Quake Server Parallelization

ECE1747 Project Report

December 20, 2004

Authors:
Tomasz Czajkowski
Blair Fort
Ian Kuon

Abstract

Multi-player game servers present an interesting computational challenge since the desire to support large numbers of interacting players in real-time places strict constraints on server computation time. These high computational needs make these game servers attractive candidates for parallel processing because at high player counts the server becomes a performance bottleneck. This work investigates the issues in parallelizing such servers by parallelizing one particular application, the QuakeWorld game server.

The approach presented in this report focuses on utilizing the spatial locality of players within the game environment to achieve more efficient parallelization. The main idea is to divide the players amongst all available threads in such a way as to only process clients sequentially if they are in physical proximity to one another. This is done by partitioning the virtual world into distinct areas and processing players within each area sequentially, while each distinct area of the virtual world is processed concurrently. To further increase the capacity, other stages of the server computation are also parallelized.

This parallel implementation of the QuakeWorld game server was confirmed to be functional on multiple platforms. Testing on a four 296 MHz UltraSPARC-II processor machine found that this parallel implementation could deliver significant increases in server capacity. With four threads, the capacity was found to be increased by over 60% with respect to the sequential implementation.

Table of Contents

Table of Tables

Table of Figures

1 Introduction

Interactive games have long been one of the most significant drivers of computing technology. Traditionally, games have focused on single users with limited capabilities for the user to interact with other users. On-line games that encourage interactions between massive numbers of players are becoming increasingly popular. One central element to this popularity is the ability to support the interactions of large numbers of players. However, handling real-time interactions between such large numbers of players is computationally intensive and the game server processing can become a significant bottleneck. Parallel processing is one approach that could improve the response time of game servers and thereby enable increased capacity. The QuakeWorld Game Server [iS] is used as a test platform in this work for examining the possible gains in parallelizing multi-player game servers. This project examines various possible strategies for increasing the number of clients that can be serviced by the game server.

QuakeWorld is a multiplayer, first person shooter game, in which players roam a virtual world while interacting with other players. This genre of games is very popular as it does not require a large time commitment from players, while providing sufficient level of satisfaction from playing the game. The game itself becomes more interesting as more players enter the virtual world and interact with one another. The increased number of players increases the load on the server and to maintain adequate performance it is often necessary to utilize leading edge hardware. However, for massively interactive games, even a single leading-edge server may not be able to handle the desired numbers of clients. In such cases, parallelism is needed to handle the high client counts. An alternative motivation may be to avoid the high cost of leading-edge hardware by utilizing older parallel machines. In both cases the server must be updated to enable the parallelism and this work proposes a parallel design for the QuakeWorld server.

Prior work by Abdelkhalek and Bilas [AB04] presented one solution to this game parallelization problem. However, the capacity increase was limited to a modest 25% even when running with 8 threads. This current work attempts to improve on this performance by utilizing a new dynamic approach for task assignments to threads. The goal in this new assignment, which is based on a player's position within the game environment, is to reduce the impact of lock contention. This area-based approach to parallelism leverages game-specific information, specifically that players only interact with their immediate environment within the virtual world. Objects that are far from the client in the virtual world are not affected by the distant client's actions and, therefore, to minimize contention between threads parallelization is most effective if this location information is utilized.

This report is organized as follows: in Section 2, background on QuakeWorld is provided along with an overview of past parallelization efforts. In Section 3, the design details of the parallelized QuakeWorld game server developed for this work are described. The testing environment and challenges in finding a suitable test platform are described in Section 4. The performance results of this parallel design are then analyzed in Section 5. Finally, general conclusions and possible directions for future work are given in Section 6.

2 Background

Most past work focused on the game performance deals with topics such as speeding up artificial intelligence [KZ02] or information gathering to understand player behavior [TT02]. There has been some research into creating multi-player games using a peer-to-peer (P2P) network instead of the classic client-server based approach used in games like QuakeWorld. That research focused on the viability of P2P multi-player game systems [ED03] and creating a cheat-proof environment [GZLM04], while disregarding game performance when using such a system. While such work is an interesting direction for future game designs it did not address the challenge of improving the performance of existing games.

An alternative approach to improving multiplayer network game performance is described in [WWL04] and is based on an idea known as "gamelets". The paper describes "gamelets" as an execution abstraction for processing workloads produced by related objects in the game world. These "gamelets" are created dynamically as events in the game take place. The paper discusses running the "gamelets" on a grid-enabled multi-server network, where, as the number of "gamelets" increases, the number of servers used also increases to handle the increased workload. That paper describes the implementation of a small scale system running a simulated game; however, it is not yet clear how such work could be applied to highly interactive games such as QuakeWorld.

In terms of game parallelization of a real commercial multi-player game, the most relevant work is past work by Abdelkhalek and Bilas [AB04]. In that work they improved the performance of the QuakeWorld game server by enabling it to support 25% more clients when running 8 concurrent threads. Before describing the parallel implementation proposed in [AB04], the basic operation of the QuakeWorld server will be described.

2.1 QuakeWorld Game Server

There are three main components to QuakeWorld game server processing. These stages involve 1) world physics, 2) client requests and processing, and 3) replying to clients. Stages 1 and 2 update the game world's information while stage 3 only reads the game state information in order to produce reply messages for the clients. These 3 stages execute sequentially and make up a server **frame**. These steps are depicted graphically in Figure 1. During normal operation, the server continuously processes server frames. Each server frame may involve one or more client requests. Before a server frame starts, the server waits for a least one request from a client. Each stage is described in more detail in the following paragraphs.

In Stage 1, the server sets up the game world for the new server frame. The server applies world physics, such as gravity or momentum, to all objects in the game. Any object floating in the air will have gravity applied to it and will start or continue falling. Momentum of an object is also considered. For example, bullets flying through the air will continue along the same trajectory unless they hit another object.

Stage 2 is where the server receives and processes client requests. Clients requests may be either management or game play type requests. Examples of management requests are connection, disconnection and change game map requests by clients. These requests occur infrequently and are mostly at the beginning and ending of a game, and they can cause effects throughout the whole map. Game play requests consist of player actions, such as move forward, change weapons, and fire a weapon. Most of the client requests are of this type and generally only involve changes to the game in the vicinity of the client.

In stage 3 the server produces replies to client requests. For each client request, a single response is created. The responses include all the changes to objects in the current client's line of sight or objects that may soon be in the client's line of sight.

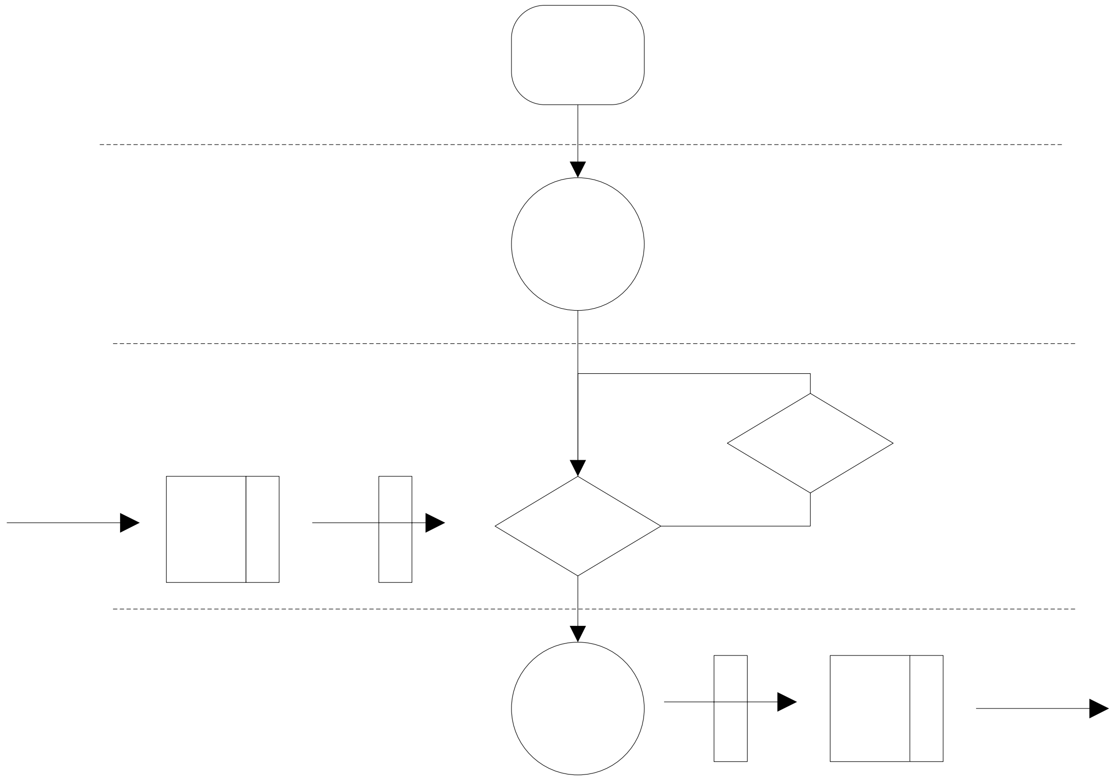

Figure 1 – Basic Quake Server Computation Frame

2.2 Parallelizing QuakeWorld

This section describes the work done by Abdelkhalek and Bilas [AB04] to parallelize computation of a QuakeWorld Server frame. Their parallel design is based on Pthreads running on a shared memory multiprocessor (SMP). In the current implementation, they create one thread per processor at initialization time. Each thread is given its own UDP port to communicate with its clients, which are statically assigned to specific threads during initialization.

The parallelization of each stage is done independently by having barriers between consecutive stages. Figure 2 illustrates their parallelization strategy.

