Load Balancing on an Interactive Multiplayer Game Server

Daniel Cordeiro[1,*], Alfredo Goldman[1], and Dilma da Silva[2]

[1] Department of Computer Science, University of São Paulo
danielc,gold@ime.usp.br
[2] Advanced Operating System Group, IBM T. J. Watson Research Center
dilma@watson.ibm.com

Abstract. In this work, we investigate the impact of issues related to performance, parallelization, and scalability of interactive, multiplayer games. Particularly, we study and extend the game QuakeWorld, made publicly available by *id Software* under GPL license. We have created a new parallelization model for Quake's distributed simulation and implemented this model in QuakeWorld server. We implemented the model adapting the QuakeWorld server in order to allow a better management of the generated workload. We present in this paper our experimental results on SMP computers.

1 Introduction

Game developers recently started a deep discussion about the uses of multiple processors in computer games. The new generation of video game consoles provides a multicore processor environment, but the current game developing technology does not use the full potential of the new video game consoles yet.

Abdelkhalek et al. [1,2,3] started an investigation of the behavior of interactive, multiplayer games in multi-processed computers. They characterized the computing needs and proposed a parallelization model for QuakeWorld, an important computer game optimized for multiplayer games.

The results presented by Abdelkhalek showed that his multithreaded server implementation suffered from performance problems because of their misuse of game semantics in order to create a lock strategy and because of the use of a static division of work between the available processors.

We present a dynamic load balancing and scheduling mechanism that utilizes the semantics of the simulation executed by the server. The performance analysis shows that we are able to improve the parallelism rate from 40% obtained by Abdelkhalek to 55% of the total execution time.

This paper is organized as follows. Section 2 gives an overview of the QuakeWorld server. Section 3 describes previous efforts in parallelizing the Quake server. Section 4 presents the proposed parallelization model for Quake and Section 5 presents the performance analysis. Section 6 presents our conclusions.

* The author would like to thank The State of São Paulo Research Foundation (FAPESP, grant no. 03/10064-4) for the financial support.

2 Quake Server

The Quake game is an interactive, multiplayer action game developed and distributed by *id Software* [4]. Its release in 1996 was a important mark in the game industry because it was the first time that a game was developed using three-dimensional models in the simulation and graphics engines. Later that year, QuakeWorld was released with enhancements for Internet games[1].

The Quake multiplayer game follows the client-server architecture. One game session has a single, centralized server that is responsible for the execution of all physics simulations, for evolving the game model (that gives the semantics of the game to the simulation), and for the propagation of state modifications to all connected clients. Up to thirty-two clients can join an open game session. It is the client responsibility to collect and send the input events from the player to the server and to do the graphics rendering of the current state of the simulation.

2.1 Quake Server Architecture

The Quake server is a single process, event-driven program. The server-side simulation consists of execution of frames. The frame task is composed by three distinct stages: updating the world physics, receiving and processing events sent by clients and creating and sending replies to all active players.

During the simulation of the world physics, each solid game entity (players, bullets, blocks, etc.) is simulated by the engine. The engine simulates the effects of gravity, velocity, acceleration, etc. The request and response processing is the more computational intensive phase. It reads all messages from the server socket, simulates the events sent in the message (movement commands, jump command, text messages, etc.), applies the command to the game state and replies to the clients with the changes in the game state. Only active clients, i.e. clients that sent a message in this frame receives a reply.

3 Multithreaded Version

Abdelkhalek et al. started an effort to characterize the behavior and performance of the original version of Quake [1]. They increased the limit of players from the original 32 to 90 simultaneous players. Their experiments showed that the incoming bandwidth is practically constant and is low (a few KBytes/s) and the outgoing bandwidth depends on the number of simultaneous clients, but does not exceed a few KBytes/s. The actual performance limitation was the processing power bottleneck. With more than 90 simultaneous users, server performance started to degrade.

In his following works [2,3], Abdelkhalek presented a multithreaded version of the Quake server. In order to avoid semantic and correctness errors that could

[1] We will use the names “Quake” and “QuakeWorld” interchangeably in this text to refer this new improved version.

arise from reordering the frame computation stages or overlapping them, two invariants were imposed: (i) each server phase is distinct and should not overlap with other phases; and, (ii) each phase should execute in the original order: world processing, request processing, and finally reply processing.

Also, at this first parallelization attempt, there is no load balancing mechanism. Each client is assigned to a server thread at connection time. The server uses one thread per available CPU and clients are assigned to the threads in a round-robin fashion. All computations related to one client are executed by the same thread and each client sends their messages directly to the assigned thread.

