Estudo de escalabilidade de servidores baseados em eventos em sistemas multiprocessados: um estudo de caso completo

Daniel de Angelis Cordeiro
Universidade de São Paulo, Departamento de Ciência da Computação,
São Paulo, Brasil, 05508-090
danielc@ime.usp.br

Abstract

The explosive growth in the number of Internet users made software architects reevaluate issues related to the scalability of services deployed on a large scale. It is still challenging to design software architectures that do not experience performance degradation when the concurrent access increases.

In this work, we investigate the impact of the operating system in issues related to performance, parallelization, and scalability of interactive multiplayer games. Particularly, we study and extend the interactive, multiplayer 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 that model in QuakeWorld server with adaptations that allows the operating system to manage the execution of the generated workload in a more convenient way.

Keywords: Interactive Multiplayer Games; Parallelization; Load Balancing

Resumo

O crescimento explosivo no número de usuários de Internet levou arquitetos de software a reavaliarem questões relacionadas à escalabilidade de serviços que são disponibilizados em larga escala. Projetar arquiteturas de software que não apresentem degradação no desempenho com o aumento no número de acessos concorrentes ainda é um desafio.

Neste trabalho, investigamos o impacto do sistema operacional em questões relacionadas ao desempenho, paralelização e escalabilidade de jogos interativos multi-usuários. Em particular, estudamos e estendemos o jogo interativo, multi-usuário, QuakeWorld, disponibilizado publicamente pela id Software sob a licença GPL. Criamos um modelo de paralelismo para a simulação distribuída realizada pelo jogo e o implementamos no servidor do QuakeWorld com adaptações que permitem que o sistema operacional gerencie de forma adequada a execução da carga de trabalho gerada.

Palavras-chaves: Jogos interativos, multi-usuários; Paralelização; Balanceamento de Carga

1 Introdução

A tecnologia de redes de computadores atual possibilita a troca de informações entre milhões de usuários através da Internet. Tal ambiente rapidamente despertou o interesse de diversos prestadores de serviços, que perceberam que a Internet é a maneira mais eficaz atender seus usuários, a custos relativamente baixos.

Redes velozes e confiáveis não são o único pré-requisito para a criação de serviços que consigam atender tal demanda. É necessário que as aplicações disponibilizadas sejam capazes de atender a esse grande número de usuários sem apresentarem degradação no desempenho.

O entendimento da escalabilidade de tais serviços é fundamental para que os arquitetos de *software* consigam disponibilizá-los para um grande número de usuários. Ao mesmo tempo em que a escalabilidade de aplicações científicas foi exaustivamente estudada, pouco se conhece sobre a escalabilidade de aplicações comerciais.

Um dos paradigmas bastantes utilizados em sistemas distribuídos escaláveis é o paradigma de programação *event-driven* (baseado em eventos), onde o fluxo de execução do programa é determinado por eventos e o processamento corresponde a respostas a estes eventos. Tipicamente, o aplicativo informa ao despachante de eventos quais os eventos em que está interessado e o despachante notifica o programa sempre que o evento ocorre.

Neste trabalho, investigamos o impacto do sistema operacional em questões relacionadas ao desempenho, paralelização e escalabilidade de jogos interativos multi-usuários. Em particular, estudamos e estendemos o jogo interativo, multi-usuário, QuakeWorld [12], disponibilizado publicamente pela id Software sob a licença GPL. Criamos um modelo de paralelismo para a simulação distribuída realizada pelo jogo e o implementamos no servidor do QuakeWorld, com adaptações que permitem que o sistema operacional gerencie de forma adequada a execução da carga de trabalho gerada.

2 O servidor do quake

Realizamos um estudo minucioso do modelo utilizado pela id Software no código-fonte do jogo QuakeWorld [12] para que fosse possível caracterizar e propor melhorias ao seu funcionamento.

Neste capítulo descreveremos a arquitetura e modo de funcionamento da versão original do QuakeWorld, resultados de nossa análise de desempenho e caracterização do uso dos recursos do sistema operacional pela versão original. Apresentaremos, também, a proposta de paralelização e resultados do trabalho de Abdelkhalek et al [2–4].

2.1 Estudo do modelo utilizado no código-fonte do Quake

O Quake é um jogo de ação interativo, multi-usuário, desenvolvido e distribuído pela id Software. Seu lançamento, em 31 de maio de 1996, foi considerado um marco na história dos jogos pois introduziu grandes avanços nos jogos de computador como, por exemplo, a utilização de modelos tri-dimensionais para representação das entidades que compõem o jogo. Até então utilizavam-se modelos bi-dimensionais e o desenho era feito utilizando projeções ortogonais. Em dezembro do mesmo ano foi lançada uma atualização com melhorias no modo de jogo multi-usuário, que proporcionou melhor desempenho em jogos via Internet. Essa nova versão foi denominada QuakeWorld. O restante do texto utiliza os termos Quake e QuakeWorld indiscriminadamente para referenciar essa nova versão.

O processamento no servidor é dividido em quadros (*frames*). Em cada quadro, o servidor evolui o modelo de simulação, processa os dados enviados pelos clientes e responde aos clientes com os dados relevantes.

O primeiro passo realizado em um quadro é verificar se há clientes que perderam a comunicação com o servidor. O modelo utilizado pelo servidor pode ser dividido em duas partes: *modelo físico* e *modelo de jogo*.

O modelo físico descreve e simula a mecânica física das entidades autônomas do jogo. O código do servidor que simula essas características físicas é conhecido no meio de desenvolvimento de jogos como *game engine*. Utilizando o modelo físico, o jogo determina, a cada quadro, qual a nova velocidade, aceleração, direção e posição absoluta do jogador levando em conta se ele está andando em uma superfície plana, se está no meio de um salto (e, portanto, deve-se considerar o efeito da gravidade) ou se está na água (que além da gravidade que puxa o jogador para o fundo do lago, deve-se considerar o atrito provocado pela água). O *game engine* é considerado parte vital de um jogo pois, junto com os gráficos do cliente, é ele quem dá a sensação de realidade do jogo.

O modelo de jogo descreve como os elementos do mundo virtual se comportam no jogo. O modelo de jogo é descrito em uma linguagem desenvolvida pela id Software denominada *QuakeC*. O código nessa linguagem é compilado e tem seu `bytecode` interpretado pelo servidor no momento da simulação do modelo.

O terceiro passo executado pelo servidor durante a execução de um quadro é processar as mensagens enviadas pelos clientes. Há várias mensagens relacionadas ao controle da sessão do jogo mas, tipicamente, o cliente enviará uma mensagem do tipo `MOVE`. Uma mensagem do tipo `MOVE` informa o servidor que o jogador executou um movimento. Essa é a ação mais importante e que tem mais conseqüências para a simulação do jogo. Ao receber uma mensagem `MOVE`, o servidor evolui o modelo físico da entidade que representa o personagem do jogador e determina quais outras entidades da simulação *potencialmente* serão afetadas por esse movimento.

O servidor mantém uma estrutura de dados denominada árvore *Areanode* que permite que o servidor localize rapidamente quais entidades estão próximas a um determinado jogador. Cada nó da árvore representa as entidades presentes em uma região do mapa.

No final do quadro, o servidor constrói e envia as mensagens com as atualizações do estado da simulação apenas para os clientes que enviaram mensagens durante o quadro, ou seja, o servidor assume que os clientes ativos estão freqüentemente enviando novos dados para o servidor.

A construção dessas mensagens, entretanto, não é trivial. Para que as mensagens consumam pouca banda o servidor envia apenas os dados das entidades que foram alteradas durante o quadro e que estão na possível área de alcance do jogador (*Possible Visible Set* ou *(PVN)*). O cálculo da possível área de alcance do usuário requer o uso intensivo dos recursos de processamento, mas é realizado de forma eficiente graças ao uso de uma *Binary Space Tree* [19].

2.2 Versão *multithreaded* do Quake

Ahmed Abdelkhalek et al. iniciaram um esforço para caracterização dos requisitos computacionais e paralelização do servidor do QuakeWorld. O trabalho de Abdelkhalek pode ser dividido em duas fases. A primeira fase consiste na caracterização dos requisitos computacionais e análise de desempenho do servidor do Quake [3, 4]. O resultado de suas experimentações mostraram que o aumento do número de usuários simultâneos produzia um aumento linear no consumo de banda de rede, mas super-linear no tempo de processamento. Uma rede Ethernet de 100 Mbits/s poderia acomodar mais de 1.000 clientes simultâneos.

Foi então que deram início a segunda fase do trabalho [2], onde apresentam seus esforços na paralelização e implementação de uma versão multithreaded do servidor do Quake.

2.2.1 Proposta de paralelização

Sua proposta de paralelização consiste de uma versão *multithreaded* do servidor do Quake com atribuição estática de tarefas.Todo o processamento realizado para a simulação de um mesmo cliente é feito sempre pela mesma *thread*. Os autores afirmam que é possível aplicar políticas dinâmicas de atribuição de clientes a *threads* e sugerem que uma política que levasse em conta a localização dos jogadores seria interessante. Essa idéia foi tratada como um trabalho futuro, pois seriam necessárias alterações mais intrusivas no código do servidor. Nosso modelo de paralelismo, mostra que é possível implementar distribuição dinâmica de tarefas sem a necessidade de alterações tão intrusivas.