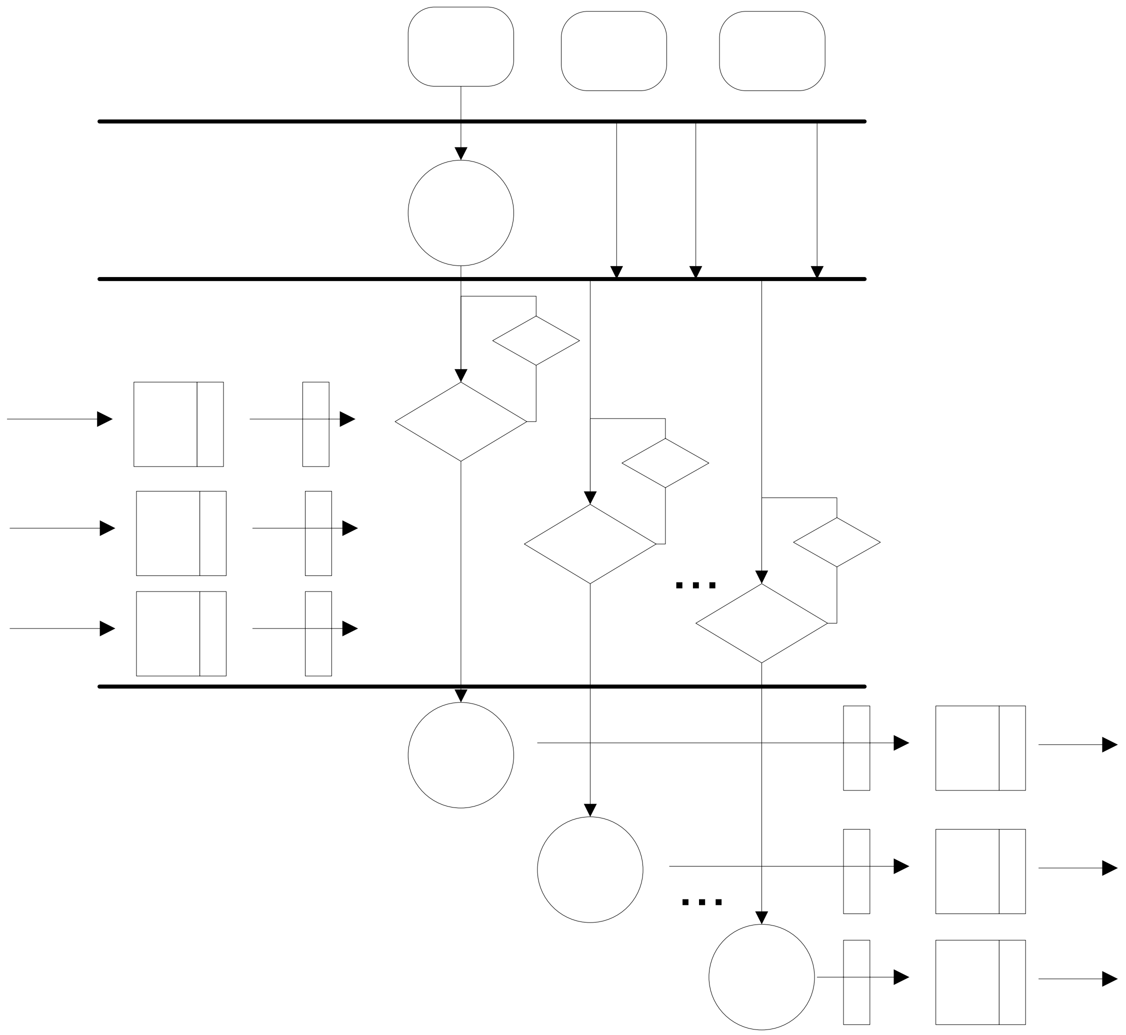

Figure 2 – Parallel Server Design implemented in [AB04]

Stage 1

In their previous work [ABM01], it was determined that Stage 1 processing constitutes only 5% to the total execution time; therefore, it was not worth parallelizing. For each frame, a single thread is chosen as the master and performs the Stage 1 computation by itself, while the other threads wait at a barrier. The master thread is chosen to be the thread which receives a client request first.

In Stage 2 each thread processes requests from its clients. Since the spatial locality of players is not considered in the assignment of players to threads, it is possible for two different threads to process clients in physical proximity to one another. As a result of this, locks on client data are needed to ensure data consistency. In addition to locks on client data, the area in which a particular client is located needed to be locked. This was done to ensure that any modification a client makes to the area around him is done consistently, despite the presence of other clients.

Rx

For Stage 3 no locks were necessary as the stage only focuses on writing data from global data structures to each individual client. Each thread creates replies to all clients from whom it receives requests.

As described previously, this approach to parallelization was only able to achieve a 25% increase in client capacity. This increase was limited by lock contention and long waits at barriers. To avoid these past problems, this current work will use an area-based approach for assigning players to threads. Each thread can then operate concurrently when processing its area of the map. When a client crosses the boundary between two quarters, client message processing will be handed off from one thread to another. The redesigned QuakeWorld game server that implements this is described in the following section.

3 Implementation

Now that the core computation of the QuakeWorld server has been described, the design of the parallel implementation can be presented. Based on past work [AB04], it is known that Stage 2 and Stage 3 are the most computationally complex. Therefore, the focus in this parallel design is to ensure the effective parallelization of these two stages of the computation. The following sections describe the parallel design of these steps in the computation. Parallelization of other stages of the computation was considered but for reasons that will be outlined in Section 3.4, such parallelization was not implemented. Finally, since the goal in parallelizing the QuakeWorld server was to increase the number of clients that can be supported, changes that were made on the client side to enable increased interaction are also described in Section 3.5.

3.1 General Parallelization Strategy

There are many approaches that could be taken when parallelizing a large application such as the QuakeWorld game server. In this work, the parallelization scheme will be based on the shared memory model of parallel program. This model is the most straightforward to use for parallelizing existing sequential applications. The key tenet principle in the parallel design will be maintaining game consistency and, therefore, any parallelization should be transparent to the user playing Quake. To maintain this consistency, the stages of the computation on the server will typically need to remain distinct. Synchronization via barriers will be used for maintaining this consistency and locks will be used to ensure proper behavior in critical sections.

The focus in this parallelization effort is on increasing capacity. Therefore, initialization issues and other infrequent events will not be parallelized. When it is necessary to perform such operations, the server will do so sequentially to ensure the behavior is consistent with the original server. It is also important to note that the QuakeWorld server does not guarantee global synchronization across all client machines. Such synchronization would be difficult across a system with widely variable network delays. However, due to this lack of synchronization, there is no global ordering for game events and the ordering of two events executed at the same time on two different client machines is not predefined. The server arbitrarily processes the requests as they are received. There is synchronization between the server and the clients so that client is aware of the global game state and can see the impact of his/her actions. This lack of synchronization between clients provides some flexibility when designing the parallel server since events that are received at the same time can be arbitrarily reordered to simplify computation.

This fact is leveraged in the design of Stage 2. With the general approach to parallelization defined, the parallel design of each stage will now be described.

3.2 Stage Two Parallelization

Past attempts to parallelize the second stage of QuakeWorld server engine computation focused on distributing clients evenly amongst available threads [AB04]. This approach disregards the locality of client interaction. For example, a client is more likely to press a button right next to him, rather than on the other side of the map. Thus, it may be better to distribute clients among threads in such a way as to keep clients that may interact with one another in a single thread.

In this work, the distribution of clients is based on the position of each client in the virtual world. To do this, the world is divided into equal rectangular areas, where each area is processed by a single thread. When a client enters one area, the thread processing that area of the map will be responsible for processing this client's requests. When the client moves from one area to the next, the client processing will be transferred to another thread. The number of distinct areas processed concurrently is a function of the number of threads available to the system. This number of threads can arbitrarily set. However, for simplicity in the proceeding discussion, the four thread case in which the world is divided into quarters will be described.

The client distribution strategy described above matches the data organization in the QuakeWorld server itself. The game server keeps track of objects and players by the area they occupy at any given moment. The structure that stores the list of objects in a particular game area is called an area node. In this work a single thread is assigned to process all players in a single area node.

The following sections of this report describe the area node structure in QuakeWorld server and the new design of stage two computation engine. Then the improved computation engine design that better exploits parallelism is described.

3.2.1 Area Nodes

An area node, henceforth referred to simply as a **node**, is a structure that defines boundaries of an area of the virtual world. While this structure deals nicely with objects fully contained within the bound set by a node, it is foreseeable that some objects could be located on a boundary of two nodes. More importantly, players that move around the virtual world will eventually travel from an area covered by one node to an area covered by another node. Despite this, it is desirable to have each object belong only to a single area node. To do this the designers of QuakeWorld game server, Id Software, introduced hierarchy into these nodes.

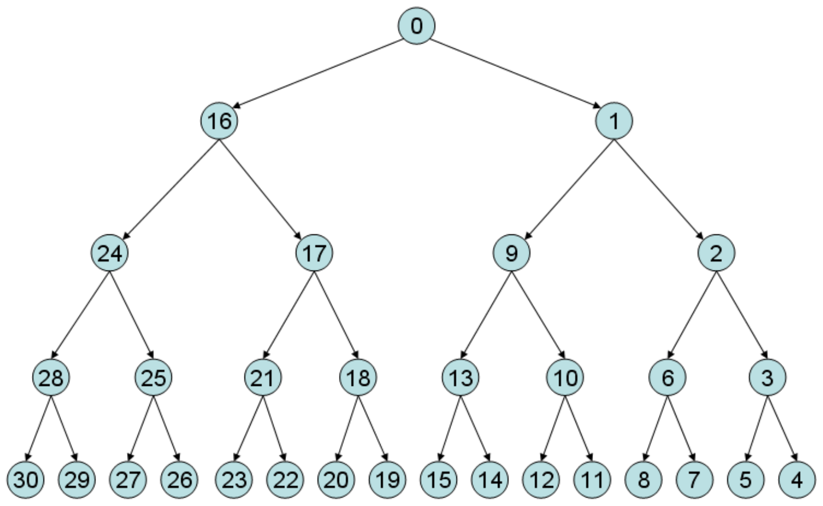

Figure 3 – Area Nodes and their Hierarchy

The node hierarchy is shown in Figure 3. First, a node is created to cover the entire virtual world. This node then creates two child nodes, where each node covers half of the virtual world. Each of these child nodes then creates two nodes that cover half of its portion of the virtual world. This process continues until a balanced tree of 31 nodes is created. The advantage of this node hierarchy is that for every pair of adjacent nodes *A* and *B*, there is a node *Q* that covers both *A* and *B*. Thus, an object or a player that is located on a boundary between nodes *A* and *B* will be considered to be located within node *Q*. For example, node 1 in Figure 3 covers half of the virtual world, while node 16 covers the other half. Node 0 covers the entire world, thus covering both nodes 1 and 16.

The following sections describe how this area node structure can be used to exploit parallelism.

3.2.2 Basic Stage Two Engine Design

The second stage of QuakeWorld game server computation, as defined in Section 2, focuses on receiving updates from players and processing their requests. To maintain this functionality and to utilize area node-based parallelism, the computation in this stage was divided into three steps.

In the first step, all available client messages are stored in a queue. The queue is then parsed to distribute client messages to different threads, based on the player location in the virtual world. Messages from clients on a boundary between nodes were marked for later processing. In the second step, messages from clients are processed concurrently. Finally, in step three messages from clients on node boundaries are processed which completes stage two computations. The block diagram for the implementation of this design is shown in Figure 4.

In Figure 4, each thread is shown as a single vertical line. First, thread 1 receives packets from clients. For each message, thread 1 determines which thread should process the message based on the location of the player within the virtual world. Once the decision is made, the message is stored in a task queue for the corresponding thread (1 through 4). In a case where the client is located on a boundary of area nodes, the message is stored in queue 5, signifying that the client needs to be processed sequentially after all other clients. When message sorting is complete, thread 1 arrives at a barrier where threads 2 through 4 are waiting and the second step begins.