3.1 Tasks Decomposition

The frame tasks were decomposed as follows. The authors noted that the world processing stage takes less than 5% of the total execution time of the original version regardless of the quantity of players [2] and excluded this execution from the parallelization effort. So, there is a synchronization barrier before and after the world processing stage and only the coordinator thread executes this phase.

After that, each thread starts to read and execute all received messages at the thread's socket. Each thread does not proceed to the next synchronization barrier until all messages are consumed. The last phase is the reply processing, where each thread determines which entities are of interest to each client and sends out information only for those, i.e. notifies a client only about changes in entities that are visible to him.

3.2 Synchronization

Threads must synchronize the access to shared data structures between the synchronization barriers. During the world processing stage – which is sequential – and the reply processing stage there is no need to do any synchronization. In the last stage, the thread reads from shared data structures, but only writes in thread local buffers and sockets. However, during the request processing, the execution of the event from one client can alter the state of another entity, possibly being simulated by another thread.

The synchronization mechanism utilized by Abdelkhalek uses the semantics of the data structure known as Areanode tree. The Areanode tree is a binary-tree that allows the server to quickly find the list of entities that a single entity can interact using the location of this particular entity.

A node in an Areanode tree is defined by a tuple (3D region, plane). This plane is perpendicular to one of the three axes and divides the 3D region in two subregions. Each subregion generated by this division is used to define each one of the two tuples that will be children of this node. The 3D region of the root is defined as the entire virtual game map. The planes are calculated when server starts and are based on the Binary Space Partitioning (BSP) [5] representation of the game map. The maximum depth of this tree is hard-coded to be four.

Using the Areanode tree, the synchronization implements a region-based locking scheme. For each message consumed by a thread, the server finds the client that sent the message, locates the leaf in the Areanode tree that contains this client (based in its position) and acquires a lock in each node from leaf to root until the region determined by the current node contains all the entities that can be modified by the execution of this message. This method guarantees that two different threads will not change the state of the same entity nor the global data structures of the server in a undesirable manner.

3.3 Analysis

The performance analysis presented in [2] showed that in up to 70% of the execution time of the server, at least one thread is waiting at a synchronization point: 30% in lock contention and 40% in synchronization barriers.

Using the semantics of the game objects, Abdelkhalek did some performance optimizations in the mechanism used to acquire locks and improved the time spent with lock contention from 30% to 20%. We believe that there are two major problems with this approach:

- **lack of a dynamic load balancing method.** The static allocation of clients in threads at connection time may create a large load unbalance and may increase the time spent by the threads at synchronization barriers;
- **misuse of the semantics of game objects.** Locking by node in the Areanode creates a coarse grained mechanism that penalizes performance. Also, using the proposed lock mechanism makes the lock acquire operation time-consuming, since the Areanode must be traversed from leaf to root.

4 Parallelization Methodology

We propose a new parallelization model for the Quake server based in the Bulk Synchronous Parallel model [6] implemented using an event-driven architecture [7] with only one synchronization barrier.

4.1 Motivation

A preliminary analysis indicated that the original Quake server has low memory footprint and indicated also that only 10% of the time is spent with network-related system calls. This means that all the remaining time is spent on the execution of the server frames. According to the classification proposed by Pai et al. in [7], the original Quake server is implemented as a single-process event-driven (SPED) server. The server uses non-blocking system calls to perform asynchronous I/O operations. The server is able to overlap CPU and networking operations, achieving an efficient utilization of computing resources.

The nature of Quake simulation and the fact that it is written as an event-driven program makes Quake a good candidate for parallelism. The good results achieved by Zeldovitch et al. with his libasync-smp [8] showed that it is possible to have good performance improvements using coarse-grained parallelism in event-driven programs that have natural opportunities for parallel speedup.

Abdelkhalek's work with Quake, described in Section 3, introduced a multi-threaded version for Quake server that uses a static distribution of work between the available processors. Lack of load balancing and a locking strategy that misuse game objects semantics leaded to suboptimal speedups.

Using the ideas proposed by Zeldovitch and inspired by Abdelkhalek's work we created a multi-process event-driven system that is composed by multiple SPED process, as shown in next section.

4.2 Methodology

The nature of the simulations done by the Quake server determines which events can be executed concurrently. In order to keep the correct game semantics, all events originated in the same client must be executed sequentially and in the same order as in the original server. However, if the simulation of one entity does not interfere on the simulation of another entity, then the execution of both entities' events can be multiplexed by the server without any side effect to the simulation.