2.2.2 Paralelização de um quadro do servidor

A fim de manter a corretude da simulação realizada pelo servidor, dois invariantes foram impostos pelos autores: (i) cada fase do processamento do servidor é distinta e não deve se sobrepor às outras e (ii) cada fase deve ser executada em sua ordem original: simulação do modelo físico, processamento da requisição e processamento da mensagem de resposta.

No início da execução do servidor, todas as *threads* são criadas e ficam suspensas esperando novas requisições. A *thread* eleita como coordenadora de um quadro é responsável por executar todas as etapas seqüenciais do servidor. Em particular, é ela que realiza a etapa de simulação física do mundo assim que um novo quadro de simulação é iniciado. Terminada a simulação do modelo físico, a *thread* coordenadora notifica as demais *threads* para prosseguirem com a execução.

A próxima etapa é o processamento das mensagens enviadas. Durante essa etapa, cada *thread* entra em um laço responsável por consumir e processar todas as mensagens contidas em seu soquete de mensagens. A *thread* fica neste laço até que todas as mensagens tenham sido processadas.

A última fase é a fase de processamento de respostas. Cada *thread* computa e envia uma mensagem de resposta para todos os clientes que tiveram uma mensagem processada neste quadro de simulação. Assim que cada *thread* termina de enviar as mensagens uma chamada à `select()` é realizada e o quadro de simulação é terminado.

2.2.3 Sincronização

Dado que as fases de simulação são separadas por barreiras de sincronização, o servidor só precisa realizar sincronizações intra-fases para acessar estruturas de dados globais.

A etapa de criação e envio de respostas não apresenta nenhum conflito de leitura e escrita nas estruturas de dados compartilhadas e, por isso, não há necessidade de sincronização nesta etapa. A etapa de processamento das mensagens enviadas pelos clientes, entretanto, é mais complicada. A simulação dos comandos enviados

nas mensagens envolvem a atualização das características das entidades e deve realizar algum mecanismo de sincronização.

A sincronização das entidades do jogo consiste em proteger com um *lock* o nó da árvore *Arenode* que contém todas as entidades relevantes para a atualização da estrutura de dados. Antes de atualizar uma entidade, o servidor tenta colocar um *lock* na folha da árvore que contém o elemento. Todos os nós são percorridos até que seja encontrado um ancestral que contenha todos os objetos que potencialmente serão afetados pela atualização. Assim que encontrado, esse ancestral também é protegido com um *lock*.

Os resultados dos experimentos mostraram que o maior gargalo da Versão-Multithreaded proposta por Abdelkhalek é justamente o tempo gasto com a sincronização. Ao utilizar oito *threads*, o tempo gasto com sincronização chega à 70% do tempo total. 30% do tempo total foi gasto com obtenção de *locks* e 40% com barreiras de sincronização global.

Abdelkhalek aprimorou a técnica e desenvolveu uma nova estratégia para a utilização de *locks*. A otimização consiste em utilizar a semântica dos objetos para a realização dos *locks*. Com isso, o tempo gasto com obtenção de *locks* foi reduzido de 30% para 20% do tempo total.

3 Metodologia de paralelização

Utilizamos o modelo de programação BSP [22] aliado a uma extensão da arquitetura baseada em eventos para ambientes multiprocessados para obter um modelo de paralelização que necessita de apenas um ponto de sincronização entre os processadores. Neste modelo não é necessário, portanto, sincronizar o acesso à memória compartilhada durante a etapa paralela de processamento.

3.1 Motivação

3.1.1 Análise da versão multithreaded do Quake

A versão *multithreaded* do servidor do Quake implementada por Abdelkhalek [2] apresentou graves problemas de desempenho em seu estágio de desenvolvimento inicial. Tais problemas foram ocasionados, principalmente, por problemas de contenção de *locks*.

Em testes realizados pelos autores em uma máquina SMP com quatro processadores (onde todos tinham a funcionalidade *Hyper-Threading* ativada) a versão *multithreaded* com 8 *threads* gastava 35% do tempo com a sobrecarga devida à contenção de *locks*. Em 40% do tempo, as *threads* ficavam suspensas esperando que outras *threads* terminassem alguma tarefa antes de poderem continuar.

Nota-se que além da falta de investigação mais aprofundada sobre o problema de balanceamento de carga, houve uma falha na definição de uma política eficiente de utilização de *threads*. O artigo analisa o desempenho do servidor quando é utilizada uma *thread* por cliente, por exemplo, o que vai contra os resultados mais recentes da área de escalabilidade de sistemas.

Além disso, o problema de contenção de *locks* causou um grande impacto no desempenho da versão *multithreaded* do servidor. A análise do problema fez com que os autores percebessem que utilizar a semântica das entidades do jogo para a obtenção dos *locks* possuía um grande potencial para melhorar esses problemas.

Por estas razões, acreditamos que uma proposta para um novo modelo de paralelismo para o servidor do Quake deve: (i) levar em conta a semântica da simulação para distribuição da carga; (ii) considerar o problema de contenção de *locks* e (iii) fazer um mapeamento mais preciso entre a abstração de múltiplos processos utilizada (*threads* ou outros processos criados via `fork()`) e os processadores disponíveis.

3.2 Metodologia

3.2.1 Visão geral

A natureza da simulação executada pelo servidor do Quake define quais eventos podem ser tratados de forma concorrente e quais devem ser tratados seqüencialmente. Em geral, para executar a simulação de uma mesma entidade do jogo, a ordem de tratamento dos eventos deve ser seqüencial e deve respeitar a ordem previamente definida pela versão seqüencial do servidor do Quake. Entretanto, se a simulação de uma entidade não interferir na simulação de outra entidade, então a execução dos eventos das mesmas podem ser multiplexadas pelo servidor sem alteração nos efeitos da simulação do servidor.

Tendo isso como base, a versão paralela do servidor – denominada doravante como Versão-Paralela foi dividida em quatro fases:

1. simulação do modelo físico e tarefas administrativas;
2. distribuição dinâmica de tarefas;
3. tratamento paralelo da fila de eventos do servidor;
4. sincronização global entre os processadores.

O modelo de paralelização desenvolvido para o servidor do Quake segue o modelo *Bulk Synchronous Parallel* (BSP) [22] de computação paralela. Cada quadro de simulação é um super-passo, onde as fases 1 e 2 formam a *input phase* do modelo BSP. A computação local é realizada durante a terceira fase. Por fim, a *output phase* é realizada pela quarta fase da versão paralela.

3.2.2 Simulação do modelo físico

A primeira fase, simulação do modelo físico e distribuição de tarefas, é realizada por um processo coordenador e é executada em uma fase em que não há paralelismo no servidor. Nesta etapa seqüencial da execução do servidor do Quake também são realizadas algumas tarefas administrativas para manutenção da sessão do jogo. Dentre as tarefas, podemos destacar:

- detecção e remoção automática de clientes inativos;
- manutenção dos *logs* gerados pela simulação;
- execução de comandos de administração digitados no console de administração do servidor (envio de mensagens aos jogadores, término de uma sessão, alteração do mapa utilizado no jogo, etc.).

3.2.3 Distribuição dinâmica de tarefas

Criamos uma estratégia para realizar a distribuição dinâmica de tarefas que utiliza a posição de todas as entidades relevantes ao processamento de um dado quadro do servidor como critério para o balanceamento de carga entre as CPUs disponíveis.

A estratégia consiste em detectar em cada quadro de simulação do jogo todos os grupos formados por entidades que estão próximas entre si e que, portanto, possivelmente podem interagir entre si. São considerados apenas os elementos relevantes à este quadro de simulação. Ou seja, serão considerados para inclusão nesses grupos apenas os elementos que realizarem alguma ação e aqueles que possivelmente serão afetados por alguma ação neste quadro de simulação.

Dois elementos α e β estão próximos entre si se α está na área de atuação de β. Geometricamente, a área de atuação de uma entidade é definida pelo servidor do Quake como sendo o espaço delimitado por um cubo centrado nas coordenadas da entidade com aresta de tamanho 256 unidades de distância. Se as coordenadas de β estão contidas dentro do espaço delimitado pelo cubo centrado nas coordenadas de α, então α e β estão próximos entre si. Note que o conceito de proximidade aqui definido desconsidera elementos definidos pelo cenário, como paredes e portas.

Definimos um grupo de entidades próximas como sendo o conjunto que contém os elementos representados pelos vértices de cada componente conexo do grafo $G = (V, E)$, onde cada elemento de V representa uma entidade relevante ao quadro e cada aresta (α, β) em E indica que as entidades α e β estão próximas entre si.

Uma propriedade interessante que decorre da lógica de simulação do servidor é que se a execução de uma ação de uma entidade α produz uma alteração no estado de outra entidade β, então α e β estão em um mesmo grupo de entidades próximas.

Dessa forma, foi possível criar um método de paralelização onde não é necessário nenhum tipo de sincronização para o acesso de leitura e de escrita aos atributos das entidades. Basta que todos os elementos de um mesmo grupo sejam simulados por um mesmo processador. Apenas as modificações às estruturas de dados globais, como, à árvore *Areanode*, por exemplo, devem ser sincronizadas.

O balanceamento de carga entre os processadores é realizado utilizando-se o algoritmo de escalonamento de tarefas denominado *Longest Processing Time First* (LPT) [10, 13].

No contexto do servidor do Quake, quanto mais entidades estiverem próximas a um jogador, mais custoso será o tratamento das mensagens enviadas por ele e maior será a quantidade de informações que serão

enviadas nas mensagens de resposta. Por esse motivo, definimos como tarefa a simulação de todos os eventos de um mesmo grupo e definimos que o tamanho de uma tarefa é o número de elementos de cada grupo.