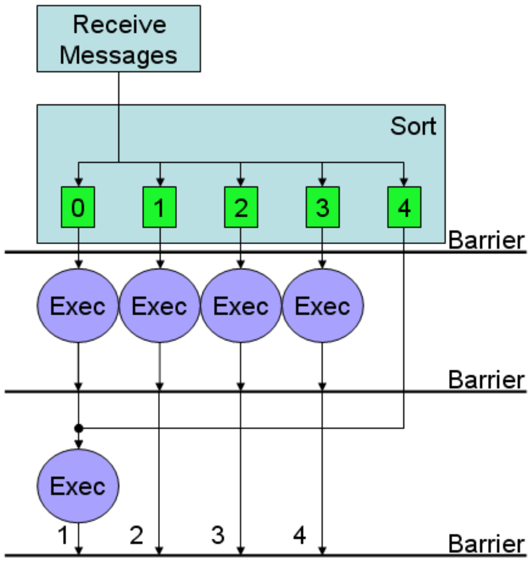

Figure 4 – Block diagram for Stage Two Computation

In the second step each thread processes all the messages in its queue. Once each thread completes the processing of its own message queue, it waits at the next barrier. Once all threads arrive at the second barrier, the third step begins. In the third step thread 1 processes messages in queue 4. This is done to ensure that only one thread processes clients on boundaries of area nodes to avoid data hazards. Then, thread 1 clears contents of all queues and arrives at the final barrier, completing the second stage of QuakeWorld game server computation.

This simple design allows the QuakeWorld game server to exploit parallelism based on player locality. It is possible to further parallelize this stage by concurrently processing client messages for clients in nodes at the same depth. This reduces the number of clients that must be processed sequentially, thereby reducing the time required to process client messages. The improvements to the design of stage two are discussed in the next section.

3.2.3 Optimized Stage Two Engine Design

The design in previous section showed that the second stage of the QuakeWorld game server can process client messages concurrently whenever one client does not affect another client. This was done by dividing the playing field into four quarters and processing clients in each quarter of the map concurrently. Then all clients on edges of each quarter were processed sequentially. However, it is possible for a pair of clients to be on distinct edges of map quarter such that they still do not affect one another and should be processed concurrently. This scenario is shown in Figure 3.

In Figure 5 the quarters of the game map are marked *A*, *B*, *C* and *D*. The first client is on the edge between quarters *A* and *B*, while client two is on the edge between quarters *C* and *D*. Recall from discussion on area nodes that for every pair of adjacent nodes *A* and *B* there is a node *Q* that covers both *A* and *B*. Then if node *R* covers quarters *C* and *D*, and nodes Q and R are distinct these clients can be processed concurrently. To implement this functionality the design in Section 3.1.2 is modified by adding concurrent client processing for two threads just before the step three.

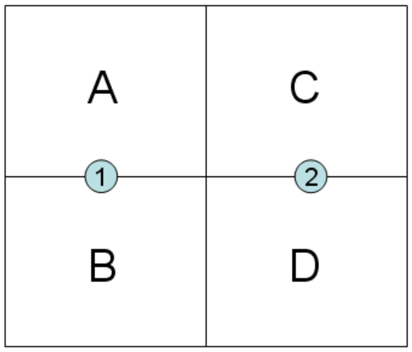

Figure 5 – Concurrently Processed Clients

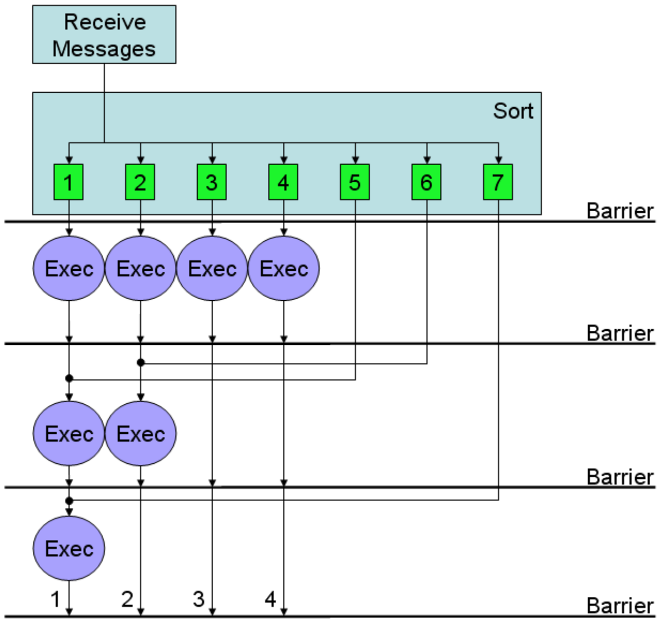

Figure 6 – Optimized Stage Two Computation Engine Design

The block diagram for the redesigned stage two processing is shown in Figure 6. In this design the messages are still received sequentially, but instead of distributing client messages into 5 queues, messages are distributed into 7 queues. Queues 1-4 have the same purpose as before. Messages from clients located on edges of map quarters, who can still be processed concurrently are stored in queues 5 and 6. Finally, queue 7 stores messages from clients that must be processed sequentially.

Once the messages are in their respective queues, four threads process client messages from queues 1 through 4 concurrently. They then arrive at a barrier before they proceed to process messages in queues 5 and 6. Messages in queues 5 and 6 are only processed by two threads, while the other two threads wait at the subsequent barrier. Once these messages are processed, client messages that must be processed sequentially are handled.

This is done the same way as in Section 3.1.2. When more than four threads are available to the server, the depth of the area nodes assigned to each thread initially is increased. This then increases the number of steps in this stage as each higher level in the area node tree is examined.

When all messages are processed the second stage of QuakeWorld game server execution completes. The server must now respond to each client and transmit the results of processing messages from each client, updating each client about the stage of the game. Efforts to parallelize that operation are discussed in the next section.

3.3 Stage Three Parallel Design

In the third stage of the QuakeWorld server computation, replies are sent to any client whose request was processed during the current server frame. During this stage, there are two types of replies that are sent to each client. The first type of messages consists of information messages that contain data such as a client's score and broadcast messages such as those announcing when a player is killed. The second type of messages contains client state information. This includes their position and velocity in the game environment and similar information for the set of objects and players visible from client's current location. In the sequential server, these two types of messages are handled separately. The same distinction is maintained in the parallel server as well.

The processing of the information messages involves iterating over all the clients in the game and writing the necessary updates to the appropriate clients. Parallelizing this part of stage 3 would require locking to ensure consistent data for all the clients. However, this stage is not computationally challenging. Any parallelism could only provide limited speed and the overhead of locking would limit performance. Instead, the stage will be processed sequentially and synchronization using a barrier is performed after this step is complete.

The next step in stage three is significantly more computationally complex. This step involves iterating through all the clients and identifying the objects that are closest to each client. Then for each object close to the client, information such as position and velocity must be recorded for transmission to the client. These accesses to the global game state are read-only while writing is only performed to the individual client's structures. Therefore, it was decided to parallelize this stage of computation by dividing the clients among multiple threads.

QuakeWorld maintains an array of pointers to all possible clients. The size of this array is static and initially each client slot is initialized as being available for new connections. As clients connect to the server, they are assigned to client spots sequentially. Therefore, a cyclic distribution of client slots is necessary to balance the load among threads. A block distribution would have lead to a significant load imbalance since some threads would be assigned to client slots which do not have active clients. The potential still exists for load imbalance between threads. This can occur because all of the updates may not be equally difficult to compute. A severe load imbalance could also occur if all the active clients for one particular thread were to exit the game. New clients would be added to the empty clients slots but, until any new clients entered the game, the thread whose clients left the game would remain relatively unloaded.

In past work [AB04], the server maintained a list of active clients for each thread. However, that approach also suffers from the potential for a load imbalance again if by chance all of clients handled by a single thread exit the game. The benefit in the approach taken in the current work is the overhead in maintaining this list of active clients is avoided. The only disadvantage is that the each thread must check all its potential client slots to determine if the client is active. Fortunately, checking this activity is relatively inexpensive and this should not prove to significantly impact performance.

3.4 Examination of Stage One

The first stage of computation in the server was found in previous work [AB04] to require approximately 5% of the execution time. The low execution time suggested little benefit could come from parallelizing this stage; however, overlapping the computation in this stage with other stages was considered a more promising approach. Unfortunately, detailed investigation found that this was unlikely to be the advantageous.

There were two major factors that led to this conclusion. The first issue is that Stage 1 involves writing to the global server state variables. Parallelizing Stage 1 with Stage 3 from the previous frame would necessitate duplicating the entire global server state. This would be necessary because Stage 3 reads the state to send the appropriate updates to the clients and, therefore, the data must not be modified until Stage 3 completes. This need to duplicate the structure for the game state would limit the potential gains from parallelism with Stage 3. The alternative to parallelization with Stage 3 is to perform Stage 1 and Stage 2 concurrently. Unfortunately, this is not possible because Stage 2 relies on the availability of the latest world state information and such changes might alter the behavior of the game environment.

Ignoring the cost of the copying that is necessary for concurrent execution of Stage 3 and Stage 1, another factor suggests this approach is not advantageous. At issue is the fact that Stage 3 is known to be computationally intensive. Therefore, for performance issues, it would be preferable to allocate any spare resources during Stage 3 computation to improving the parallelism in Stage 3 itself. Provided the load is reasonably balanced between threads in Stage 3, there would only be minimal savings in starting stage 1 slightly earlier. When the impact of the game state copying is considered, it is unlikely that any significant gains would be achievable.

Given these factors, it was decided that Stage 1 would remain sequential. If further analysis indicated that this was severely limiting performance, then this decision would be reassessed.

3.5 Client Support for Large Player Counts

The focus in this project is on increasing the client capacity of the QuakeWorld server. However, it was found that there are other factors aside from server computation that limit the scalability of the game. In particular, it was necessary to make changes to the standard clients to enable support for more than 32 clients. The current standard clients are hard coded to only handle 32 clients. While this limit was easily removed, but other issues arise arose when increasing player count. As more players enter the game, there is more potential for interaction and this means more information must be transmitted between the server and the clients. To enable this increase, the transmission packet sizes were enlarged. Along with this change it was necessary, to increase buffer sizes throughout the client and server to handle message processing for these large packets.

Each client also performs move prediction for all the items a player may see in the game. This prediction makes the game world appear realistic even in the face of occasional network delays. With more players in the game, each client must be capable of handling move predictions for a greater number of visible players. Therefore, the clients were also modified to enable higher capacity move prediction.

It had been hoped that compatibility with the original standard clients would be preserved but unfortunately, because of these changes the QuakeWorld server will only function with clients that are updated to handle large player counts. This highlights one of the weaknesses in the design of QuakeWorld which is that it was not designed for scalability. This makes modifications to support more players unnecessarily difficult. With a slightly different design philosophy, it would have been possible to facilitate later extensions of the game capabilities.

4 Experimental Platform

There are two aspects to testing the parallel game server. First, it is obviously necessary to have a parallel platform on which the server can run. Finally,it it is also necessary to be able to generate sufficient number of clients as necessary to stress the server. This section examines both these issues.

4.1 Generic Threads Package

It was decided that the shared memory model would be used for parallelization. Since the target platform was not initially known, a generic threads package was created to abstract any platform specific details from the actual parallelization. Using this generic threads package allows the same interface to be used for running pthreads on a Shared Memory Multiprocessor (SMP) and for using Treadmarks over a cluster of machines. The package provides general functions for thread creation, mutexs, conditions, semaphores, and barriers. Compile time flags then configure the package for the appropriate platform.