Our model decomposes the frame execution in four phases: world processing phase; task scheduling; parallel request and reply processing; synchronization barrier. Phases 1 and 2 runs in a coordinator process and does not run in parallel. The third phase is executed in various processors and the last phase is a synchronization barrier that waits the completion of each process frame. Each server frame is a super-step in the BSP model where phases 1 and 2 compound the input phase, local computation occurs in phase 3 and global communication and a synchronization barrier (output-phase) takes place in phase 4.

World Processing. The world processing phase is the first phase of this model and runs sequentially on the coordinator process. There is an inversion of control in this point of the server execution. The server runs the interpreter for the script programming language called QuakeC used by *id Software* to implement the game semantics. Each entity in the game has its own QuakeC functions associated to them and these functions implements the game semantics for actions and physical simulation for this entity.

Task Scheduling. Our task scheduling uses the entity position update mechanism employed by Quake server.

During the request processing, for every client α that sent an event during the current frame, a list L_α with all entities whose distance from α is not bigger than 256 pixels (α's action range) is computed by the Quake server. This list is used by the server to predict what entities (L_α) can be affected by the event sent by the client (α) being processed.

The game semantics of this list guarantees that if we use the same processor to simulate the events from client α and from all entities in L_α, then we do not need to do any kind of communication between the processors during the request and reply processing. We just need to schedule the simulation of all entities in the same list to the same processor. Our scheduling and load balancing algorithm works as follows.

First, we compute the set of connected components of the undirected graph $G = (V, E)$, where V is the list of all active entities in the current frame and $E = \{(\alpha, \beta) \mid \alpha \in V \land \beta \in L_\alpha\}$, i.e. an edge in (α, β) in E exists if the execution of events sent by α can potentially affect the entity β. It is important to note that the original quake code must compute the L_α for all active α clients. We use the already created lists to compute the connected components set using a union-find with path compression algorithm. The overhead to compute this connected components is not significative.

Then, we use the Longest Processing Time First (LPT) [9] heuristic to distribute the jobs among the available processors. Each job is defined as the task of simulating all the events related to the entities in one connected component and the weight of the job is the number of vertices in this connected component. In the context of the Quake simulation, the load produced by one active client is proportional to the number of entities that are near that client. Using the total number of entities in the same connected component as the weight of the job produced good load balancing results, as showed in Section 5.

The use of LPT allowed a lightweight implementation and guarantee good results with many simultaneous clients [9]. Note, however, that this definition of job weight can lead to load unbalance in cases where many entities are concentrated in one spot of the map (a special region of interest in the game map, for instance). We analyzed the influence of the scenario in load balancing in Section 5.2. After the scheduling, each processor starts the execution of request and reply processing.

Request and Reply Processing. The request and reply processing is the most computing intensive phase during the execution of a server frame. It comprehends reading the messages from the network, executing all events sent during the previous frame and computing and sending the replies to the clients. In the sequential version, this phase represents 80% of the server execution time.

Using the ideas proposed in libasync-smp [8], we create one worker process for each available CPU via the `fork()` system call. Each worker process runs as an independent SPED program that executes only the entities in the connected components assigned by the scheduler algorithm to this CPU.

The performance overhead caused by the creation (with `fork()`) of the worker processes is reduced because of the low memory footprint of each independent SPED process and mainly because of the copy-on-write (COW) [10] semantics of `fork()` implemented by the Linux Kernel. The impact of the use of `fork()` will be analyzed in Section 5.

Other implementation detail is related to the operating system scheduler. Using the Linux Trace Toolkit [11], a kernel tracer that generates traces of the execution of programs in an instrumented Linux kernel, we found that the default Linux scheduler does not work well with the workload generated by our parallel Quake server. The worker threads were sometimes scheduled to run in the same processor used by the coordinator process.

Using the new `sched_setaffinity()` and `sched_getaffinity()`, introduced in the Linux kernel 2.6, we could implement a more strict behavior in the kernel scheduler, that forced every new process to start in an idle processor. The performance improvements are shown in Section 5.

Inter-process synchronization. After sending the replies to the clients, the server starts the global communication phase. Each worker thread writes the updated entities state in a pre-allocated shared memory region and the coordinator reads and updates the new state.

Global data structures that depend on entity states, like the Areanode tree, are updated automatically during the next frame, at the world processing phase, so there is no need to update these structures at this point.

Instead of using the usual `wait()` system call to wait for the termination of the worker process, we use an alternative Linux mechanism that allows the coordinator process to proceed while the operating system destroys the auxiliary processes. The world processing phase – which is sequential – runs in parallel with the destruction of the worker process. In order to do that, we used the `clone()` system call without the option `SIGCHLD`. This means that the new child process is independent from the parent process and does not need to wait for a call to `wait()` to be destroyed. In the next section we present the performance analysis of our new parallel server.