Divididas as tarefas entre os processadores, o próximo passo da versão paralela do servidor do Quake é o tratamento da fila de eventos do servidor.

3.2.4 Tratamento paralelo de eventos

O tratamento dos eventos é a fase do servidor que é efetivamente paralela. Compreende tanto o tratamento das novas mensagens enviadas ao servidor, quanto a criação e envio das mensagens de resposta para os clientes que estavam ativos nesse quadro, etapas que correspondiam a cerca de 80% do tempo utilizado pela CPU na versão do servidor modificada para 160 clientes.

Após distribuir os grupos de entidades próximas utilizando o algoritmo proposto em 3.2.3, o processo pai ativa todos os processos auxiliares – um processo por CPU disponível – e estes começam a simulação de seus respectivos clientes.

É interessante notar que cada processo auxiliar age como se fosse uma instância independente do servidor do Quake, onde os únicos clientes ativos no atual quadro de simulação são os clientes contidos nos grupos que foram atribuídos à este processo. Portanto, cada processo auxiliar funciona como um servidor *single-process event-driven* independente.

Dessa forma, não há necessidade de realizarmos nenhuma sincronização entre os processos durante toda essa etapa do servidor. Com isso, conseguimos evitar o problema de contenção de *locks* apresentado na implementação de Abdelkhalek. Por outro lado, nesta abordagem há a necessidade de se transmitir as modificações para o processo coordenador. A sincronização desses dados é a próxima fase da execução do servidor.

3.2.5 Sincronização inter-processos

Após o término do tratamento de eventos, cada processo auxiliar inicia uma fase de troca de informações com o processo coordenador para que este tenha a chance de absorver as mudanças ocorridas com cada cliente e de atualizar o estado global da simulação.

O processo coordenador precisa obter dos processos auxiliares as novas informações sobre as variáveis específicas da simulação do jogo, sua nova posição e, então, recalcular a nova posição do jogador no mapa virtual e atualizar a árvore *Areanode* para que esta reflita essa nova posição.

Assim que todos os processos auxiliares terminam de trocar informações com o coordenador, este super-passo da execução paralela é terminado e o processo coordenador pode prosseguir para a fase seqüencial da execução e iniciar um novo quadro de simulação do servidor.

3.3 Detalhes de implementação

Nesta seção apresentamos os detalhes de implementação do modelo de paralelização apresentado na Seção 3.2.

A implementação deu-se através de modificações no código original do servidor, disponível publicamente no sítio do fabricante sob a licença *GNU General Public License* (GPL) versão 2. Disponibilizamos o código-fonte da versão paralela do servidor no endereço `http://grenoble.ime.usp.br/~gold/orientados/`.

3.3.1 Processos auxiliares e comunicação inter-processos

No protótipo de implementação do modelo de paralelização proposto por este trabalho, os processos auxiliares são processos do sistema operacional criados através da chamada de sistema `fork()`. De acordo com a classificação sugerida por Flynn [9], o protótipo segue o modelo de programação paralela conhecido como *Single Program, multiple data* (SPMD).

Após o cálculo dos grupos de entidades próximas formados pelas entidades ativas de um quadro de simulação, o servidor determina quantos processos auxiliares serão necessários para o processamento dos grupos. Será utilizado um número de processos auxiliares menor ou igual ao número de processadores disponíveis no computador menos um, uma vez que o processo coordenador também executa uma parte da simulação.

A criação de apenas um processo auxiliar por processador se deve ao fato do servidor do Quake utilizar a arquitetura *single-process event-driven* [16] em sua plenitude. O servidor não se utiliza de nenhuma chamada de sistema que possa bloquear o processo uma vez que, tais chamadas poderiam levar à má utilização da

CPU devido a problemas inerentes à implementação destas no sistema operacional. Como observado em [24], essa metodologia de paralelização garante a utilização eficiente das CPUs disponíveis.

Outra decisão que deve ser discutida é a utilização de processos do sistema operacional ao invés de *threads* para a implementação dos processos auxiliares. Duas razões principais levaram a decisão de utilizarmos processos ao invés de *threads*.

A primeira razão é que a proposta de implementação com `fork()` torna a paralelização transparente para todo o mecanismo de processamento de eventos – desde a leitura da mensagem até o envio da resposta. Não há preocupação com *locks* nessa etapa e o problema de determinar quais variáveis possuem escopo do tratamento de eventos não existe. Entretanto, há a necessidade de integrar as modificações locais às estruturas de dados globais, como veremos na Seção 3.3.4.

A segunda questão sobre o uso de processos ao invés de *threads* foi a percepção de que após as modificações no código-fonte do Quake o esforço do sistema operacional para criar uma cópia do processo do servidor do Quake através do `fork()` seria relativamente pequeno.

A criação de *threads* no Linux segue os mesmos passos da criação de novos processos, através do uso da chamada de sistema `clone()`. Na verdade, para a criação, execução e escalonamento de tarefas no sistema operacional não há diferenciação entre *threads* e processos comuns. Ambos são representados por uma `task_struct` e criados através de uma chamada à `clone()`; a diferença é que certos recursos são compartilhados entre o processo criado como *thread* e o processo que o criou. Desses recursos compartilhados, talvez o mais importante seja o ponteiro para a tabela de memória virtual, que permite que a memória seja compartilhada entre os processos criados dessa forma, definindo, assim, a semântica de *thread* desses processos.

Quando o novo processo não possui semântica de *thread* e, portanto, não compartilha a memória virtual com o processo pai, é necessário criar uma cópia dos recursos não compartilhados para que as duas linhas de execução possam prosseguir independentemente. O Linux realiza essa cópia de recursos sob demanda, utilizando um mecanismo que é conhecido como *copy-on-write*.

O mecanismo de *copy-on-write* utilizado pelo sistema de memória virtual funciona da seguinte forma: quando o processo filho é criado, as páginas de memória do processo pai são marcadas como sendo somente para leitura e a estrutura `task_struct` do processo filho é configurada para apontar para as mesmas páginas do pai. Quando algum dos processos realizar uma operação de escrita em uma das páginas, ocorrerá uma falha de página e o processo fará uma cópia da página para si. Nesse momento, o núcleo do SO marca as duas páginas – a original e a nova cópia – como sendo de leitura e escrita. A partir daí, cada processo utiliza sua própria cópia da página de memória. Portanto, o sobrecusto de utilização do `fork()` está associado diretamente à quantidade de memória utilizada pelo processo.

No servidor do Quake, a maior parte da memória RAM é consumida na armazenagem das estruturas de dados que contêm as propriedades utilizadas para a simulação de cada entidade do jogo. Tais propriedades são atualizadas em duas situações: na etapa de simulação física do servidor e no processamento de mensagens.

Essas informações precisam ser repassadas aos processos auxiliares após a etapa de simulação física e precisam ser devolvidas para o processo coordenador após o processamento e envio das mensagens. É necessário, então, uma forma de comunicação inter-processos adequada para fazer isso.

Para minimizar a quantidade de páginas de memória copiadas pelo mecanismo de *copy-on-write* do `fork()`, escolhemos utilizar como forma de comunicação inter-processos memória compartilhada.

Dessa forma, o mecanismo de *copy-on-write* não precisa copiar as páginas de memória com as propriedades das entidades, uma vez que estas estão em uma área de memória compartilhada, e nem precisa copiar as páginas de memória que contêm apenas informações que são utilizadas exclusivamente para leitura. Apenas as páginas de memória que armazenam áreas de memória utilizadas no escopo do processamento e envio de mensagens – que não precisam ser sincronizadas com os outros processos auxiliares – é que serão copiadas.

O uso de um *pool* de processos diminuiria o custo da criação de processos, mas não o da cópia das páginas de memória, uma vez que todos os dados da simulação necessariamente precisariam ser copiados do processo coordenador para o filho ao término da fase de simulação física. Além disso, páginas de memória que não fossem utilizadas em um quadro ou que fossem utilizados apenas para leitura teriam que ser copiadas em todos os quadros de simulação e não somente quando fossem necessários, como ocorre com a utilização do mecanismo de *copy-on-write* do sistema operacional.

Essas observações são importantes, pois garantem que a implementação deste protótipo no próprio servidor do Quake é adequada e garante uma boa aproximação do que seria possível se a implementação utilizasse *threads*.

3.3.2 Distribuição dinâmica de tarefas

A implementação do mecanismo de distribuição dinâmica de tarefas descrito na Seção 3.2.3 exigiu uma modificação no modo como o servidor do Quake consome as mensagens do soquete de rede.

As mensagens disponíveis no soquete são lidas e armazenadas em uma fila de mensagens. O servidor lê, no máximo, um número de mensagens igual ao número de clientes com que o servidor pode lidar a fim de que o quadro de simulação não fique demasiadamente grande. Durante o processamento das mensagens, estas são lidas da fila de mensagens ao invés de serem lidas diretamente do soquete. Com o enfileiramento prévio das mensagens, determinar quais os clientes ativos neste quadro de simulação é trivial.

Para cada cliente desta fila, são realizados os seguintes passos:

- construção de um `pmove`;
- detecção das entidades na área de atuação do movimento;
- adição dos clientes e das entidades possivelmente afetadas ao grafo de entidades próximas.