The package was fully tested using a simple parallel program, but only the pthreads implementation was used for this project. Treadmarks requires the use of two functions to create shared memory and to distribute the contents of that shared memory [TM]. With the abundance of global data structures used in the QuakeWorld Game Server, it would be a significant undertaking to modify QuakeWorld to work with Treadmarks. It also believed that, with the fine grain data accesses prevalent in the server, there would be little gain from using Treadmarks. Therefore, for this current project, only the pthreads implementation will be used.

4.2 Client Machines

For many parallel applications, measuring execution time is an adequate approach for determining the improvements due to parallelization. Unfortunately, for game servers such as QuakeWorld, execution time is not an appropriate metric. The server spins in a loop processing messages as they are received. It does not wait for messages from all clients. Therefore the server may execute many frames per single client frame (where a client’s frame is approximately 32 msec since the QuakeWorld client operates at 31.25 frames per second by default). Once the server receives a message it will start its frame, but as that is taking place more client requests are received, and therefore the server will start a new frame as soon as it finishes the current frame. The server frame can range from 0.5 milliseconds to tens of milliseconds depending on the number of requests to be processed. A better metric is the server’s response rate.

The server response rate metric measures the ability of the server to send as many replies as client requests it receives. Unfortunately, this means that many actual clients must be used to saturate the server. As seen in the work by Abdelkhalek [ABM01], the sequential server can handle 96 clients if running on an 800 MHz machine. This causes two main issues. These issues are the need for players to generate traffic and the need for computers on which the clients can run.

To resolve the first issue, virtual players created from files containing pre-recorded moves were used to simulate real players. This is similar to the approach used by [AB04]. Next, sound and video were disabled to facilitate running multiple clients on a single machine. However, the game remains computationally intensive and only a few clients can be active on a single computer. To measure the limitations in generating client traffic, the output transmission rate from a single dual processor Athlon 1800 was observed and the transmission rate is shown in Figure 7. The transmission rate is expected to be 31.25 packets per second. Clearly, the single machine can only support up to five clients with ideal behavior and can only offer satisfactory behavior when running between 5 and 10 clients. Beyond this threshold, the behavior of the clients becomes unacceptable.

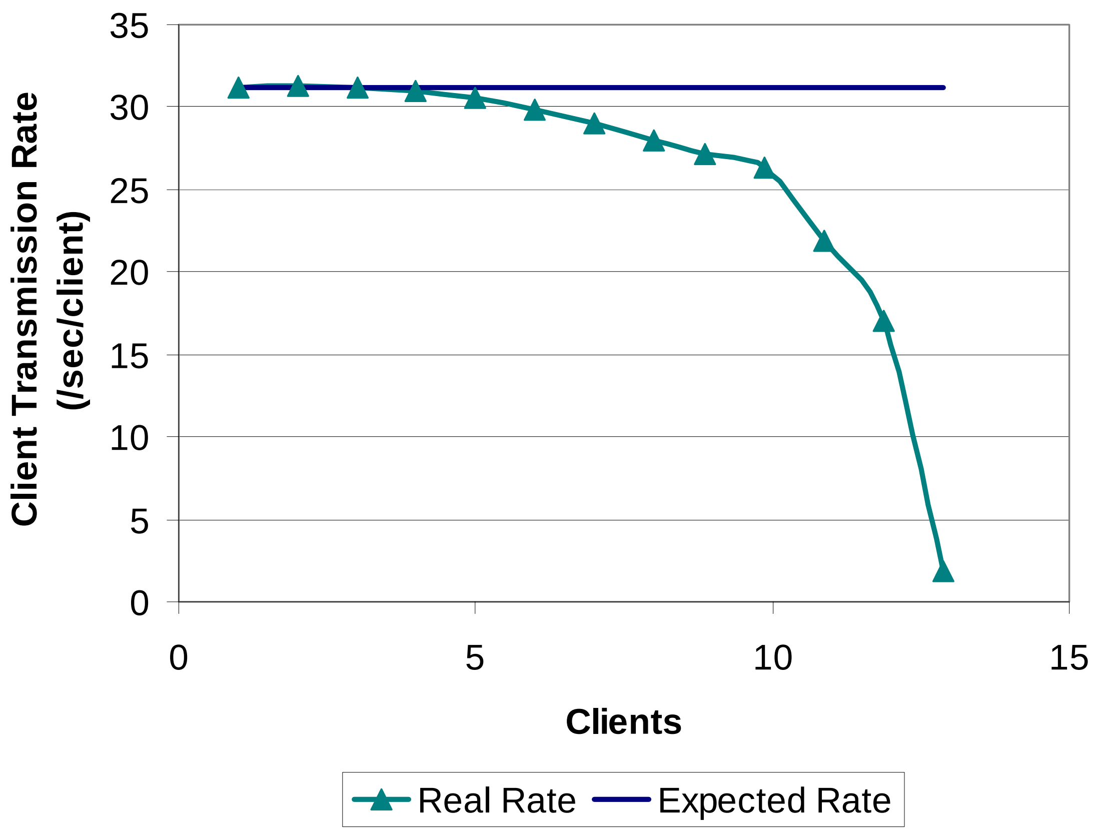

Figure 7 – Client Send Rate as the number of clients on the machine increases

A set of eight of the aforementioned machines was obtained for use in this project. However, generating clients using this cluster of machines is a daunting task. Using the console or remote desktop to manage the machines is exceedingly time consuming and not reliable since it depends on human interaction. To simplify this process, a Distributed Client Manager (DCM) program was created.

The DCM allowed a user to run one program, which communicates with programs running on each of the client machines, to start and stop QuakeWorld clients. Any command the user enters entered into the main program is transferred to the client machine using a TCP connection. Another useful command in the DCM was the ability to change the server IP address used by the QuakeWorld clients. This allowed for quick and easy testing of the server on a variety of platforms.

The DCM also resolved many problems encountered when running multiple Quake clients from a single machine. One such problem is that the client machine would become unresponsive if the many QuakeWorld clients crashed. With the DCM it was still possible to communicate with the machines and kill the offending processes. Occasionally, the system would not be able to recover from the crashes of the client and the machine would require a reboot.

To address this problem, a second program, called the Quake Program Manager (QPM) was created to reboot the computers on command. The QPM used the windows reboot facilities in order to have a “safe windows” reboot for the computers. This new program was also created to be a windows service so that it would automatically restart when windows restarted. The QPM also had the ability restart the DCM. The QPM was put on all the machines and was again controlled by a single program using a TCP connection.

Changes to the QuakeWorld clients on the machines were also necessary occasionally. To facilitate the distribution of these builds a simple FTP program was created to transfer any file to all the machines quickly and easily.

4.3 Server Machines

As seen in the previous section in Figure 7 a single client machine can only handle around 9 or 10 Quake clients running concurrently. Since, only eight computers for running clients were available, it meant that only 72-80 Quake clients could be run concurrently. This presented a significant challenge because as mentioned previously the sequential version can handle 96 clients on an 800MHz machine [ABM01]. This means more client computers are needed, or a much slower server machine is required. Four machines were tested in order to check their feasibility of them being used as the potential servers. The specifications of these machines are provided in Tables 1 through 4. Figure 8 illustrates the results of trying to saturate each possible server. Platform D could not be saturated with the limited number of clients available. Platform C was saturated but at a relatively high number of clients meaning that parallelization might increase the capacity beyond the client-generating capabilities of this project. Platforms A and B however, appeared to be easily saturated and therefore Platform A was used for future performance tests.

Table 1 Experimental Platform A

Model	Sun Ultra Enterprise 450 Model 4300
Processor	4 – 296 MHz Sun UltraSPARC-II
Memory	1.5 GB
OS	SunOS 5.8
Compiler	gcc 2.95.2

Table 2 Experimental Platform B

Model	Sun Ultra Enterprise 450 Model 4400
Processor	4 – 400 MHz Sun UltraSPARC-II
Memory	2.0 GB
OS	SunOS 5.8
Compiler	gcc 2.95.2

Table 3 Experimental Platform C

Processor	533 MHz Intel Pentium III (Coppermine)
Memory	512 MB
OS	Linux 2.4.18-27.7.x
Compiler	gcc 2.95.2

Table 4 Experimental Platform D

Processor	4 – 2666 MHz Intel Xeon
Memory	4.0 GB

OS	Linux 2.4.18-27.7.xbigmem
Compiler	gcc 2.95.2

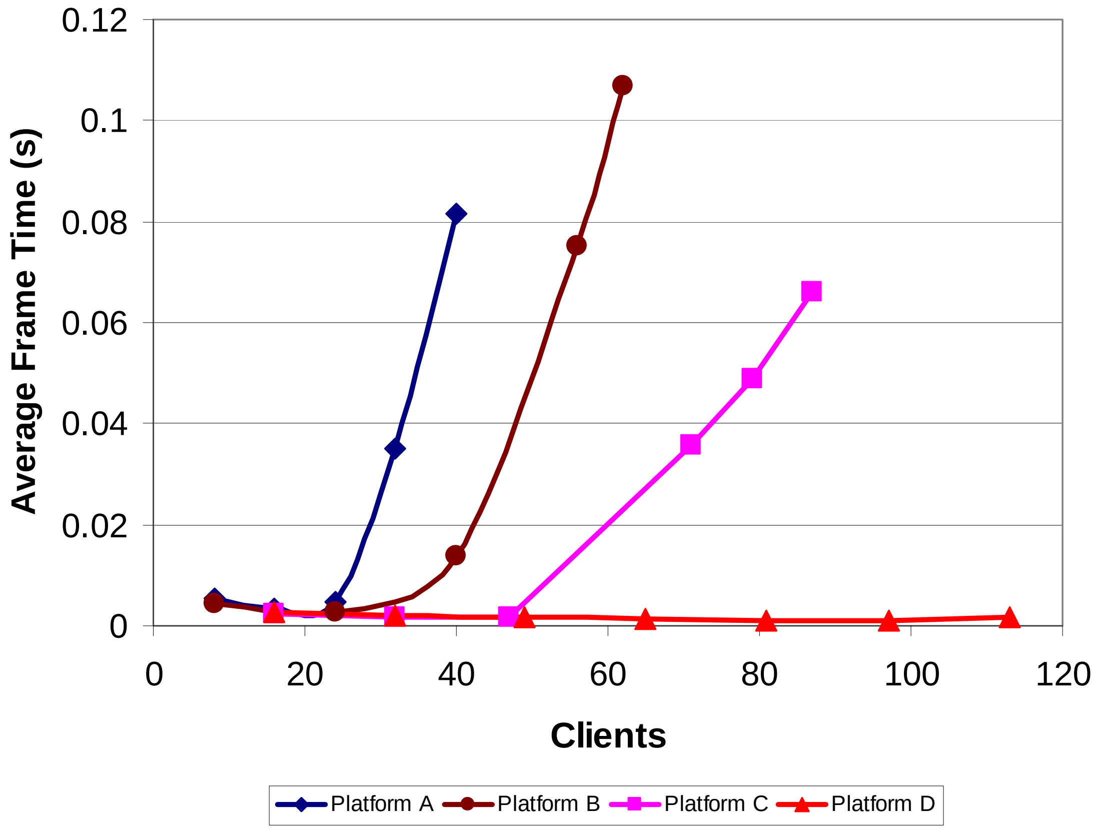

Figure 8 – Performance of Potential Servers

5 Results

Now that a server that can be saturated with a limited number of clients available has been found, the benefits of the parallel server design can be analyzed. A general analysis of the benefits of the server's performance will first be presented. Then, to better understand the limitations and benefits of the parallel design, the results for each of the parallelized stages will be examined in detail. For all test results, Platform A, the 4 x 296 MHz UltraSPARC-II machine, will be used. The client count in all tests is increased in multiples of eight until the server is overloaded. The server is considered to be overloaded when it drops over 20% of the client packets. Testing beyond this point is not informative since the server is no longer delivering acceptable performance to the user.