5 Experimental Results

In order to evaluate the workload using several clients running the game simultaneously, we implemented[2] an automated Quake player in the Quake client code. Every ten client frames, the player starts to walk. After five client frames, the player chooses randomly one action event to send to the server. The action events available to the automated player are *jump*, a *gun shot* and *change direction* (that changes the direction of the movement being executed). After five more client frames, the client stops, and the procedure is repeated. We believe that this simple scheme are well suited for our analysis purposes. The event send rate is very close to the event send rate of an actual human player.

All results presented in this section are measured during the execution of game sessions with 160 simultaneous clients and all experiments were repeated at least thirty times.

5.1 Environment

The experimental environment that has been used to run the clients is made up of five AMD Athlon™ XP 2800+ (2.25 GHz), 1.0 GB of RAM and connected through a 100 Mbps Ethernet network. We utilized the Linux kernel version 2.6.16.2.

[2] Client and server source code can be downloaded from: `http://grenoble.ime.usp.br/~danielc/quake.tar.bz2` under GPL license.

The server was tested in two different systems. One has two Intel Xeon 3.0 GHz processors and 2.0 GB of RAM running Linux kernel version 2.6.17.7 with the patches required by the Linux Trace Toolkit version 0.5.76. The other has 8 UltraSPARC 900 MHz processors and 32.0 GB of RAM running Solaris 9.

5.2 Load Balancing

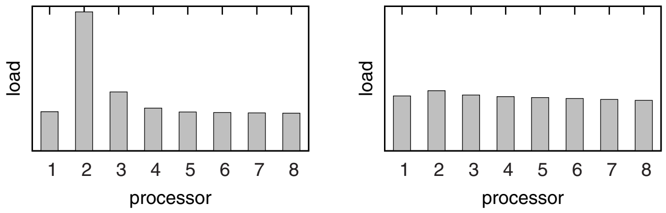
In order to analyze the results achieved by our load balance algorithm, we must understand the influence of the scenario in the server performance.

Scenario Influence. The scenario influence in our implementation showed up as an important issue since earlier tests. There are three issues related to the scenario that impacts our parallelization scheme:

- **quantity of map entities:** Quake maps have special entities that implements some functionalities related to map elements like elevators;
- **size of reply messages:** if the map defines more small rooms, the size of the reply message tend to be small, but the reply processing tends to be more processor intensive. If the map defines big rooms, the size of messages tends to increase;
- **concentration areas:** some areas of the map can have more importance to the game semantics and tend to concentrate a large number of players.

Small rooms and concentration areas can induce the creation of big connected components in our load balancing algorithm. We used two different scenarios in our analysis. A regular map created to games with 32 simultaneous users distributed by *id Software* called *death32c*[3] and one created by us, that are free of concentration areas.

Job size. In a 160 clients session, using the map *death32c* our load balance algorithm creates in average nine jobs with weight equals to eight in a single frame. With our own created map, without concentration areas, there are twenty jobs with weight equals to four in average. The normalized distribution of this jobs among the processors are shown in Fig. 1.



(a) Load balance in *death32c* (b) Load balance in our map

Fig. 1. Load balance among processors in the Solaris environment

[3] Available at `ftp://ftp.idsoftware.com/idstuff/quakeworld/maps/`.

Performance Results. Section 4.2 describes some implementation details that allowed us to control the operating system scheduling and how we parallelized the destruction of worker process.

Table 1. Performance of implementations evaluated in Linux environment

Evaluated implementation	frames/s	messages/frame	messages/s
using `fork()` and `wait()`	95	33	3.135
using `clone()` and `wait()`	198	18	3.564
using `clone()` without `wait()`	350	11	3.850

By analyzing the traces, we discovered that when the server uses `fork()` and `wait()` primitives just one of the available processors is used almost all the time. The results presented in Table 1 show that using `clone()` and controlling the operating system processor allocation, we increased the frame rate from 95 to 198 frames/s. Parallelizing the destruction of the worker process improved the frame rate from 198 to 350 frames/s.

Preliminary results using a computer with four processors showed that we achieved performance improvements using up to three processors. With more than three the server frame rate starts to drop.

Workload Analysis. The analysis of the execution trace generated by the Linux Trace Toolkit brought some interesting results.

The creation of new worker processes in each server frame represents 2.12% of the execution time in *death32c* and 1.49% in our map. The copy of the memory pages through the Linux copy-on-write mechanism corresponds to 4.69% of the execution time in *death32c* and 5.05% in our map.