Implementamos o algoritmo de balanceamento de carga denominado *Longest Processing Time First* (LPT) [10, 13]. O tamanho de cada cada componente conexa do grafo calculado foi escolhido como estimativa para o tempo de processamento de cada grupo, uma vez que quanto maior o grupo, possivelmente haverá mais interações entre as entidades e, por conseqüência, mais processamento.

Todos os processadores iniciam o processo com carga total igual a zero. Exceção é feita para o processo coordenador, que inicia com carga igual a um como medida para forçar uma melhor utilização dos processos auxiliares. Para cada grupo da lista ordenada, removemos aquele com o maior tamanho da lista e atribuímos ao processador com menor carga naquele momento. A nova carga do processador escolhido é igual ao valor da carga anterior, acrescido do tamanho do grupo atribuído.

Assim que todos os grupos são atribuídos a algum processador, o servidor pode prosseguir com a criação dos processos auxiliares e com o tratamento paralelo de eventos.

3.3.3 Tratamento paralelo de eventos

A etapa de tratamento paralelo de eventos compreende a leitura e processamento das mensagens enviadas pelos clientes. A natureza do modelo de paralelização proposto para o servidor do Quake permitiu que o resultado da implementação desta etapa fosse simples e praticamente transparente ao mecanismo de tratamento de eventos original do servidor do Quake.

Inicialmente, o processo coordenador cria os processos auxiliares através do comando `fork()`. Sendo n o número de processadores disponíveis e m o número de grupos de entidades próximas criados na etapa de distribuição dinâmica de tarefas, são criados $\min\{n-1, m\}$ processos auxiliares. Ou seja, são criados no máximo um processo auxiliar por processador disponível, mas não são criados mais processos do que grupos para serem processados. Além disso, o próprio processo coordenador executa o processamento de alguns grupos e, portanto, não há necessidade de criarmos mais do que $n-1$ processos para utilizar os n processadores disponíveis.

Cada processo auxiliar prossegue com sua execução independentemente. Nesse trecho, há poucas modificações no código-fonte do servidor.

3.3.4 Sincronização inter-processos

A sincronização inter-processos ocorre após a fase de tratamento paralelo de eventos e marca o fim do super-passo do modelo BSP de paralelização. Todos os processos auxiliares devem enviar o conjunto de modificações realizadas em suas entidades para o processo coordenador que, após receber as modificações de todos os auxiliares, prossegue para o próximo super-passo.

Como dito anteriormente, o mecanismo utilizado para a realização de comunicação inter-processos nesta implementação é o mecanismo de memória compartilhada definido pelo *System V*. Para máquinas do tipo SMP, memória compartilhada é o método mais eficiente de comunicação inter-processos disponível no sistema operacional Linux [17].

Ao fim do super-passo, cada processo auxiliar deve separar os dados referentes à simulação dos dados referentes à árvore *Areanode* e copiá-los para uma área de memória compartilhada. O processo coordenador, após esperar todos os processos auxiliares terminarem, copia esses dados (armazenados em regiões de memória

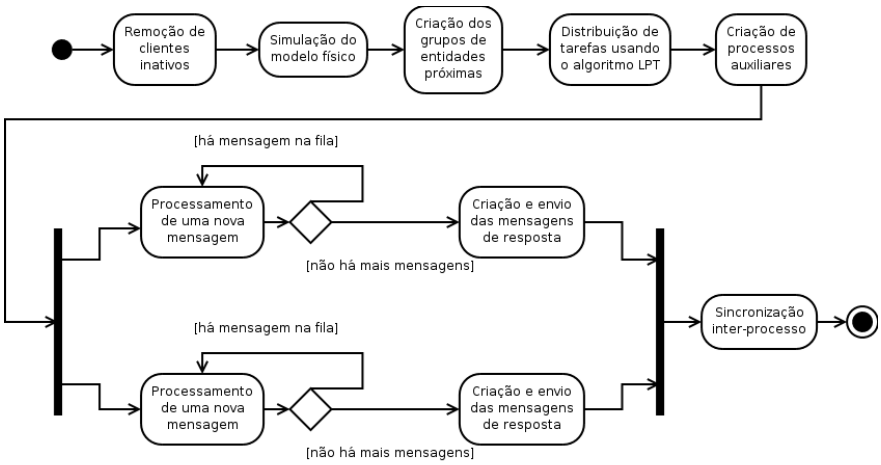


Figura 1: Diagrama de atividades da execução de um quadro de simulação da versão paralela do servidor do Quake

compartilhada) de volta às estruturas de dados locais. Note que no momento do `fork()` não é necessário tomar nenhum cuidado adicional para copiar essas informações para os processos auxiliares, já que estas são herdadas diretamente das páginas de memória copiadas do processo pai.

Por fim, os elementos modificados devem ser re-inseridos na árvore *Areanode* para que a árvore passe a refletir o novo posicionamento das entidades modificadas pelos processos auxiliares. Essa atualização ocorre naturalmente na etapa de atualização do modelo físico do servidor, já que os atributos das entidades são atualizados pela simulação e a re-inserção é realizada pelo código original do Quake.

A Figura 1 exibe o diagrama de atividades para a execução de um quadro de simulação em um computador com dois processadores disponíveis.

3.4 Efeitos do escalonador do SO na implementação

Os primeiros experimentos com a implementação proposta na Seção 3.3 não produziram bons resultados. Os experimentos – apresentados em 4.4.4 – mostravam que a paralelização praticamente não havia produzido efeito na taxa de quadros de simulação executados no servidor por segundo devido à otimizações realizadas pelo escalonador do sistema operacional.

A seção seguinte apresenta as adaptações na implementação que permitiram que o servidor apresentasse um paralelismo real na execução dos processos.

3.4.1 Adaptações

Implementamos uma nova metodologia de criação e destruição de processos que permitiu que a execução do servidor paralelo do Quake ocorresse de forma mais adequada. Com as modificações impostas por essa nova metodologia, foi possível obter um comportamento do escalonador mais adequado ao tipo de carga de trabalho exercido pela versão paralela do servidor.

Tais modificações, entretanto, necessitam que o sistema operacional utilizado disponibilize um mecanismo que forneça um controle fino, configurável em nível de usuário, para questões como criação, escalonamento e inter-dependência entre processos pai e filho. O sistema operacional Linux possui mecanismos desse tipo e, por isso, a versão adaptada tornou-se dependente desse sistema operacional.

3.4.2 Criação dos processos auxiliares

A metodologia de criação de processos foi modificada para que as decisões de escalonamento baseados em afinidade entre processos e processadores fossem realizadas pelo próprio servidor, e não automaticamente pelo escalonador do sistema operacional.

A partir da versão 2.5 do Linux é possível definir manualmente a política de afinidade dos processos em execução [15]. O sistema operacional disponibiliza a chamada de sistema `sched_setaffinity()` que permite

que o programa defina manualmente a afinidade entre seus processos e os processadores disponíveis. A chamada de sistema `sched_getaffinity()` recupera a informação da afinidade entre processos e processadores das estruturas de dados do escalonador. Por padrão, todo processo possui, inicialmente, afinidade com todos os processadores.

Usando-se `sched_setaffinity()`, associamos cada processo a um processador disponível com uma distribuição *round-robin*. O processo coordenador é associado sempre ao processador zero.

Para definir a afinidade de um processo é necessário que esse processo já esteja criado e que, portanto, possua um identificador de processo. Entretanto, como vimos na seção anterior, o escalonador elege para execução o processo filho imediatamente após a criação do processo pelo `fork()`.

Foi necessário, então, modificar o código de criação de processos do servidor do Quake para que o processo filho fosse criado em um estado suspenso, inelegível para utilização de CPU. Ao invés de utilizarmos a chamada de sistema `fork()` para criação do processo, utilizamos a chamada de sistema `clone()` com alguns parâmetros de configuração especiais para que fosse possível manter um controle mais fino da criação do novo processo. Como visto em 3.3.1, `clone()` é o ponto comum de criação de todos os processos e *threads* no sistema operacional Linux.

Substituímos a chamada à `fork()` por uma chamada à `clone()` com os mesmos parâmetros passados pelo `fork()` – exceto pelo parâmetro `SIGCHLD`, como será explicado na Seção 3.4.3 – acrescido do parâmetro `CLONE_STOPPED`, disponível desde a versão 2.6.0 do núcleo do Linux, que faz com que o processo criado seja suspenso, como se este tivesse recebido um sinal do tipo `SIGSTOP`.

Com o processo criado com estado suspenso, podemos definir a afinidade do novo processo através do método `sched_setaffinity()`. Definida a afinidade, o processo coordenador envia um sinal do tipo `SIGCONT` para que a execução do processo auxiliar seja retomada no processador apropriado.

Dessa forma, duas adaptações ao escalonamento dos processos foram realizadas. Os processos filhos não executam imediatamente após a chamada à `fork()`/`clone()` e não executam necessariamente no mesmo processador que o processo pai, viabilizando, portanto, a execução paralela do servidor.

3.4.3 Destruição dos processos auxiliares

Como efeito colateral do fato dos processos auxiliares terem ficado com pouca carga de execução, o custo de criação e de destruição dos processos auxiliares pode chegar a até 30% do tempo total de execução de um processo auxiliar.

A destruição do processo é uma boa candidata à paralelização nesta implementação. Para implementar a destruição paralela dos processos auxiliares, é necessário resolver dois problemas: implementar outra forma de barreira de sincronização que não o uso de `wait()` e resolver o problema dos processos ficarem em estado *zombie* após seu término.

