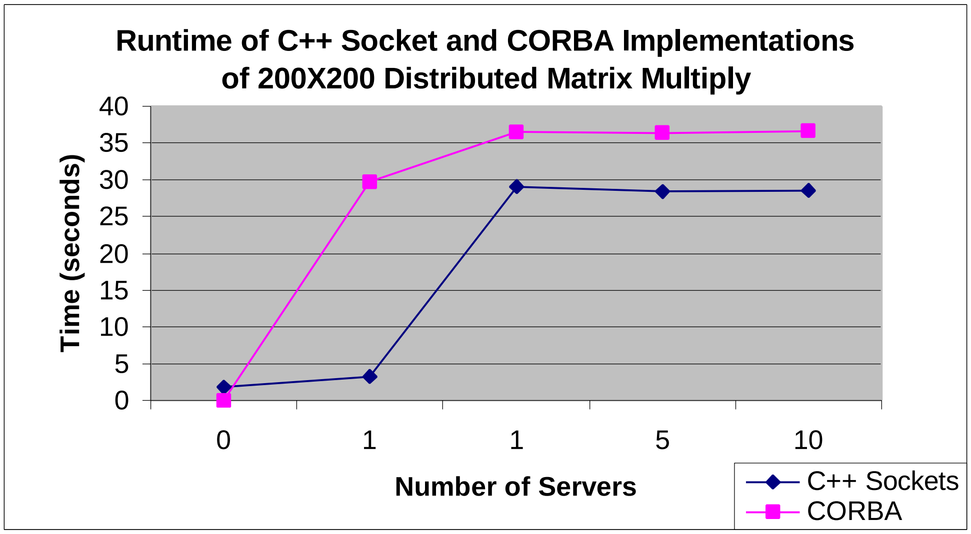

Figure 4: Socket vs Corba Matrix Multiply Performance

(Note: On the X axis, the first 1 represents client and server on same host, the second 1 represents client and server on different hosts).

We were going to end our analysis of Sockets vs Corba here, but then we realized that there were several other areas that were relevant to the analysis. These are training, development time, and code size.

The training required to learn C++ sockets, for someone who is trained in C++ and basic Unix, is approximately 1 week. Contrast this to the requirements for Corba proficiency that may be as great as 3 months.

The advantage of Corba becomes apparent in development time. While it took Matt about 1 week to write and fully debug his implementation, Alpay finished the Corba implementation in 1 day. This is significant when you consider the cost of a developer, the cost of equipment and facilities, and the cost of getting a product fielded or to market earlier than the competition.

Another advantage of Corba is the size of the code generated. Alpay's Corba implementation took 197 lines of code, while Matt's socket implementation took 859 lines. This is significant not only because of the costs of development, but from a maintenance standpoint. As lines of code increase, the potential for bugs also increases, as does the cost to repair those bugs. Additionally, shorter code is easier to understand and maintain.

So, when deciding whether to use Corba versus a lower level implementation, at least the factors addressed above must be considered.

B. C++ Sockets with Stateful Servers.

We were somewhat surprised to find no speedup in our initial implementation of the distributed algorithm. We realized that communication time greatly exceeded computation time because we were using loosely coupled servers implemented over TCP-IP. To try to observe speedup, we increased the problem size: we tried both 500 X 500 and 1000 X 1000 matrices, with no observed speedup. In fact, the distributed algorithm ran several times longer than the non-distributed algorithm.

We then did a quick and dirty big "O" analysis (appendix 6.E), and found that, not only did computation time increase with the cube of the problem size,but so did our message traffic and the corresponding messaging times. Because, as problem size increased, both computation and messaging complexity increased with$O(N^3)$, we realized that our current implementation of the matrix multiply algorithm would never show speed up.

Our solution was to implement stateful servers, as described at the end of section 2B. By using stateful servers, we only had to send the B matrix to each server once, then on every other transmission, we only sent each server the A row, index and dimensions. This reduced communication complexity from$O(N^3)$ to $O(N^2)$ (See Big O analysis, appendix 6.E). We found that if the problem size was large enough to overcome connection and message costs, we were able to see significant speed up results.

We found that the problem size had to be greater than 100 X 100 to see any speed up; at that problem size or lower, adding servers actually dropped speedup to less than 1 due to the cost of establishing connections and sending messages. This drop in speedup due to distributing too small of a problem is illustrated in Figure 5.