We can classify the execution in four different moments according to the server parallelism. During the first moment, the frame starts and the world processing phase runs in parallel with the destruction of the worker process from the last frame. This corresponds to 17,93% of the execution time (with standard deviation $\sigma = 0.51\%$).

During a second moment, the server finishes the world processing phase and then the server computes the scheduling to the next server phase. The server uses only one processor. This takes 21.66% ($\sigma = 0.67\%$) of the total time.

The third moment corresponds to request and reply processing. This two phases correspond to 80% of the total time in the sequential version. In our parallel version, it corresponds to 37.69% ($\sigma = 0.42\%$) of the execution time.

In the last moment, the coordinator waits for the last worker process to finish its jobs and executes the global communication phase. It takes 22.70% ($\sigma = 22.70\%$) of the total execution time. This means that in the *death32c* map we utilize all available processors' processing power during 55.65% of the time.

The results obtained with our map are similar. The first moment takes 19.13% ($\sigma = 0.83\%$) of total time, the second takes 22.77% ($\sigma = 0.97\%$) and the third 34.43% ($\sigma = 0.63\%$) of the total time. The last moment uses 23.65% ($\sigma = 0.97\%$). We use all available processor' processing power in 53.65% of the total time.

6 Conclusion

In this work we investigate the parallelization and a dynamic load balance strategy to interactive, multiplayer game servers. We used QuakeWorld to describe the implementation issues of the proposed methodology in a real world system. Quake is an important representative of this class of applications.

Previous works from other researchers presented a multithreaded architecture with static distribution of work among available processors. This work distribution and the shared memory synchronization scheme did not take advantage from the semantics of the objects being simulated by the game. Their implementation obtained full parallelism only in 40% of the time, mainly because lock contention and load unbalance.

We present a new dynamic load balancing and scheduling mechanism that utilizes the semantics of the simulation executed by the server. The division of the tasks allowed the creation of a Bulk Synchronous Parallel model for QuakeWorld, in which the execution of the computational phase is lockless.

Our experiments analyze the impact of certain external parameters of the simulation such as the virtual map utilized. With kernel instrumentation we measured a total parallelization time improvement from around 40% from previous work to 55% in our implementation.

References

1. Abdelkhalek, A., Bilas, A., Moshovos, A.: Behavior and performance of interactive multi-player game servers. In: Proceedings of the International IEEE Symposium on the Performance Analysis of Systems and Software (ISPASS-2001), Arizona, USA, November 2001, IEEE Computer Society Press, Los Alamitos (2001)
2. Abdelkhalek, A., Bilas, A., Moshovos, A.: Behavior and performance of interactive multi-player game servers. Cluster Computing 6(4), 355–366 (2003)
3. Abdelkhalek, A., Bilas, A.: Parallelization and performance of interactive multi-player game servers. In: Proceedings of 18th International Parallel and Distributed Processing Symposium, apr 2004, p. 72a. IEEE Computer Society Press, Los Alamitos (2004)
4. id Software homepage (2006), `http://www.idsoftware.com`
5. Shimer, C.: Binary space partition trees (1997) Available at `http://www.cs.wpi.edu/~matt/courses/cs563/talks/bsp/bsp.html`
6. Valiant, L.: A bridging model for parallel computation. Communications of the ACM 33(8), 103–111 (1990)
7. Pai, V.S., Druschel, P., Zwaenepoel, W.: Flash: An efficient and portable Web server. In: Proceedings of the USENIX 1999 Annual Technical Conference, California, EUA, June 1999, pp. 199–212 (1999),

8. Zeldovich, N., Yip, A., Dabek, F., Morris, R., Mazieres, D., Kaashoek, F.: Multiprocessor support for event-driven programs. In: Proceedings of the 2003 USENIX Annual Technical Conference, June 2003, pp. 239–252 (2003)
9. E.C. Jr., Sethi, R.: A generalized bound on LPT sequencing. In: SIGMETRICS '76: Proceedings of the 1976 ACM SIGMETRICS conference on Computer performance modeling measurement and evaluation, pp. 306–310. ACM Press, New York (1976)
10. Bovet, D.P., Cesati, M.: Understanding the Linux Kernel. 3rd edn. O'Reilly Media (November 2005)
11. Yaghmour, K., Dagenais, M.R.: Measuring and characterizing system behavior using kernel-level event logging. In: Proceedings of the 2000 USENIX Annual Technical Conference, Berkeley, CA, June 2000, pp. 13–26. USENIX Association (2000)