O primeiro problema foi resolvido com o uso de um semáforo. Antes de realizar a chamada à `exit()`, cada processo auxiliar notifica o processador sobre término incrementando o valor do semáforo em um. O processo coordenador, por sua vez, decrementa o semáforo em um número igual ao número de processos auxiliares criados neste quadro do servidor. Dessa forma, o processo coordenador é suspenso até que todos os processos auxiliares terminem suas tarefas. Neste ponto, a liberação dos recursos utilizados por cada processo auxiliar no sistema operacional ocorre em paralelo.

Para resolver o problema dos processos ficarem em estado *zombie*, duas modificações foram feitas. A primeira foi excluir o parâmetro `SIGCHLD` da chamada à `clone()` no momento de criação do processo auxiliar. A exclusão desse parâmetro faz com que o processo filho não notifique o pai sobre alterações no estado de sua execução.

Além disso, utilizamos a chamada de sistema `sigaction()` para configurar o tratamento dos sinais recebidos pelo processo coordenador. Ao passar o parâmetro `SA_NOCLDWAIT` – disponível a partir do Linux versão 2.6 – para a chamada `sigaction()`, o coordenador instrui o sistema operacional a não manter os processos filhos no estado *zombie* caso terminem sua execução.

Dessa forma, o processo coordenador não precisa realizar nenhuma tarefa adicional para impedir que seus processos auxiliares fiquem em estado *zombie* e a liberação dos recursos do sistema operacional utilizados para manter as informações sobre os processos auxiliares é realizada em paralelo à execução da etapa seqüencial do servidor.

4 Parte experimental

Neste capítulo serão apresentados dados que permitem caracterizar o desempenho do protótipo inicial e da versão adaptada da implementação do modelo paralelo para o servidor do Quake.

A Seção 4.1 descreve o ambiente onde os experimentos foram realizados. Na Seção 4.2 descreveremos o impacto que o mapa utilizado durante uma sessão do jogo exerce sobre o desempenho da versão paralela do servidor. A Seção 4.3, caracterizamos a formação dos grupos de entidades próximas durante a execução do servidor e, por fim, a Seção 4.4 descreve as métricas de desempenho analisadas e os resultados obtidos.

4.1 Ambiente de testes

O ambiente de execução dos clientes utilizados nos experimentos foi composto por 5 computadores equipados com processador AMD Athlon^{TM} XP 2800+ (2,25 GHz), 1,0 GB de memória RAM e inter-conectados através de uma rede Ethernet de 100 Mbit/s. Utilizamos o sistema operacional Linux versão 2.6.16.2.

Para a execução dos experimentos, criamos uma versão modificada do cliente do Quake que permite a execução automatizada de uma sessão do cliente.

Os experimentos com a implementação da versão paralela do servidor foram realizados em dois ambientes distintos, um com sistema operacional Solaris e outro com sistema operacional Linux.

Adaptamos o código-fonte do Quake para executá-lo no sistema operacional Solaris. O ambiente de testes Solaris é composto de um computador Sun Fire V880, equipado com 8 processadores UltraSPARC 900 MHz e 32,0 GB de memória RAM. O sistema operacional utilizado foi o Solaris versão 9.

No ambiente Linux, utilizamos um computador equipado com dois processadores Intel Xeon de 3.0 GHz e 2,0 GB de memória RAM. O sistema operacional utilizado foi o Linux versão 2.6.17.7, com os *patches* do *Linux Trace Toolkit* versão 0.5.76 aplicados.

4.2 Influência dos cenários

Desde os primeiros testes com a versão seqüencial do Quake, a influência dos cenários mostrou-se determinante na execução da simulação do jogo.

Três fatores principais explicam o impacto do mapa virtual na execução da simulação:

- número de entidades associadas ao mapa;
- tamanho das mensagens de resposta;
- concentração de usuários em áreas específicas.

4.2.1 Entidades associadas ao mapa

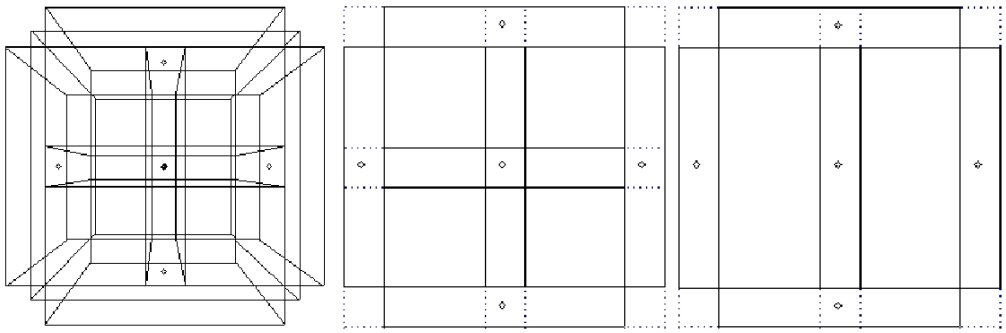
Associado a cada mapa virtual do Quake, existem entidades que devem ser simuladas ao longo da execução do servidor. Podemos classificar tais entidades em dois grupos: entidades que definem os objetivos do cenário e entidades que atribuem funções especiais a objetos do cenário.

Entidades do primeiro tipo definem metas a serem alcançadas pelo jogador. Em geral, assim que todas as metas são alcançadas, o código dessas entidades, escrito em QuakeC, determina que um novo mapa seja carregado pelo servidor.

As entidades do segundo tipo permitem que sejam definidos modos de interação entre um jogador e os elementos do cenário. Elevadores, por exemplo, são definidos como entidades sólidas que disparam a simulação de movimento vertical – simulação esta definida em QuakeC – no momento de sua colisão com um jogador.

Quanto mais complexo for o mapa virtual, maior o número de entidades associadas a ele. Como conseqüência, maior se torna o tempo total de execução da etapa de simulação física do jogo. Além disso, um mapa com muitas entidades faz com que o tamanho dos grupos de entidades próximas aumente, uma vez que, para cada mensagem executada, o servidor deve verificar se algumas dessas entidades foram afetadas pelo movimento.

Em nossos experimentos, foram utilizados mapas que possuem poucas entidades associadas. Cenários idealizados para um grande número de jogadores possuem poucas entidades associadas, uma vez que para esse tipo de jogo a interação entre os usuários é priorizada.



(a) Visão 3D (b) Visão Frontal (c) Visão Superior

Figura 2: Representação estrutural do mapa criado

4.2.2 Tamanho das mensagens de resposta

O cálculo do *Possible Visible Set* (PVN) utilizado na construção das mensagens de resposta é feito de forma eficiente, porém computacionalmente intensa. O uso de uma árvore BSP na implementação faz com que apenas os elementos visíveis pelo jogador sejam considerados na construção da resposta. Elementos separados do jogador por obstáculos como, por exemplo, paredes e portas são desconsiderados na construção de uma mensagem de resposta.

Isso determina a importância que o desenho do labirinto que define o cenário utilizado tem para o desempenho da execução da simulação. Se o mapa for composto por grandes áreas abertas, ou seja, se definir poucos obstáculos como paredes ou portas, a estrutura da árvore BSP gerada por esse mapa será mais simples e, por isso, a computação desta etapa será menos intensa. Entretanto, o tamanho das mensagens será maior, uma vez que mais entidades serão consideradas para inclusão na mensagem de resposta.

Por outro lado, mapas formados por áreas menores, compostos por pequenas salas ou corredores, tornam o processo de criação da possível área de alcance do jogador computacionalmente mais difícil. Porém, o conjunto de entidades contidos na área será muito menor e, por isso, o tamanho das mensagens será menor.

4.2.3 Concentração de usuários em áreas específicas

A configuração do labirinto virtual e os objetivos de cada cenário podem induzir a existência de áreas com diferentes graus de importância no mapa virtual do jogo. Áreas de maior importância tendem a concentrar um maior conjunto de jogadores. Essa concentração pode ser explicada pelo desenho do labirinto ou pelas entidades definidas pelo cenário.

O desenho do labirinto pode propiciar o aparecimento de uma área de maior importância. Por exemplo, suponha um mapa virtual composto de dois andares, onde não há forma do jogador voltar para o andar superior. Com o decorrer da simulação, o andar inferior se tornará uma área de maior importância. Do mesmo modo, o objetivo do cenário pode criar áreas mais importantes.

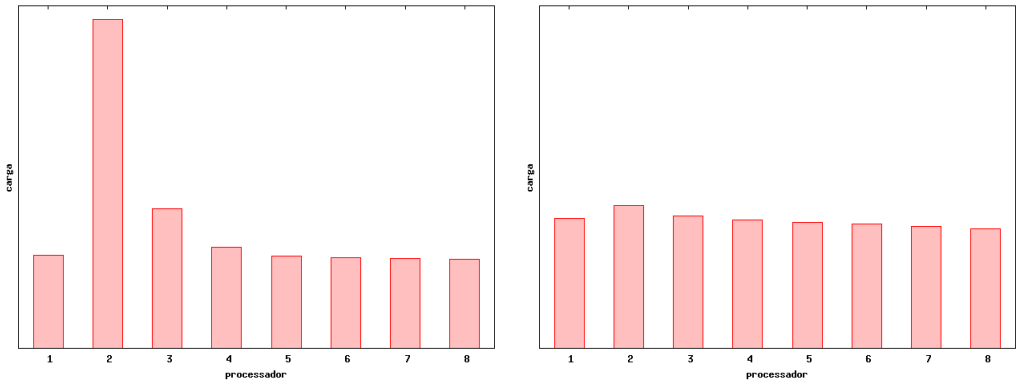
O modelo de paralelização proposto por esse trabalho não trata áreas de concentração como casos especiais. Por isso, áreas de concentração podem levar à criação de grupos grandes de entidades próximas e, possivelmente, a algum desbalanceamento de carga.