Note that all of our speedup calculations use the non-distributed (eg no server) results as the base for calculation.

The best speedup for a 500 X 500 matrix multiply occurred with 6 servers and was2.94; this speedup was achieved at an efficiency of 0.490. The most efficient speedup was 0.9795 at an efficiency of 0.9795 using 1 server on a different host than the client. These results are illustrated in Figure 6 and Figure 7 below.

The best speedup for a 1000 X 1000 matrix multiply occurred with 9 servers and was4.874; this speedup was achieved at an efficiency of0.542. The most efficient speedup was 1.032 at an efficiency of 1.032. This was achieved using 1 server on the same host as the client. This result shows superscalar speedup – the parallel version runs faster than the sequential version. We believe that, although we attempted to keep both the sequential and parallel algorithms as close as possible, the distributed algorithm introduced some more efficient methods than the sequential, causing thissuperscalar speedup. These results are illustrated inFigure and Figure 7 below.

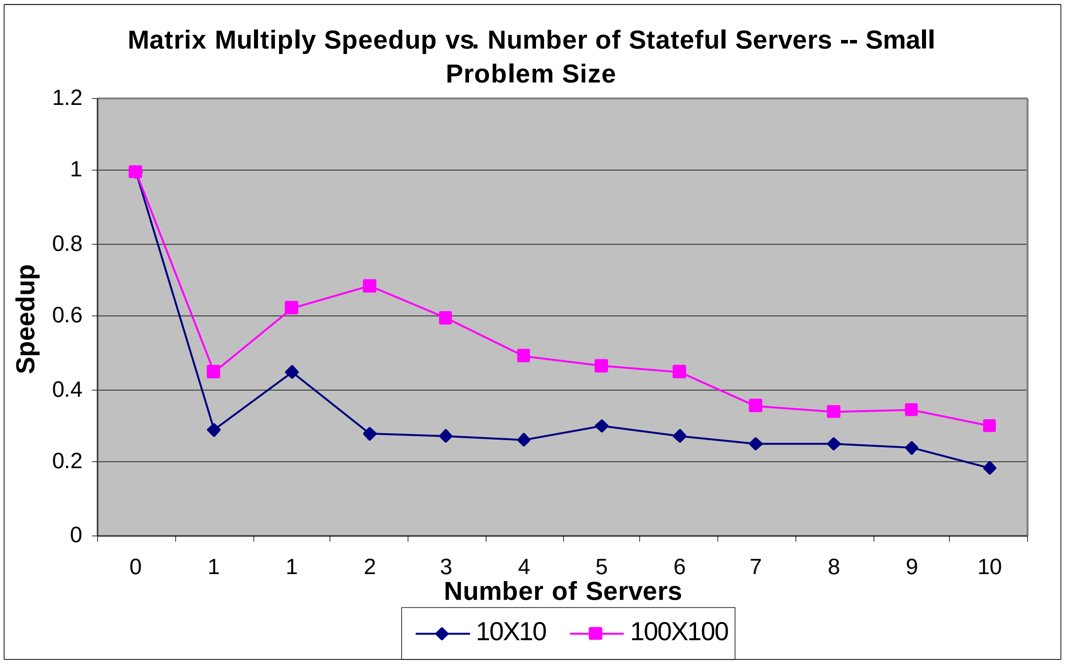

Figure 5: Speedup Curves -- Small Problem Size

(Note: On the X axis, the first 1 represents client and server on same host, the second 1 represents client and server on different hosts).