5.1 Server Capacity

The primary goal in parallelizing the QuakeWorld server was to increase the number of clients that the server can support. However, before examining the server performance, it is important to review the expected behavior of the server. As shown previously in

Figure 1, the server processes one frame after another, executing each of the stages in the computation. In each frame, responses are sent to all the clients whose requests were processed during this frame. This behavior of sending a packet to a client for every request received by the client is the core feature of a functioning QuakeWorld game server. Without the response from the server, the client does not have the latest game state information. Maintaining the server in a state where a response is sent for each packet received is essential for ensuring acceptable behavior on all client machines.

Given this desired behavior it is appropriate to measure the server's capacity by monitoring the ratio of the server's receive rate to the response rate. However, the receive rate is relatively constant since all clients are configured to run at 31.25 frames per second and requests are sent to the server every client frame. Therefore, it is sufficient to examine the number of responses sent by the server every second. (The rate at which packets are received by the server is monitored to ensure it is acceptable but it is not reported here) As the number of clients increases, the aggregate response rate of the server should also increase as the server handles the requests from the additional clients. Figure 9 illustrates this expected behavior along with the results from a sequential implementation of the server and the multi-threaded parallel version.

The figure reveals that the multi-threaded versions provide a significant increase in capacity. The sequential version labeled "1 thread" saturates at approximately 24 clients since at higher client counts the response rate no longer meets the expected performance. The 2, 3 and 4 threaded versions all significantly increase this limitation. Acceptable performance is maintained until 40 clients and, at 48 clients, the response rate has only degraded slightly. This increase in capacity to 40 clients from 24 clients is an increase of 66%. It is also significant to note the performance degrades much more gracefully in the multi-threaded versions compared to the sequential version, in which the performance drops sharply after the server is saturated.

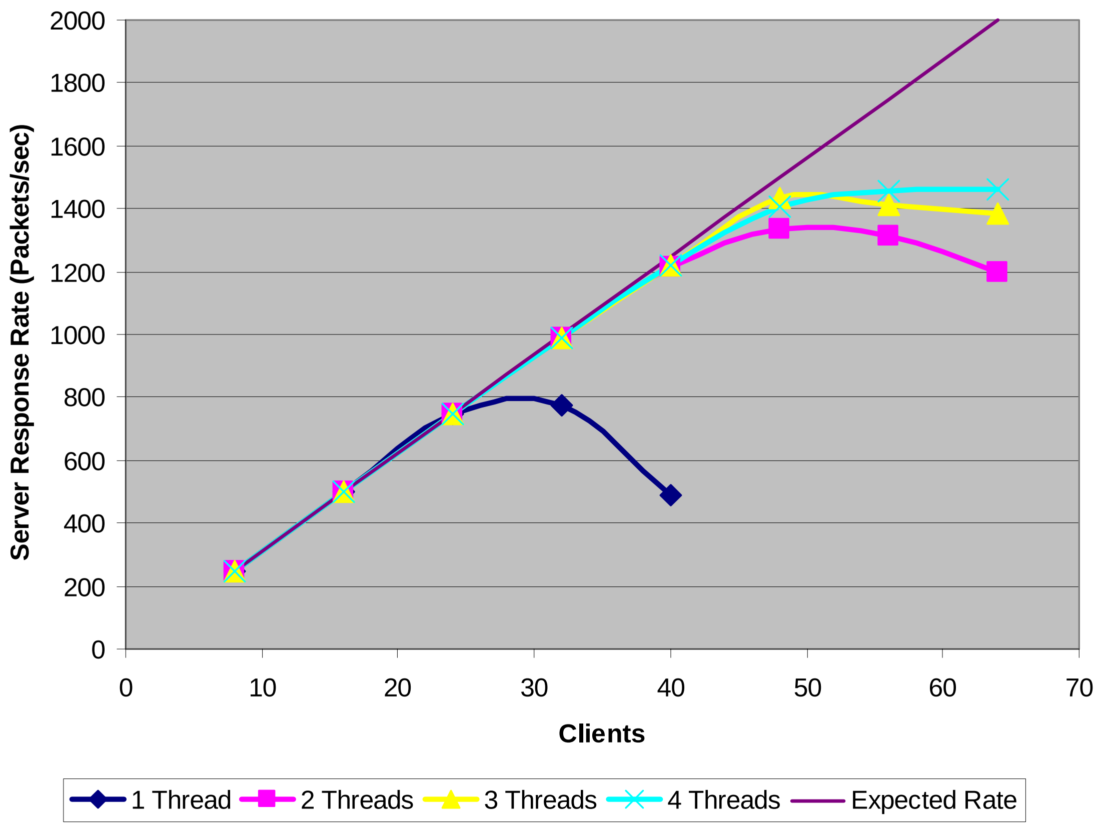

Figure 9 – Server Response Rate with Improved Stage 2 Parallelization

Given this capacity increase, it is interesting to examine where the execution time is spent. A high-level breakdown of the execution time is shown in Figure 10 and Figure 11. In the figures, Idle time is the time spent waiting for client packets, Intra-Frame Wait is the time spent waiting at barriers anytime during the computation and Stages 1, 2 and 3 track the time spent on actual computation in these stages of a server frame. In Figure 8, the normalized average server time spent per server frame is shown. This figure demonstrates the relative importance of each stage in the computation. The idle time is the time spent waiting for a packet from any client. As expected, Stage 2 and Stage 3 dominate the execution time. The time required for Stage 1 remains relatively constant for all player counts, which was expected from the prior work in [AB04]. The figure also highlights the impact of the design decision to target a significant number of players. As a result of this decision, the overhead of the parallel design means that for low client counts, the server actually requires more execution time per frame. These issues will be discussed in greater detail in the proceeding two sections where the parallel design of each section in the server is analyzed.

This data also supports the previous analysis of the server capacity based on the response rate. The server is clearly not saturated if time is spent waiting for client requests. In the case of the sequential server, saturation appears to occur at slightly greater than 24 clients since at 24 clients idle time remains while at 32 clients the single threaded implementation is overloaded. The multi-threaded versions do not saturate until much higher player counts as can be seen by the idle time when processing 40 and 48 clients.