Para analisar o desempenho da implementação sem a questão das áreas de concentração, criamos um mapa para o jogo Quake que não contém áreas especiais. O mapa consiste de um cubo, dividido por duas paredes que o dividem transversalmente e longitudinalmente. Não há entidades que definam metas no mapa. A Figura 2 mostra a representação estrutural do mapa criado.

Os resultados obtidos com a utilização dos diferentes mapas serão discutidos na Seção 4.3.

4.3 Distribuição dinâmica de carga

Para analisar os resultados obtidos com a metodologia de distribuição de carga é necessário entender qual o efeito que o mapa utilizado pelo servidor exerce sobre a criação dos grupos de entidades próximas e como estes se caracterizam.



(a) Distribuição de tarefas no mapa *death32c* (b) Distribuição de tarefas no mapa *cubo*

Figura 3: Distribuição das tarefas entre os processadores

4.3.1 Áreas de concentração de mapas

Realizamos experimentos com um dos mapas disponibilizados pela id Software (*death32c* [11]) para jogos do tipo *deathmatch*. No ambiente de testes Solaris, a execução mostrou um grande desbalanceamento de carga em um dos processadores, como mostra a Figura 3(a).

Investigações através de instrumentação do código-fonte mostraram que em todo quadro de simulação havia um grupo com um número de elementos muito maior do que o número de elementos dos demais grupos. A existência de tal grupo apontou a possível existência de uma área de concentração de usuários no mapa utilizado para os experimentos, conforme explicado na Seção 4.2.3.

Para validar a hipótese de existência de uma área de concentração, repetimos os experimentos no mapa que denominamos *cubo*. Esse mapa, por construção, não possui áreas de concentração. Os resultados obtidos são apresentados de forma normalizada na Figura 3(b), mostram que, de fato, o desbalanceamento de carga era causado pelo mapa utilizado.

4.3.2 Caracterização dos grupos

Modificamos o código do Quake a fim de colher estatísticas sobre a quantidade e tamanho dos grupos gerados durante a execução do servidor.

Utilizando o mapa *death32c*, realizamos uma análise da criação dos grupos na versão seqüencial do servidor. Os resultados mostram que apenas 4 grupos são criados em cada quadro de simulação. Cada grupo, por sua vez, possui uma média de 7 entidades por grupo.

A execução da versão paralela do servidor mostra um resultado diferente. A execução no ambiente Linux, com a utilização de um processo coordenador e um processo auxiliar, resultou na criação de 9 grupos por quadro de simulação onde cada grupo possuía, em média, 8 entidades.

Tal diferença ocorre por causa da diferença no tamanho dos quadros de simulação entre as versões paralela e seqüencial, como é explicado na Seção 4.4.

Utilizando o mapa *cubo* no servidor paralelo, no mesmo ambiente de testes, verificamos que em média são criados 20 grupos de entidades próximas que possuem uma média de 4 entidades por grupo.

A diferença entre as execuções com mapas diferentes são explicadas pelo desenho do mapa. Como não há áreas de concentração no mapa *cubo*, os usuários estão melhor distribuídos no espaço e, portanto, existe a tendência da criação de grupos menores.

4.4 Métricas de desempenho

4.4.1 Motivação

Tradicionalmente [7, 14, 16], servidores web são avaliados de acordo com sua vazão (*throughput*), isto é, o número de requisições atendidas por segundo. Optamos por analisar o servidor do Quake através de duas métricas: quantidade de quadros processados por segundo e quantidade de mensagens enviadas por quadro de simulação.

O número de quadros de processamento realizados por segundo pelo servidor nos dá informações sobre a velocidade com que a simulação é realizada e sobre a latência da comunicação entre clientes e servidor. Um número pequeno de quadros por segundo pode indicar a saturação do servidor.

O número de mensagens enviadas pelo servidor por quadro de simulação indica qual a vazão (*throughput*) do servidor. Nos permite avaliar, também, qual o número médio de clientes que são processados em cada quadro do servidor.

Todos os testes apresentados nesta seção foram realizados utilizando-se 160 clientes simultâneos.

4.4.2 Análise da versão seqüencial

Avaliamos, primeiramente, o desempenho da versão original do Quake. A execução no ambiente de testes Linux mostrou que são executados 2.250 quadros por segundo, mas que são enviadas apenas 0,35 respostas por quadro de simulação do servidor.

O baixo número de respostas é conseqüência do mecanismo de controle de utilização de banda de rede empregado pelo servidor. Esse mecanismo evita que os clientes recebam muitas mensagens em um período muito curto de tempo. Com a utilização de redes mais rápidas, a velocidade com que as mensagens são enviadas e recebidas pelo cliente é maior e, por isso, há um aumento no número de mensagens trocadas entre cliente e servidor. Isso leva o mecanismo de controle de banda a impedir o envio de muitas das mensagens e faz com que alguns quadros de simulação sejam executados sem que nenhuma mensagem seja enviada ao seu final.

Modificamos o servidor original para que não fosse utilizado esse controle. Nesse caso, o número de quadros por segundo cai para cerca de 2.050, porém o número de mensagens por quadro sobe para 2. A queda no número de quadros por segundo se deve ao esforço extra do servidor para o envio das mensagens.

4.4.3 Custo do cálculo de grupos

A fim de avaliar o impacto do esforço de criação dos grupos no desempenho do servidor, executamos a versão paralela de forma que os grupos fossem calculados, mas que não fosse criado nenhum processo auxiliar. Dessa maneira, toda a simulação ocorre em um mesmo processador. O número de quadros de simulação por segundo caiu para 420. O número de mensagens por quadro, porém, aumentou para 10.

Para entender o motivo da queda de 2.050 para 420 quadros por segundo, é necessário entender a execução de uma sessão do jogo Quake como sendo a execução de uma simulação distribuída.

Ao mesmo tempo em que o servidor está executando a simulação de um quadro, os clientes estão processando as mensagens recebidas, capturando a entrada dos usuários e enviando novas mensagens para o servidor.

Com o custo extra de processamento ocasionado pela criação de quadros, há mais tempo para o servidor acumular mensagens durante o processamento de um quadro. Portanto, o aumento do tempo de simulação de um quadro acarreta em um acúmulo de mensagens para o próximo quadro, o que implica no aumento da quantidade de tarefas que um quadro de simulação precisa realizar.

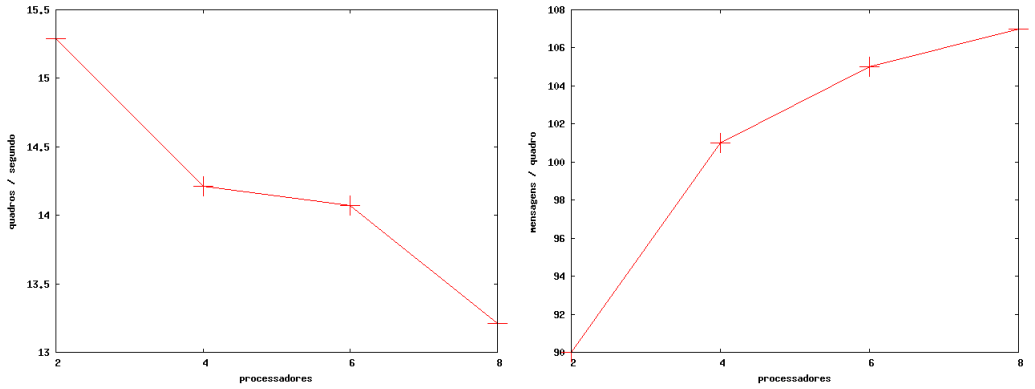
O aumento na quantidade de tarefas realizadas por cada quadro de simulação explica tanto a queda do número de quadros por segundo, como o aumento no número de mensagens por quadro. Com o aumento do número de mensagens tratadas em um mesmo quadro, ocorre o aumento da quantidade de entidades ativas no quadro de simulação e, conseqüentemente, um aumento no tamanho de cada grupo de entidades próximas, como verificamos na Seção 4.3.2.

4.4.4 Resultados iniciais

Os primeiros experimentos com a versão paralela em ambientes multiprocessados não produziram os resultados esperados. Como explicado na Seção 3.4, técnicas empregadas pelos escalonadores de processos dos sistemas operacionais modernos privilegiam a execução de processos criados através da chamada de sistema `fork()` em um mesmo processador a fim de aproveitar melhor a afinidade de *cache* entre processos e processadores.

Os resultados experimentais com a primeira implementação da versão paralela refletem essa característica dos sistemas operacionais testados.

A Figura 4 mostra os dados das métricas de desempenho coletadas em simulações com 160 clientes, utilizando-se 2, 4, 6 e 8 processadores. Os resultados deixam claro que a criação de processos auxiliares



(a) Quadros executados por segundo (b) Pacotes enviados por quadro

Figura 4: Métricas de desempenho coletadas no ambiente Solaris

apenas aumenta o tamanho de cada quadro, devido ao custo de criação dos processos auxiliares. Devido ao aumento no tamanho de cada quadro, o número de mensagens processadas também aumenta.