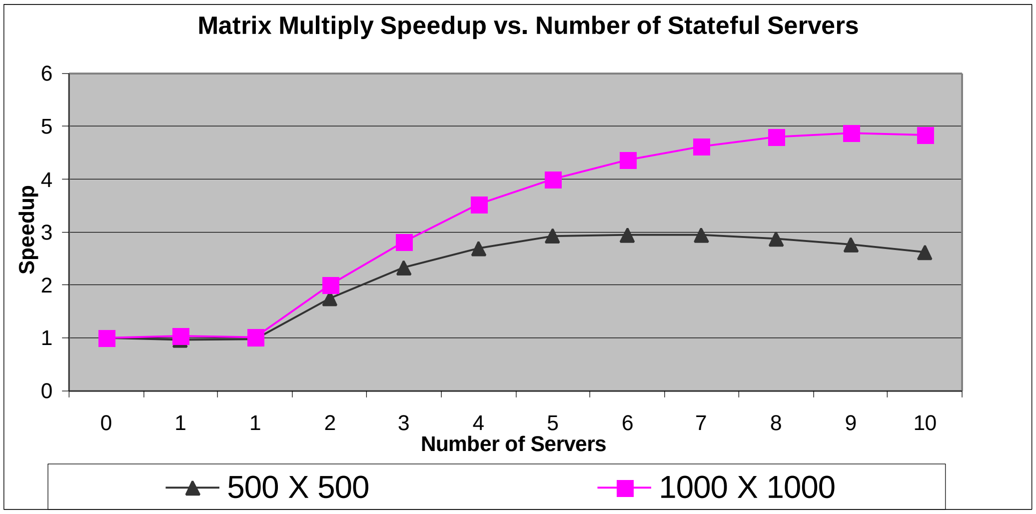

Figure 6: Stateful Server Speedup Curves
(Note: On the X axis, the first 1 represents client and server on same host, the second 1 represents client and server on different hosts).

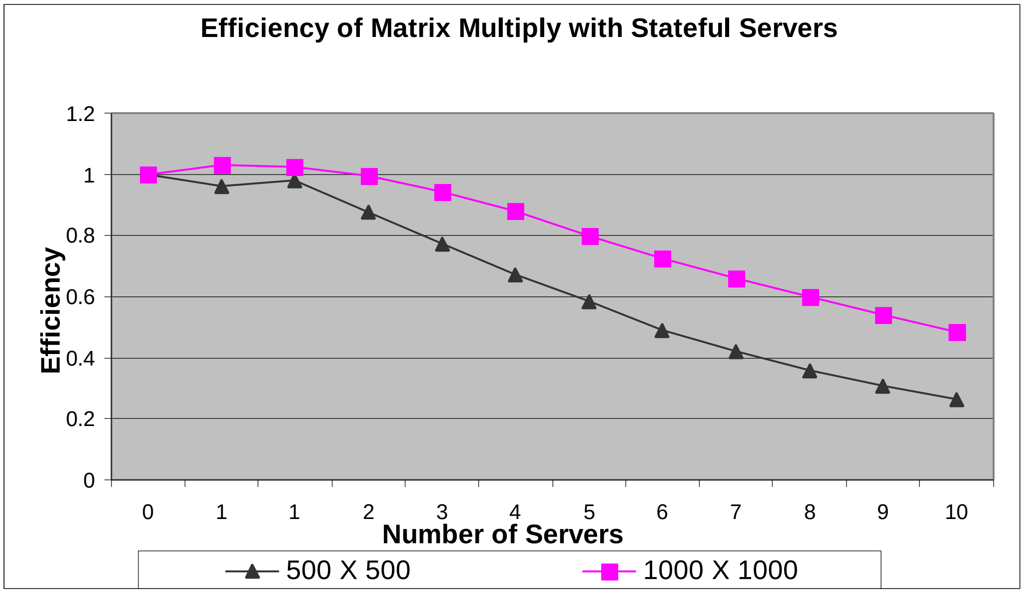

Figure 7: Stateful Server Efficiency
(Note: On the X axis, the first 1 represents client and server on same host, the second 1 represents client and server on different hosts).

C. Bugs and Lessons Learned

1) Achieving Speedup. Getting a sequential algorithm, even a simple one, to run faster on a distributed system is not as simple as we first thought. A lot of care in must be taken in how messages are passed to prevent creating a sequential algorithm with speedup less than 1.

2) Buffer Deadlock. We ran into a buffer deadlock problem in the stateless C++ implementation which took a while to find. The problem resulted from the sequential nature of the client program; the client would send all rows and matrices to all servers before ever receiving any messages. The servers replied immediately after calculating the result row, sending the reply back to the client. These replies piled up into the received buffer on the client side. At some point, the buffer became full and a server “hung” waiting to be able to send a message to the client. Meanwhile, the client is still writing messages across the socket. When it tries to send the message to the “hung” server, it is able to connect but not send the message, so it buffers this message. This continues until the client’s buffer is full, and it hangs. The server is waiting for the client to read its messages, and the client is waiting for the server to read its messages, resulting in deadlock. We solved this by forking off the send portion of the client as a child process, so that the client can both send and receive at the same time.