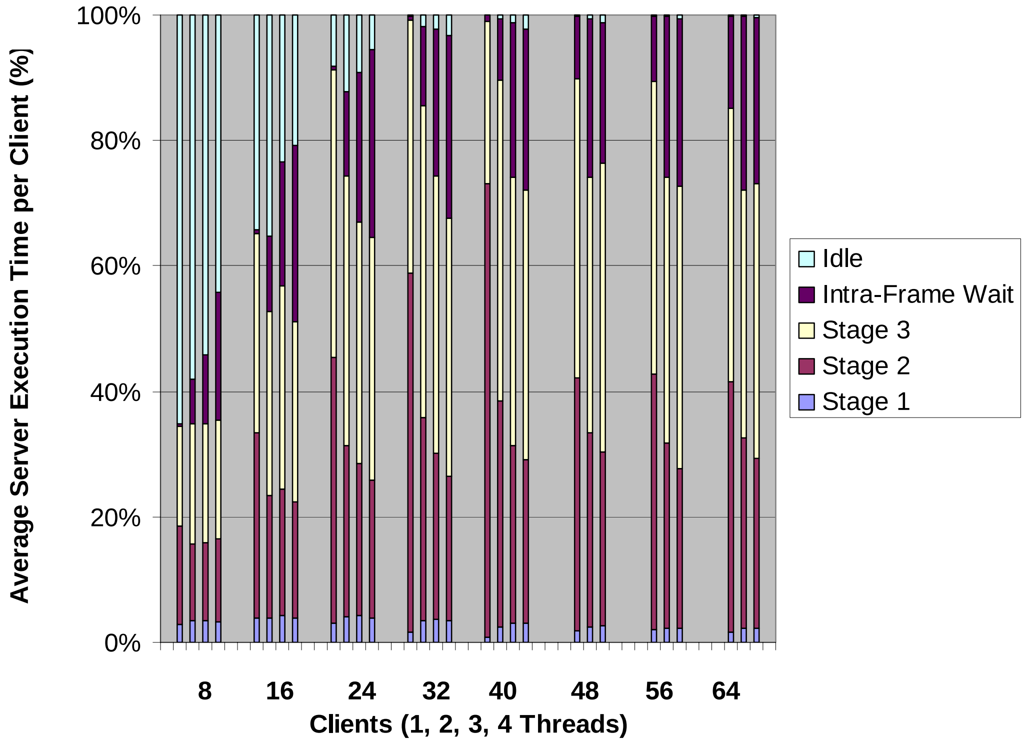

Figure 10 – Average Server Execution Time Breakdown with Optimized Stage 2

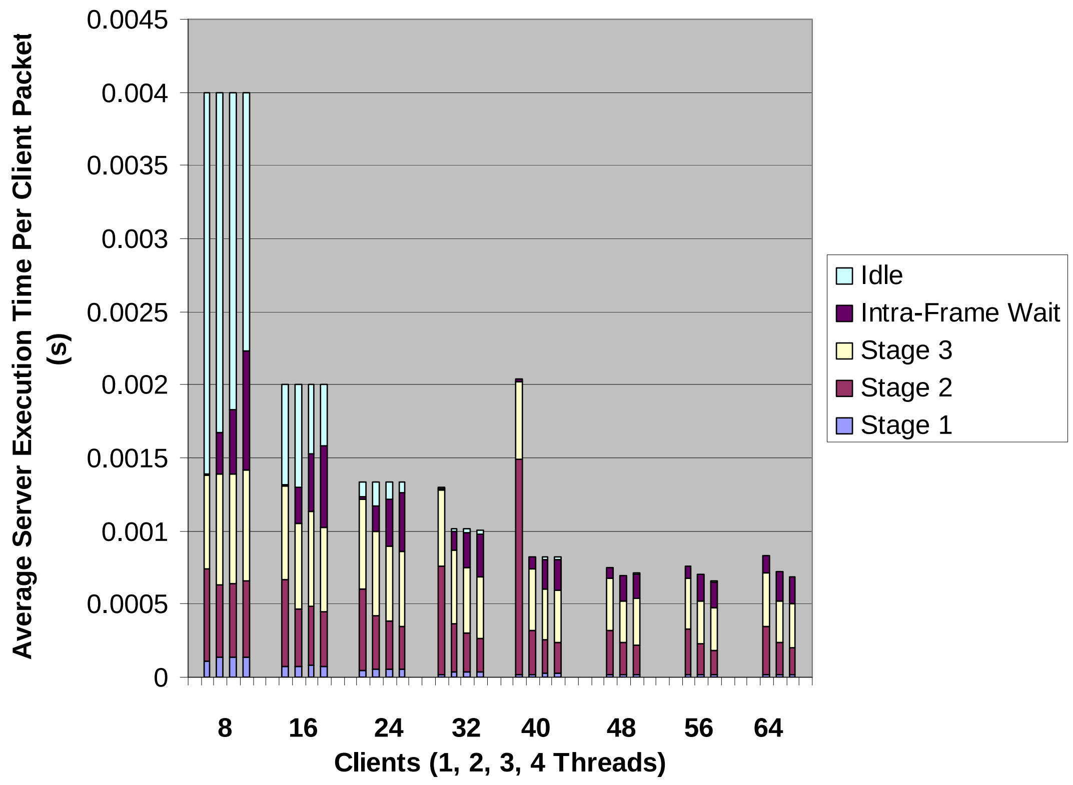

Figure 11 – Average Server Execution Time per Client with Optimized Stage 2

Given this capacity increase, it is interesting to determine what role the two parallelized stages played in improving the server's client capacity. Figure 12 plots the response rate for a varying number of threads across a varied number of players. It compares the response rate when the parallel designs of both Stage 2 and Stage 3 are used to the case when only Stage 3 makes use of the multiple threads while Stage 2 is complete sequential. As expected, without the multi-threaded implementation of Stage 2 the capacity increase is not as significant. When just Stage 3 operates in parallel the response rate has degraded from the ideal rate at only 40 clients. Therefore, the server can only be considered functional when handling up to 32 clients. However, this is still an increase of approximately 33% over the sequential implementation. When both Stage 2 and 3 make use of parallelism, the capacity increase is much greater as discussed previously. Performance remains acceptable even at 40 clients and, at 48 clients, the response rate has only degraded slightly from the expected behavior. It is interesting to observe that when Stage 2 is parallelized, the performance degradation when the server capacity is exceeded is much more gradual than the case when just Stage 3 is parallelized. This graceful degradation is the preferred behavior since it places a less severe restriction on server capacity. This is in contrast to the situation when only Stage 3 is parallelized where it is essential that the capacity is never exceeded.

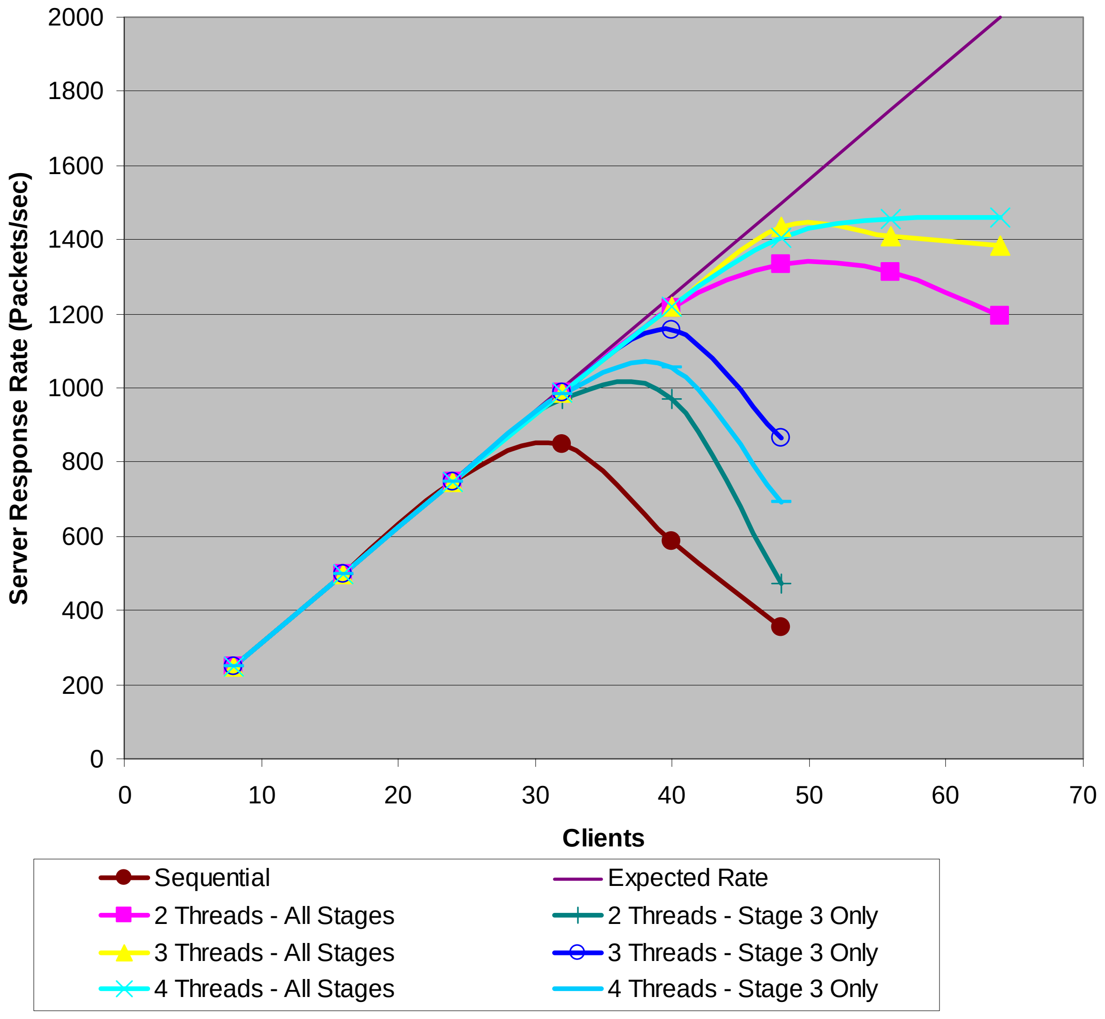

Figure 12 – Contribution of Stages to Capacity Increase

5.2 Analysis of Stage 2

The results from the previous section demonstrated that a significant capacity increase is possible thanks to this new parallel design of the second and third stages of computation. To better understand, the performance advantages and limitations of this design, this section analyses the performance of Stage 2. The two approaches to parallelization for Stage 2 are considered, the basic approach with a single sequential stage and the optimized approach in clients on boundaries are processed concurrently where possible.

The basic parallelization approach allows QuakeWorld game server to process client messages in parallel only when four threads are available in the system. To observe the effect this approach has, consider Figure 13. In Figure 13, the time required to process events from a single client for a given number of threads is shown. This time is then divided between stage one, stage two, stage three, the intra-frame and the idle times. The graph shows that when 8 clients are playing the game, the original server performs best. In comparison, for four threads the system takes much more time processing client requests in stage two. This is because with so few clients the overhead of locking and preprocessing client messages is dominant. This trend continues for up to 32 clients, progressively decreasing the advantage the original server has. When 40 clients are in the game, the original server can no longer process client messages sufficiently fast to maintain the client frame rate. The time taken by the parallelized stage two continues to decrease. The parallelized server saturates at 56 clients and is no longer able to provide sufficient processing speed to support more clients.

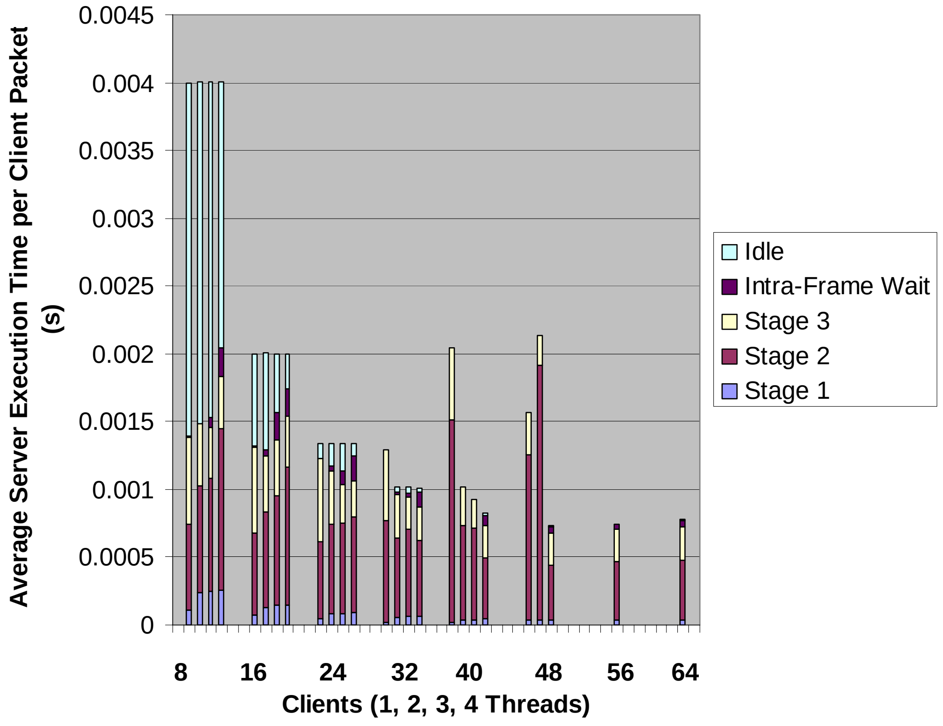

Figure 13 – Average Server Execution Time per Client with Basic Stage 2

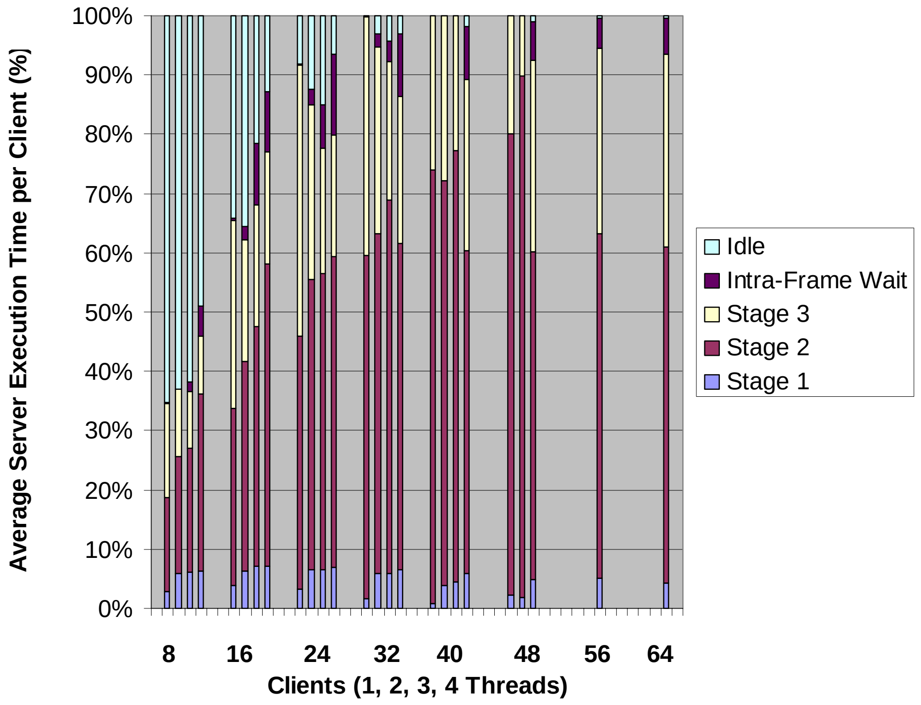

Figure 14 – Average Server Execution Time Breakdown with Basic Stage 2

This behavior is explained by Figure 14 which shows the average percentage of time taken by each stage of processing. This graph shows that while the sequential server's stage two processing time increases linearly, the parallelized stage two processing increases at a much lower rate. This demonstrates that as the number of clients increases, each thread has similar number of clients to process. The only exception is thread 1 which processes clients on area boundaries as well.