No ambiente Linux, os testes com dois processadores mostraram que o servidor executa 95 quadros por segundo e envia, em média, 33 pacotes por quadro.

A prova cabal de que o escalonador de processos estava afetando o nível de paralelismo do servidor se deu com a análise dos traços de execução através do *Linux Trace Toolkit*. O traço gerado pelo LTT deixou claro que apenas uma das duas CPUs disponíveis estava sendo utilizada durante a execução do experimento.

4.4.5 Resultados da versão final

As adaptações descritas na Seção 3.4.1 foram realizadas utilizando recursos específicos para o sistema operacional Linux e, por esse motivo, os testes não puderam ser repetidos no ambiente Solaris. Para garantir a precisão dos resultados, repetimos cada experimento com a versão final pelo menos 30 vezes.

A análise da versão com a nova implementação de criação de processos auxiliares – descrita na Seção 3.4.2 – mostrou bons resultados. A quantidade de quadros executados por segundo subiu de 95 para 198 quadros por segundo. A taxa de mensagens tratadas caiu de 33 para 18 pacotes por segundo.

Os traços da execução, porém, demonstraram que a utilização das CPUs disponíveis ainda era ineficiente. A destruição do processo filho correspondia a cerca de 30% do tempo total de execução do processo auxiliar. Além disso, durante toda a destruição do processo, o sistema rodava seqüencialmente, uma vez que ao executar o método `wait()`, o processo pai fica suspenso até que todos os recursos utilizados pelo sistema operacional sejam liberados.

A versão adaptada para destruição paralela dos processos, descrita na Seção 3.4.3, apresentou um melhor desempenho. O número médio de quadros por segundo aumentou para 350 (com desvio padrão $\sigma = 6,12$) e a taxa de mensagens simuladas foi de 11 mensagens por quadro de simulação ($\sigma = 0,64$).

A Tabela 1 mostra um sumário com os resultados.

Implementação Avaliada	quadros/s	mensagens/quadro	mensagens/s
versão com `fork()` e `wait()`	95	33	3.135
versão com `clone()` e `wait()`	198	18	3.564
versão com `clone()` e sem `wait()`	350	11	3.850

Tabela 1: Resultados das medições de desempenho da versão paralela do servidor do Quake no ambiente Linux

4.4.6 Análise do traço de execução

A análise do traço de execução mostra dados interessantes sobre a versão final, testada no ambiente Linux.

O traço da com o mapa *death32c* mostra que o custo total de utilização de CPU com a cópia das páginas de memória, ocasionadas principalmente pelo mecanismo de *copy-on-write* do Linux, corresponde a 4,69% do total de execução. O custo com a execução da chamada de sistema `clone()` – correspondente ao custo da criação de um novo processo, sem contar o custo das cópias de página de memória – corresponde a 2,12% do tempo total da execução.

Em função do paralelismo, podemos dividir o processamento de um quadro de simulação em 4 fases.

Na primeira fase, o quadro de simulação acaba de iniciar e executa em paralelo com a destruição do processo auxiliar utilizado pelo quadro anterior. Esta etapa corresponde a 17,93% ($\sigma = 0,51\%$) do tempo total de execução do quadro de simulação.

Durante a segunda fase, o processo pai termina de executar a etapa seqüencial do servidor do Quake. Nesta etapa não há paralelismo e são consumidos 21,66% ($\sigma = 0,67\%$) do tempo total de execução do quadro.

A terceira fase corresponde ao tratamento paralelo da fila de eventos, até que o primeiro processo auxiliar termine de processar todas as suas tarefas. Esta fase é paralela e corresponde a 37,69% ($\sigma = 0,42\%$) do tempo de execução do quadro.

Por fim, a última fase compreende o término das tarefas do último processo auxiliar, somada a etapa de sincronização global entre os processadores. Esta fase é seqüencial e consome 22,70% ($\sigma = 0,67\%$) do tempo total de execução do quadro de simulação.

Temos, portanto, a utilização das duas CPUs disponíveis em 55,65% do tempo.

A análise de uma sessão utilizando o mapa *cubo* mostrou que 5,05% do tempo total de utilização de CPU foram gastos com o tratamento de falhas de páginas. As chamadas à `clone()` corresponderam a 1,49% do tempo total da execução.

Quanto às taxas de utilização de CPU, 19,13% ($\sigma = 0,83\%$) do tempo foi gasto na primeira fase, 22,77% ($\sigma = 0,97\%$) na segunda fase, 34,43% ($\sigma = 0,63\%$) na terceira fase e 23,65% ($\sigma = 0,97\%$) na última fase. No total, há utilização simultânea das duas CPUs disponíveis em 53,56% do tempo.

A Figura 5 mostra o diagrama de Gantt que representa dois dos quadros de simulação observados durante a execução da versão paralela do servidor do Quake. Estão indicadas na figura, também, as fases de paralelização que caracterizam a versão final do servidor.

5 Conclusão

5.1 Comentários finais

Neste trabalho, estudamos arquiteturas de sistemas de computação que almejam a criação de serviços que consigam atender um grande número de usuários simultâneos sem que apresentem degradação em seu desempenho. Particularmente, estudamos técnicas e questões relacionadas ao sistema operacional que permitem a utilização eficiente dos recursos computacionais disponíveis.

Das técnicas estudadas, apresentamos aquelas que permitem que sistemas baseados em eventos executem de forma eficiente em ambientes multiprocessados. Tais técnicas abordam arquiteturas de *software* que permitem utilização eficiente de todos os processadores disponíveis e métodos que avaliam o custo de algumas operações com o sistema operacional e que propõem maneiras de amenizá-las.

Baseados nessas técnicas, desenvolvemos um modelo para a paralelização do jogo interativo, multi-usuário, Quake. A análise do funcionamento do Quake mostrou que o jogo nada mais é do que um sistema de simulação distribuída, onde os eventos de entrada são gerados pelos jogadores conectados a uma sessão da simulação e enviados para serem processados pelo servidor. O servidor realiza a simulação do modelo físico dos objetos e do modelo do jogo e consolida o novo estado da simulação. O novo estado é, então, enviado novamente para os clientes. A simulação dos modelos físico e lógico e o tratamento de mensagens e envio de respostas é realizado utilizando-se uma arquitetura baseada em eventos.

Abdelkhalek propôs um modelo de paralelização de granularidade fina com atribuição estática de tarefas. Apesar do modelo de proteção contra modificações concorrentes levar em consideração o posicionamento das entidades, a distribuição de tarefas entre os processadores disponíveis não considera as modificações no posicionamento das entidades ao longo da simulação. Isso aliado ao controle fino de concorrência, levou a graves problemas no desempenho. Contenção de *locks* e desbalanceamento de carga fizeram com que a versão inicial de sua implementação tivesse alguma CPU ociosa em até 70% do tempo total da simulação.

Neste trabalho propusemos um novo modelo de paralelização para o servidor do Quake que utiliza a semântica da simulação para realizar a distribuição dinâmica de carga entre os processadores disponíveis.

Utilizamos o conceito de grupos de entidades próximas para agrupar as entidades que estão próximas entre si. Qualquer ação realizada por um elemento contido em um grupo só poderá causar algum efeito colateral em um elemento contido no mesmo grupo.

Dessa forma, foi possível simular cada um dos grupos de entidades próximas em processadores diferentes, sem que fosse necessário nenhum tipo de sincronização de acesso concorrente durante o processamento dos eventos de um quadro de simulação. Nossa implementação cria um processo por processador disponível, distribui os grupos de entidades próximas que possuem entidades ativas entre os processos criados e, a partir daí, cada processo interpreta as mensagens recebidas pelo servidor e constrói as mensagens de resposta. Cada processo funciona como uma instância independente do servidor.

A comunicação inter-processos ocorre em dois momentos: durante a criação e imediatamente antes da destruição dos processos auxiliares. Nossa implementação utiliza-se da semântica de *copy-on-write* de cópia de páginas de memória que os sistemas operacionais baseados no Unix apresentam quando processos são criados através da chamada de sistema `fork()`. Após a criação dos processos auxiliares, são copiadas para os processos auxiliares apenas as páginas de memória que sofrerem alguma modificação. Conseguimos, assim, evitar a cópia de informações que são utilizadas apenas para leitura e das que não são utilizadas em nenhum momento do processamento dos eventos de um determinado quadro. Nossos experimentos mostraram que o tempo de CPU gasto com essas cópias de páginas corresponde a, aproximadamente, 5% do tempo total de execução e o custo com a criação de processos auxiliares corresponde a cerca de 2% do tempo total de execução.

Os primeiros experimentos mostraram resultados aquém dos esperados. Dois foram os principais motivos: cenário da simulação e questões relacionadas ao escalonamento de processos em sistemas operacionais que aplicam o conceito de afinidade entre processos e processadores.

Mostramos que o modelo de distribuição dinâmica de cargas proposto por este trabalho é afetado diretamente pela existência de áreas de concentração do cenário, ou seja, áreas de maior interesse para os jogadores ou que, devido ao desenho do labirinto, acumulam maiores quantidades de jogadores. Mostramos, assim, que o desenho do labirinto e o tipo de jogo afetam diretamente o desempenho dessa classe de aplicação.

Descobrimos, também, que a simulação realizada pelo Quake determina que modificações no tempo de duração de cada quadro de simulação do jogo impliquem em variação na carga de trabalho realizada em cada quadro de simulação. Quanto maior for o quadro anterior, maior será o número de mensagens acumuladas para processamento no quadro seguinte.