3) Dynamically Allocating 2d Arrays in C++. You cannot dynamically allocate a 2d array directly in C++ (eg this declaration and definition is not valid:
int* myAry = new int[numRows][numCols]; where numRows and numCols are not known at compile time). To get around this, Matt implemented a 2D Array Class and Alpay allocated space using a “for” loop.

4) Corba Buffer Size. The Corba implementation had a limited buffer size; we could not pass more than a 200 X 200 matrix from client to servers (160,00 bytes). Using C++ sockets, we just had to statically allocate sufficient buffer space prior to run time.

5) TCP vs UDP. Since UDP does not guarantee delivery, or order of delivered packets, large messages (large arrays) could not reliably be reassembled, and TCP had to be used for the C++ sockets implementation.

6) Corba Server and Client on Same Machine. As discussed in paragraph XXX, when the Corba Server and Client ran on the same machine, it ran considerably slower than the C++ Socket client and server equivalent.

4. Source Code

- A. C++ Sockets Program Listing
- B. Corba Program Listing

5. Bibliography

1. Sean Baker, *CORBA Distributed Objects Using Orbix,* Addison-Wesley, 1997
2. John A. Zinky, David E. Bakken, and Richard D. Schantz*, Architectural Support for Quality of Service for CORBA Objects,* John Wiley & Sons, inc, 1997.
3. Robert Orfali, Dan Harkey and Jeri Edwards,*Instant Corba,* John Wiley & Sons, inc, 1997.
4. Robert Orfali, Dan Harkey and Jeri Edward *The Essential Distributed Objects Survival Guide* John Wiley & Sons, inc March 1996.
5. Robert Orfali and Dan Harkey *Client/Server Programming with Java and CORBA* John Wiley & Sons, inc, 1997
6. IONA Technologies, *Orbix Programmer's Guide,* Iona Technologies, 1997
7. Hesham El-Rewini, Ted G. Lewis, *Distributed and Parallel Computing,* Manning, 1998.
8. John Shapely Gray, *Interprocess Communications in Unix, the Nooks and Crannies,,* Prentice Hall, 1997.

6. Appendix

A. C++ Sockets with Stateless Servers and Corba Results Table

Results of Distributed Matrix Multiply Using C/Unix Sockets, Stateless				
		Matrix Size		
Server Configuration	**Iteration**	**10X10**	**100X100**	**200X200**
no servers, only client	1	0.03422	0.213527	1.72539
	2	0.001418	0.21494	1.76686
	3	0.001334	0.210902	1.80662
	4	0.001368	0.216048	1.83943
	5	0.00136	0.214953	1.7988
	average	0.00794	0.214074	1.78742
1 server, same machine As client	1	0.02962	0.565334	3.19687
	2	0.060903	0.611573	3.20937
	3	0.026343	0.536085	3.227
	4	0.0328	0.53701	3.21336
	5	0.027317	0.535676	3.21543
	average	0.035397	0.5571356	3.212406
1 server, different machine from client	1	0.024698	3.69694	28.9662
	2	0.023073	3.70562	29.0327
	3	0.023013	3.68424	28.9536
	4	0.022988	3.69203	29.0587
	5	0.02301	3.70943	29.0125
	average	0.023356	3.697652	29.00474
5 servers, none on client machine	1	0.032801	3.71558	28.4544
	2	0.028046	3.724	28.4172
	3	0.02794	3.72808	28.4549
	4	0.039553	3.76833	28.4211
	5	0.030445	3.77034	28.4322
	average	0.031757	3.741266	28.43596
10 servers, none on client machine	1	0.029442	3.72934	28.558
	2	0.027547	3.71921	28.473
	3	0.041277	3.71126	28.549
	4	0.02741	3.7319	28.458
	5	0.028253	3.70922	28.545
	average	0.030786	3.720186	28.5166