The tests were repeated for the optimized Stage 2 design and the results were shown previously in Figure 10 and Figure 11. With this optimized approach, Stage 2 can make use of two or more threads. As a result, there is no longer the large jump in performance from three threads to two threads. Instead the Stage 2 execution time decreases with increasing number of threads as expected. It is instructive to look further at the performance of this optimized Stage 2 design. To assist in this analysis Figure 15 highlights the contributions to execution time for each component of Stage 2. This is shown for a varying number of players and a varying number of clients. As before, it is clear that for low number of clients a single threaded version of the server is better. As the number of clients increases, multithreaded server begins to function better. At 32 clients, the average time spent processing a single client reaches almost 0.8 ms. This means that the server spends about 0.8 ms x 32 clients = 26 ms processing all the client messages. Since all clients require full updates every 32 ms, this does not leave sufficient time for stage three processing. However, at this player count, the parallelized Stage 2 starts to show its benefits. In fact, for 2-4 threads the parallelized stage two shows satisfactory performance until about 48 clients. At this point the performance of the two threaded version of stage two begins to degrade, while the time spent on stage two processing for four threads is still low. At 64 clients, the four threaded implementation begins to suffer from increased wait time for locks and performance suffers as a result. It should be noted that the four threaded version of the server could only be tested on a machine with only three unloaded processors available.

It is also important to note two key features of this parallel design: the low level of contention for locks and the short waits at barriers. The past implementation in [AB04] suffered from significant lock contention. The data in Figure 15 illustrates that while lock time is increasing with player count, it remains relatively minor. This lower contention was achieved thanks to the area-based thread assignment. To perform this assignment, it is necessary to perform the initial sequential preprocessing. However, the time for this preprocessing stage is relatively small which demonstrates that the trade-off of more initial sequential computation for reduced locking contention was worthwhile.

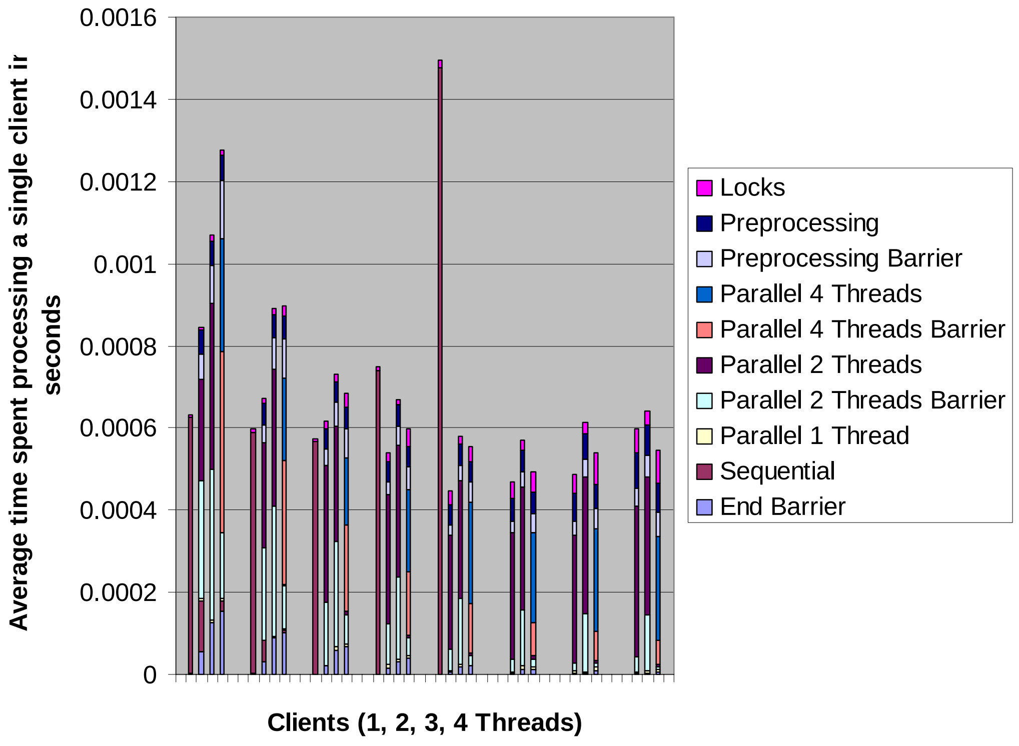

Figure 15 – Optimized stage two performance breakdown

These results demonstrate that with the optimized parallel Stage 2 design the server can process almost two times as many clients as the sequential version of the server. While further testing on larger number of clients needs to be done to verify this speedup, the results obtained here are encouraging.

5.3 Analysis of Stage 3

With Stage 3 requiring potentially 40% of the execution time on a saturated server, it is important to better understand the impact of parallelization on Stage 3. To assist in this understanding, Figure 16 illustrates the breakdown of the execution time specifically within Stage 3. This execution time is the average across all threads and it is reported as the average per client packet processed.

This data demonstrates a few interesting trends. First, examination of the execution times for eight players reveals that the multi-threaded versions take longer to execute than the sequential implementation. While unexpected, the reason for this is clear. With only eight players, typically only one client is processed in each server frame. As a result, there is no potential benefit from the parallelization scheme used in this stage since only one client message is generated. The use of more threads simply adds additional overhead since most threads spend the majority of their time waiting at synchronization barriers. This can be seen in the figure by the large amount of time spent at the Internal Barrier and End Barrier. However, the server was designed to handle many clients and as the number of clients is increased the gains from this parallelization scheme become evident. Once the number of clients exceeds the sequential server's capacity at approximately 32 clients, the multi-threaded versions now spend less execution time per client than the sequential implementation. It is this decrease in execution time that enables the capacity of the server to increase.

In the design of the parallel implementation of Stage 3, it was decided that the initial update portion of the stage would be performed sequentially. The results shown in Figure 16 indicate the validity of this decision since compared to the second part of Stage 3 computations,; the time required for Part 1 is much less significant. At low player counts the time spent on Part I of Stage 3 is not trivial but, for these limited player counts, execution time is not a concern. At high player counts, many clients are processed in a single server frame. This makes the execution time for Part 1 less significant since it is effectively divided amongst all the clients processed in the current frame.

It is interesting to examine the time spent at barriers in Stage 3 since it is this time that may limit the gains of the parallel server implementation. The first part of the computation in Stage 3 is performed sequentially. While this computation is performed, the other threads wait at the Internal Barrier. The time spent on these two components, Part I and Internal Barrier is significant at low player counts but, again, at high player counts the time is relatively minor. The behavior of the time spent at the End Barrier is also similar but even at high player counts the time spent at the barriers is significant. Unlike the first barrier, where it was known that there would be a load imbalance between threads, this was not expected for the end barrier. Therefore it is important to consider the behavior of each thread and this is shown in Figure 17. In the figure the components of the server execution time are shown for each of the threads when operating with four threads.

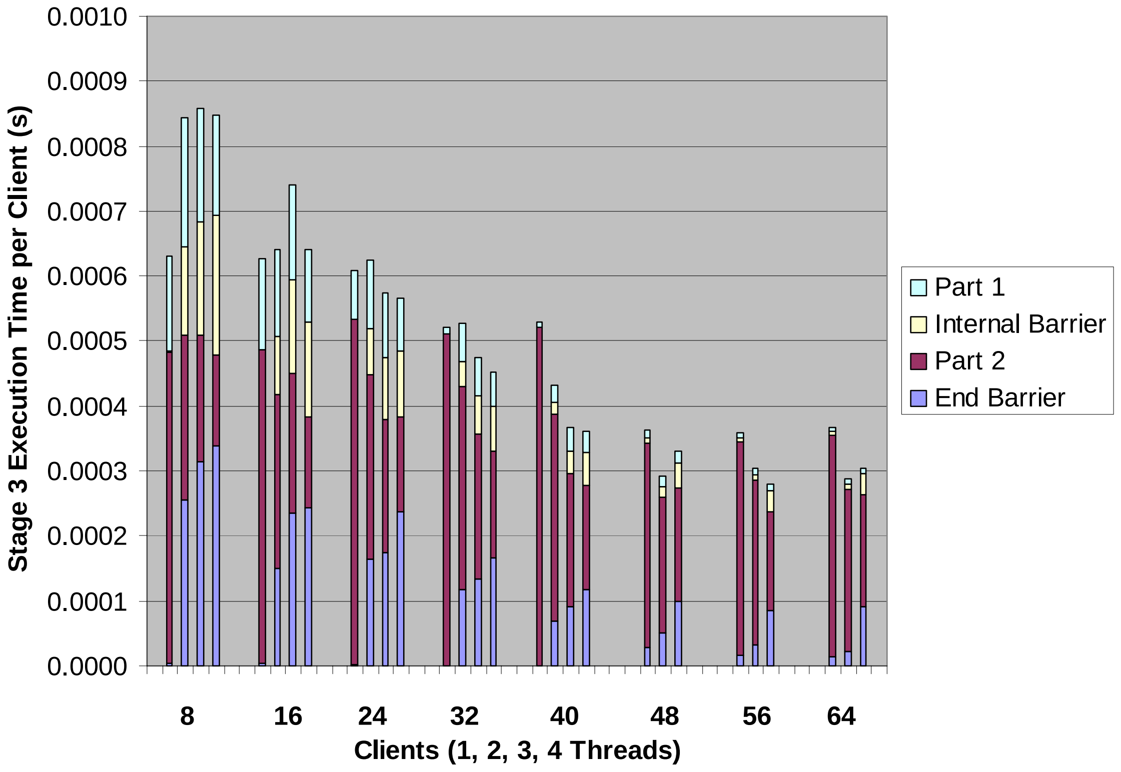

Figure 16 – Breakdown of Stage 3 Execution Time

As expected, only threads 2 through 4 spend time at the internal barrier. The time is spent at this barrier while Thread 1 performs Part 1 of the Stage 3 computation. All threads appear to spend time at the Stage 3 stop barrier. The time appears to be relatively balanced between threads as the variation between times spent at this barrier is less than 5%. Given this balance between threads, it is somewhat surprising that all threads spend over 10% of the average execution time at this barrier. A naïve analysis suggests that this should not be the case. If all threads are at a barrier then execution should proceed to the following stages. There is some overhead incurred in simply entering and exiting a barrier but the many other barriers in the program that require much less total time demonstrate the overhead is well below 10% of the execution time.