Results of Distributed Matrix Multiply Using Corba				
		Matrix Size		
Server Configuration	**Iteration**	**10X10**	**100X100**	**200X200**
no servers, only client	1	N/A	N/A	N/A
	2	N/A	N/A	N/A
	3	N/A	N/A	N/A
	4	N/A	N/A	N/A
	5	N/A	N/A	N/A
	average			
1 server, same machine as client	1	0.100563	3.30899	30.9037
	2	0.105458	3.31681	28.8873
	3	0.111166	3.29292	29.3455
	4	0.118575	3.2985	30.1138
	5	0.095567	3.22849	29.0806
	average	0.106266	3.289142	29.66618
1 server, different machine from client	1	0.142726	4.52266	36.616
	2	0.10984	4.5406	36.0907
	3	0.119963	4.5302	36.564
	4	0.10807	4.53395	36.5332
	5	0.117222	4.49347	36.649
	average	0.119564	4.524176	36.49058
5 servers, none on client machine	1	0.162902	4.58567	36.396
	2	0.14028	4.57764	36.371
	3	0.156322	4.58606	36.2936
	4	0.168699	4.5618	36.4416
	5	0.171204	4.56622	36.32
	average	0.159881	4.575478	36.36444
10 servers, none on client machine	1	0.293782	5.18686	36.8186
	2	0.299528	5.26731	36.679
	3	0.24345	5.21294	36.6034
	4	0.271893	5.30495	36.5714
	5	0.251008	5.28372	36.6766
	average	0.271932	5.251156	36.6698

B. C++ Sockets with Stateful Servers Results Table

Results of Distributed Matrix Multiply Using C/Unix Sockets, Stateful					
		Matrix Size			
Server Configuration	**Iteration**	**10X10**	**100X100**	**500 X 500**	**1000 X 1000**
no servers, only client	1	0.03422	0.213527	33.287	338.103
	2	0.001418	0.21494	33.2161	340.052
	3	0.001334	0.210902	33.2668	337.83
	4	0.001368	0.216048	33.4291	338.653
	5	0.00136	0.214953	33.5073	337.354
	average	0.00794	0.214074	33.34126	338.3984
1 server, same machine as client	1	0.04	0.490709	34.7345	323.934
	2	0.023908	0.481969	34.7659	324.178
	3	0.024263	0.4661969	34.821	
	4	0.023914	0.468853	34.6721	
	5	0.023867	0.463636	34.6987	
	average	0.02719	0.4742728	34.73844	324.056
1 server, different machine from client	1	0.023526	0.415318	35.5373	326.585
	2	0.022547	0.421148	35.466	328.432
	3	0.020974	0.412318	35.389	
	4	0.021013	0.411463	35.4647	
	5	0.021036	0.414522	35.3565	
	average	0.021819	0.4149538	35.4427	327.5085
2 servers, none on client machine	1	0.029879	0.318277	18.9809	167.951
	2	0.026932	0.309274	19.0294	167.979
	3	0.02794	0.31011	19.1709	
	4	0.0281	0.31355	18.9653	
	5	0.02854	0.316789	19.0358	
	average	0.028278	0.3136	19.03646	167.965
3 servers, none on client machine	1	0.029879	0.41226	14.287	118.607
	2	0.029014	0.344178	14.2014	118.563
	3	0.027264	0.346746	14.4001	
	4	0.028554	0.34821	14.4802	
	5	0.029012	0.347826	14.4111	
	average	0.028745	0.359844	14.35596	118.585
4 servers, none on client machine	1	0.026567	0.406102	12.4153	95.012
	2	0.030197	0.486873	12.4224	95.1485
	3	0.037655	0.450747	12.4341	
	4	0.027786	0.405199	12.3933	
	5	0.02765	0.421101	12.401	
	average	0.029971	0.4340044	12.41322	95.08025

Results of Distributed Matrix Multiply Using C/Unix Sockets, Stateful

		Matrix Size			
Server Configuration	**Iteration**	**10X10**	**100X100**	**500 X 500**	**1000 X 1000**
5 servers, none on client machine	1	0.025989	0.442548	11.433	83.8903
	2	0.028055	0.464511	11.3503	83.8453
	3	0.026616	0.449753	11.4211	
	4	0.025807	0.460504	11.3711	
	5	0.0264	0.491189	11.436	
	average	0.026573	0.461701	11.4023	83.8678
6 servers, none on client machine	1	0.026368	0.490709	11.3172	76.807
	2	0.029012	0.481969	11.3876	76.742
	3	0.032566	0.4661969	11.1808	
	4	0.028125	0.468853	11.4787	
	5	0.029952	0.463636	11.333	
	average	0.029205	0.4742728	11.33946	76.7745
7 server, different machine from client	1	0.029589	0.606383	11.53	72.5306
	2	0.037985	0.593823	11.2737	72.3187
	3	0.028464	0.59655	11.3548	
	4	0.029125	0.605671	11.291	
	5	0.031254	0.599932	11.3025	
	average	0.031283	0.6004718	11.3504	72.42465
8 servers, none on client machine	1	0.034046	0.644748	11.5879	69.8434
	2	0.02762	0.635974	11.5782	69.8044
	3	0.033888	0.648489	11.59012	
	4	0.029773	0.599443	11.5855	
	5	0.03102	0.60215	11.5955	
	average	0.031269	0.6261608	11.58744	69.8239
9 servers, none on client machine	1	0.0473	0.633137	12.031	68.5766
	2	0.03555	0.626462	12.1045	68.8724
	3	0.0245	0.62955	12.059	
	4	0.02789	0.631859	12.078	
	5	0.02985	0.5981	12.068	
	average	0.033018	0.6238216	12.0681	68.7245
10 servers, none on client machine	1	0.050847	0.733338	12.5087	70.1243
	2	0.046994	0.702655	12.4678	68.4136
	3	0.042469	0.720646	12.5932	71.4136
	4	0.037012	0.719209	13.258	
	5	0.032846	0.693501	12.6363	
	average	0.042034	0.7138698	12.6928	69.26895

C. Speedup and Efficiency of Stateful Servers

Speedup, Stateful Servers				
	Matrix Size			
#Servers	**10X10**	**100X100**	**500 X 500**	**1000 X 1000**
0	1	1	1	1
1	0.292015	0.4513732	0.95978	1.04425902
1	0.45203	0.6219946	0.979477	1.03325074
2	0.280782	0.6826339	1.751442	2.01469592
3	0.276226	0.5949078	2.322468	2.85363579
4	0.264923	0.4932531	2.685948	3.55908193
5	0.298795	0.4636637	2.924082	4.03490255
6	0.271875	0.4513732	2.940286	4.40769266
7	0.253809	0.3565097	2.937452	4.67242023
8	0.253922	0.3418834	2.877361	4.84645515
9	0.240475	0.3431654	2.76276	4.9239849
10	0.188897	0.2998782	2.626785	4.88528266

Efficiency, Stateful Servers				
	Matrix Size			
#Servers	**10X10**	**100X100**	**500 X 500**	**1000 X 1000**
0	1	1	1	1
1	0.292015	0.4513732	0.95978	1.04425902
1	0.45203	0.6219946	0.979477	1.03325074
2	0.140391	0.341317	0.875721	1.00734796
3	0.092075	0.1983026	0.774156	0.95121193
4	0.066231	0.1233133	0.671487	0.88977048
5	0.059759	0.0927327	0.584816	0.80698051
6	0.045312	0.0752289	0.490048	0.73461544
7	0.036258	0.05093	0.419636	0.6674886
8	0.03174	0.0427354	0.35967	0.60580689
9	0.026719	0.0381295	0.306973	0.54710943
10	0.01889	0.0299878	0.262679	0.48852827

C. Machines Used in the Experiments

The machines used were Sun Ultra 10's, 300Mhz, in Spanagel #506.

Hosts used in experiment	
Host Name	**Role**
indus	Nameserver and client
lynx	server #1
mars	server #2
mensa	server #3
crater	server #4
ariel	server #5
apus	server #6
janus	server #7
gemini	server #8
grus	server #9
libra	server #10

E. Big "O" Analysis of Stateful vs Stateless Server Messages.

Assume Square, N X N matrices

Let N = # of rows and columns in both A and B matrices
S = # of servers to distribute to
C = constant value of index, and dimensions in work request (size of 3 longs)

Stateless Server Messages.

N messages are sent.

Each message contains an N-sized row, N^2 –sized matrix, and C-sized constants.

So, message complexity is $O(N)*O(N^2 + N + C)$, drop the constant term and multiplying yields $O(N^3)$.

Stateful Server Messages.

N messages sent.

The first message each server receives contains an N-sized row, N^2 –sized matrix, and C-sized constants. S of these messages are sent.

The subsequent messages each server receives contain an N-sized row, and C-sized constants. N – S of these messages are sent.

So, message complexity is $O(S)* O(N^2 + N + C) + O(N – S)*O(N + C)$. Recognize that S and C are constants and drop out of the equation yields $O(N^2 + N) + O(N^2)$, which simplifies to $O(N^2)$.