The significant time at the barriers is instead occurring not because of a load imbalance within a single server computation frame, but because of the imbalance between server frames. During each server frame, Stage 3 must generate replies for every client from which a request was received. At low player counts, only one client may be received per frame. Therefore, as described earlier, there is no potential for gain using the current parallel implementation. As a result in these cases, on average only one thread performs any processing while the other threads wait idly at the Stop Barrier. During the next server frame, a different thread will likely be active while the previously active thread will wait at the barrier. The net effect of this is that on average all threads end up spending a significant portion of the time waiting at the barrier.

The design of the parallel Stage 3 engine was intended to enable high player counts. It was thought that as the player count increases and more clients are processed per frame the cyclic distribution of clients to threads would enable efficient parallelization in most cases. At 64 clients, the barrier time has been slightly reduced percentage-wise from the 8 client case; however, the imbalance between server frames clearly remains since over 10% of the execution time is still spent at the barrier. This indicates that an imbalance remains between server frames and in each frame a different thread causes the delay at the barrier.

The reason for this imbalance is not entirely clear. One possibility is that the times when responses are received from clients may not be completely random. Since clients are created in groups, the responses from the server may be received in the same groups. If these groups are not randomly distributed across all possible client positions, they may load only one thread while the other threads remain lightly loaded. This was known to be the worst case for Stage 3 of the server. To address this limitation in the future, it may be necessary to track active clients during each server frame. Then the active clients can be distributed among the available threads. That approach should enable more balanced load distribution and thereby minimize the time spent waiting at barrier at the end of Stage 3.

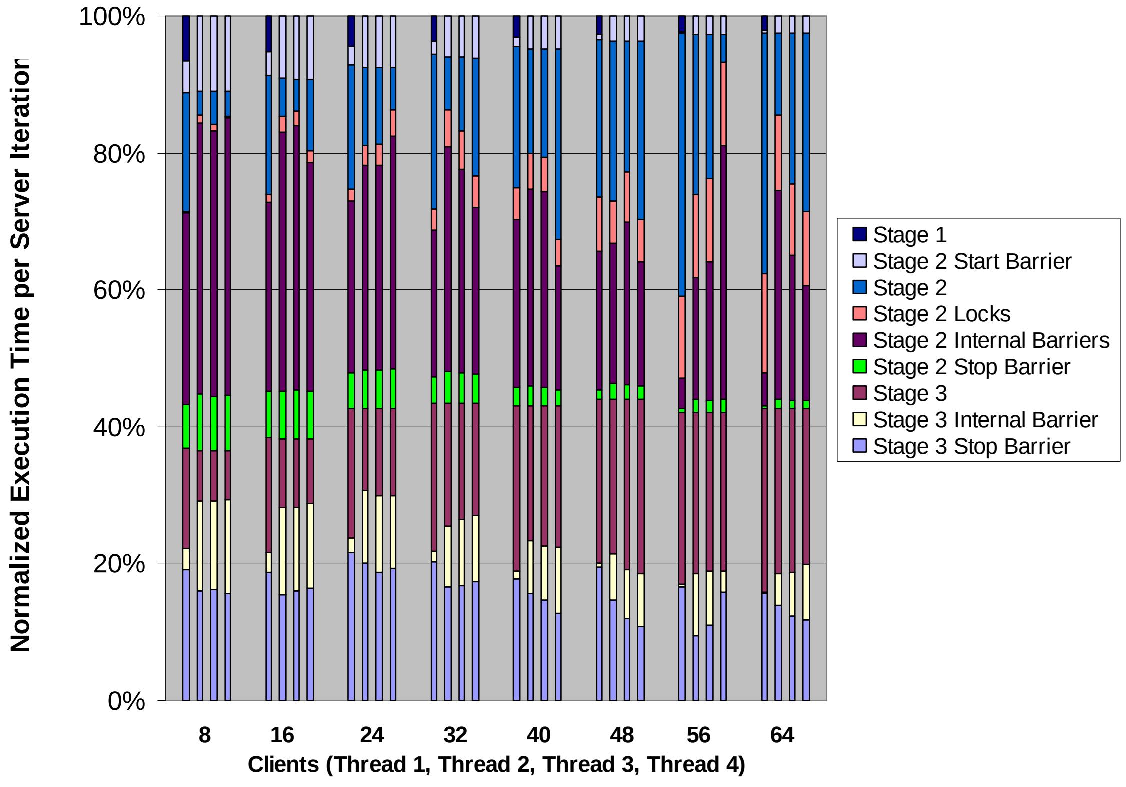

Figure 17 – Load Balance between Threads for 4 Threaded Server

6 Conclusions and Future Work

This project has successfully increased the capacity of the QuakeWorld game server. Running on a platform with four 296 MHz UltraSPARC-II's, the capacity of the parallel version with 4 threads is approximately 66% greater than the capacity of the sequential game server. This gain was accomplished by parallelization of the two most computationally intensive portions of the QuakeWorld server, request processing (Stage 2) and reply processing (Stage 3). In Stage 2, a novel area-based approach for assigning clients to threads was found to enable a capacity increase without suffering from the significant wait and lock contention problems that plagued past implementations. The parallelization of Stage 3 was based on a more standard client-based approach with all the potential clients divided statically among the available threads. This scheme did increase the capacity of the QuakeWorld server; however, it was found that this was not ideal since a significant portion of the runtime was spent waiting at synchronization barriers.

This effort to parallelize the QuakeWorld server also provided some insight into the challenges in parallelizing interactive multiplayer game servers. One major stumbling block in parallelizing the QuakeWorld server is that it was not engineered to handle a large number of clients. Interactive games become increasingly interesting as more players interact. Therefore, it is foreseeable that future increases in capacity would be desirable. With hard limits and the encoding of some game information in bit fields that could not be extended, increasing the capacity of the QuakeWorld server was non-trivial. If future designs consider the needs when increasing capacity, the process could be made much more straightforward.

Games such as Quake are often popular because of their high level of configurability. Specifically in QuakeWorld, new maps and items can be easily created because this basic game world information is separated from the actual game engine. The game engine serves as an interpreter for that separately stored game or map specific information. Unfortunately, that modularity potentially limits parallel optimizations. Since at the game engine level one cannot be certain of the actions of any interpreted functionality, it is necessary to perform conservative locking. If future servers are to be designed for eventual parallelization then this modular approach may need to be reconsidered. Alternatively, a more structured approach to modularity may be needed so that the impact of any actions is clearly defined. If the impact was known then locking might not be necessary in all cases.

The difficulty in parallelizing Stage 2 of this computation resulted partially from the inability to effectively partition the game state data. The approach used in this work assigning threads to different areas of the game environment was a partial attempt to address this. However, with most information fundamentally stored in a central game state structure locking remained essential. To enable the scalability, the game structures must be designed for effective partitioning. Retrofitting such radical changes into the game server were well beyond the scope of this work; however future game servers should consider the desire for parallelism when planning the game state structures.

The results from this parallel implementation of the QuakeWorld Server suggest many interesting avenues for future work in parallelizing this server. In the current work, it is assumed that parallelization gains achieved on the UltraSPARC-II machine will scale to faster SMPs. This assumption was made by necessity since it was not possible to generate enough clients to saturate more powerful servers. In the future, if more machines for generating clients were available, new faster SMP platforms could be used. This could confirm the client capacity gains observed from the current design.

Further improvement of stage two may also be possible by reducing barrier wait times. The current implementation conservatively waits for all threads to finish before proceeding to the next level of processing. However, there may be cases where two threads finish early. Once complete, it is possible that the players on the boundary between areas handled by these threads could then be processed without waiting for all the other threads to complete processing of their areas.

It would also be interesting to explore the potential of using a more dynamic approach for player to thread assignment. Currently, players are allocated exclusively based on their area within the map. This process is repeated with larger and larger areas to handle players bordering two separate areas until all players are handled. However, this approach may not be ideal if clients congregate in particular areas. Such congregation could lead to load imbalances. Therefore, in such cases, it would be interesting to dynamically alter the granularity of the areas assigned to each thread. For example, if half the map contained very few clients, then one thread could be allocated to that entire half while the other half would be split among all remaining threads. The difficulty in implementing such a scheme is that it increases the amount of preprocessing needed to assign players to threads. This preprocessing is by its nature sequential and therefore it could limit the performance of the system.

Improvements to the Stage 3 parallel design are also possible. As described earlier, the current approach suffers from a load imbalance between server frames with one thread having a higher load in frame than it does in another. To address this limitation, a dynamic assignment of clients to threads may be appropriate. The server could track which clients are active per server frame and then divide those active clients among the available threads. This should enable a reduction in barrier wait time and therefore enable even further gains in client capacity.

References

[AB04] Ahmed Abdelkhalek and Angelos Bilas. Parallelization and performance of interactive multiplayer game servers. In *Proceedings of the 2004 International Parallel and Distributed Processing Symposium (IPDPS2004)*, pages 72–81, April 2004.

[ABM01] Ahmed Abdelkhalek, Angelos Bilas, and Andreas Moshovos. Behavior and performance of interactive multi-player game servers. In *Proceedings of 2001 IEEE International Symposium on Performance Analysis of Systems and Software,* November 2001.

[BT01] Paul Bettner and Mark Terrano. 1500 Archers on a 28.8: Network Programming in age of Empires and Beyond. Presented at GDC2001. March 2001.

[DW]	Doom World FAQ. http://www.newdoom.com/newdoomfaq.php#13
[ED03]	Abdulmotaleb El Saddik, Andre Dufour. Peer-to-Peer Suitability for Collaborative Multiplayer Games. In *Proceedings of the Seventh IEEE International Symposium on Distributed Simulation and Real Time Applications*, Delft, Netherland 25th - 28th Oct. 2003.
[EQ]	Everquest Home Page. http://eqlive.station.sony.com/
[iS]	id Software. http://www.idsoftware.com/.
[iSD3]	id Software. Doom 3 Home Page. http://www.doom3.com/.
[GZLM04]	Chris GauthierDickey, Daniel Zappala, Virginia Lo, and James Marr. Low Latency and Cheat-Proof Event Ordering for Peer-to-Peer Games. *The 14th ACM International Workshop on Network and Operating System Support for Digital Audio and Video*, June 2004.
[KZ02]	Aaron Khoo and Robert Zubek. Applying Inexpensive AI Techniques to Computer Games. In *IEEE Intelligent Systems*: http://www.computer.org/intelligent/, July/August 2002.
[MSH03]	Mark R. Mine, Joe Shochet, Roger Hughston. Building a Massively Multiplayer Game for the Million: Disney's Toontown Online. In *ACM Computers in Entertainment*, Vol. 1, No. 1, October 2003, article 06.
[TM]	TreadMarks. Concurrent Programming with Treadmarks. http://www.cs.rice.edu/~willy/papers/doc.ps.gz
[TT02]	Amund Tveit and Gisle B. Tveit. Game Usage Mining: Information Gathering for Knowledge Discovery in Massive Multiplayer Games. In *Proceedings of the 3rd International Conference on Internet Computing, CSREA Press*, Las Vegas, USA, June 2002, pp. 636-642.
[WWL04]	Tianqi Wang, Cho-Li Wang, Francis Lau. Grid-enabled Multi-server Network Game Architecture. Presented at the *3rd International Conference on Application and Development of Computer Games (ADCOG 2004)*, April 26-27 2004, City University of Hong Kong, HKSAR