Além disso, percebemos que aplicações que criam processos auxiliares de curta duração (via `fork()` ou `clone()`) têm seu desempenho afetado por sistemas operacionais que implementam políticas de escalonamento de processos que privilegiem a execução inicial no mesmo processador para aproveitar melhor a afinidade existente entre o processador e o processo pai.

Criamos uma metodologia de criação e destruição de processos que se adequa melhor a execução da implementação da metodologia de paralelização proposta ao sistema operacional. A utilização dessa metodologia permitiu que o servidor executasse 390 quadros por segundo, sendo capaz de servir cerca de 3.850 mensagens por segundo.

Por fim, a análise dos traços de execução revelaram que foi possível manter o paralelismo do servidor em cerca de 55% do tempo total de execução.

5.2 Trabalhos futuros

A análise do modelo de paralelismo foi profundamente prejudicada pela limitação de 160 clientes imposta pelo protocolo de troca de mensagens do Quake. O pequeno número de clientes e, conseqüentemente, o pequeno número de grupos de entidades simulados por quadro de simulação fez com que cada processo auxiliar se ocupasse de poucas entidades por quadro de simulação.

Uma solução seria reescrever o protocolo de troca de mensagens do servidor. Para a realização desse projeto, seria necessário remodelar o protocolo para que aceitasse campos com valores maiores ou de tamanho variável. Seria necessário, também, estudar todos os objetos escritos em QuakeC para que as entidades que enviam mensagens passem a respeitar o novo protocolo.

Outra forma de estudar os efeitos de um número maior de jogadores simultâneos seria a criação de um sistema simulador que abstraísse a carga de execução (*workload*) e simulasse os efeitos do processamento de um número maior de entidades em ambientes multiprocessados, sem que fosse necessário levar em consideração detalhes sobre o processamento do servidor. Essa abordagem é utilizada em outros trabalhos sobre escalabilidade de jogos como [20] e [21].

O método de comunicação inter-processos proposto explorou o mecanismo de *copy-on-write* dos sistemas Unix baseados no *System V* para copiar as informações computadas durante a fase seqüencial do servidor. Para utilizar esse mecanismo, o servidor cria um novo processo auxiliar por processador disponível em cada quadro do servidor. Consideramos que seria interessante comparar esse mecanismo de comunicação inter-processos com uma implementação que fizesse a cópia explícita das informações e utilizasse um *pool* de processos ou *threads*. Gostaríamos de avaliar os ganhos que o mecanismo de *copy-on-write* produz e a sobrecarga devido a criação dos processos.

A versão final adaptada explorou aspectos de gerenciamento e criação de processos específicos do sistema operacional Linux. Outros sistemas operacionais poderiam ser explorados. Particularmente, o sistema operacional K42 [1] seria adequado para a realização de experimentos futuros.

K42 é um sistema operacional para fins de pesquisa sendo desenvolvido no IBM T. J. Watson Research Center. O K42 possui uma inovadora estrutura interna que oferece ao usuário a interface do Linux [5]: binários de aplicações que executam no Linux executam no K42 sem modificações. O sistema operacional K42 foi projetado para multiprocessadores de 64 bits com memória compartilhada.

Além disso, escalabilidade é um dos pontos cruciais deste sistema operacional, que foi concebido para máquinas com centenas (ou milhares) de processadores. Para que isso seja possível, não são utilizadas estruturas de dados ou *locks* globais pois constatou-se que estes são usualmente gargalos para se atingir escalabilidade nas implementações tradicionais do Unix. Resultados experimentais e comparações entre o Linux e o K42 foram apresentados em [6].

Duas características do K42 seriam particularmente úteis para uma futura investigação: (1) a infra-estrutura para rastreamento de execução [23] que permite um monitoramento sofisticado do comportamento dos *locks* do sistema operacional, e (2) muitos dos serviços, que usualmente estão presentes no *kernel*, são implementados no K42 em espaço do usuário, tornando muito simples fazer otimizações como as propostas por Rosu em [18].

5.3 Publicações

Os resultados obtidos durante este trabalho de mestrado foram descritos em um artigo científico intitulado ***Load Balancing on an Interactive Multiplayer Game Server*** que será apresentado no *track* de *Scheduling and load-balancing* da *The 13th European Conference on Parallel and Distributed Computing* (**Euro-Par 2007**) [8].

Referências

[1] The K42 Project. Disponível em: `http://www.research.ibm.com/K42/`. Acesso em: 02/10/2006.

[2] Abdelkhalek, A., and Bilas, A. Parallelization and performance of interactive multiplayer game servers. In *Proceedings of 18th International Parallel and Distributed Processing Symposium* (Apr. 2004), IEEE Computer Society, p. 72a.

[3] Abdelkhalek, A., Bilas, A., and Moshovos, A. Behavior and performance of interactive multiplayer game servers. In *Proceedings of the International IEEE Symposium on the Performance Analysis of Systems and Software (ISPASS-2001)* (Arizona, USA, Nov. 2001).

[4] Abdelkhalek, A., Bilas, A., and Moshovos, A. Behavior and performance of interactive multiplayer game servers. *Cluster Computing 6*, 4 (Oct. 2003), 355–366.

[5] Appavoo, J., Auslander, M., Edelsohn, D., da Silva, D., Krieger, O., Ostrowski, M., Rosenburg, B., Wisniewski, R. W., and Xenidis, J. Providing a Linux API on the scalable K42 kernel. In *Freenix* (San Antonio, TX, EUA, June 2003), pp. 323–336.

[6] Appavoo, J., Auslander, M., Silva, D. D., Krieger, O., Ostrowski, M., Rosenburg, B., Wisniewski, R. W., Xenidis, J., Stumm, M., Gamsa, B., Azimi, R., Fingas, R., Tam, A., and Tam, D. Enabling scalable performance for general purpose workloads on shared memory multiprocessors. Tech. rep., IBM Research Technical Report RC22863, 2003.

[7] Brecht, T., Pariag, D., and Gammo, L. accept()able strategies for improving web server performance. In *Proceedings of the USENIX 2004 Annual Technical Conference* (Boston, EUA, June 2004), pp. 227–240.

[8] Cordeiro, D., Goldman, A., and da Silva, D. Load balancing on an interactive multiplayer game server. In *A ser publicado em proceedings of The 13th European Conference on Parallel and Distributed Computing (Euro-Par)* (2007).

[9] Flynn, M. Some computer organizations and their effectiveness. *IEEE Transactions on Computers C-21* (Sept. 1972), 948–960.

[10] Graham, R. Bounds on multiprocessor timing anomalies. *SIAM Journal on Applied Math 17* (1969), 416–429.

[11] id Software. Conjunto de mapas para jogos do tipo *deathmatch*. Disponível em: `ftp://ftp.idsoftware.com/idstuff/quakeworld/maps/`. Acesso em: 12/09/2006.

[12] id Software. Jogo quakeworld. Disponível em: `http://www.idsoftware.com/games/quake/quake/`. Acesso em: 07/09/2006.

[13] Jr., E. C., and 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* (New York, NY, USA, 1976), ACM Press, pp. 306–310.

[14] Levy-Abegnoli, E., Iyengar, A., Song, J., and Dias, D. M. Design and performance of a web server accelerator. In *INFOCOM (1)* (1999), pp. 135–143.

[15] Love, R. Kernel korner: Cpu affinity. *Linux Journal 2003*, 111 (2003), 8.

[16] Pai, V. S., Druschel, P., and 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.

[17] Raymond, E. S. *The Art of UNIX Programming*. Pearson Education, 2003.

[18] Rosu, M.-C., and Rosu, D. Kernel support for faster web proxies. In *Proceedings of the USENIX 2003 Annual Technical Conference* (Texas, EUA, June 2003), pp. 225–238.

[19] Shimer, C. Binary space partition trees. Disponível em: `http://www.cs.wpi.edu/~matt/courses/cs563/talks/bsp/bsp.html`, 1997. Acesso: em 07/09/2006.

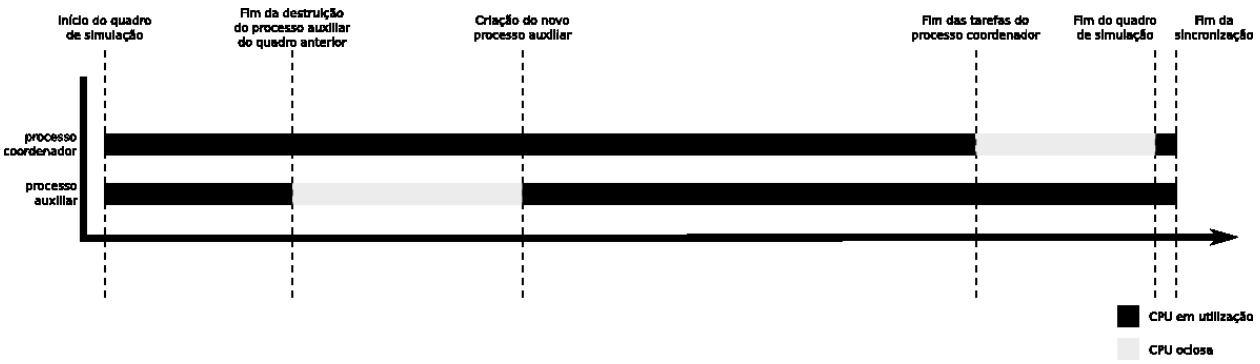
[20] Tveit, A. Empirical performance evaluation of the zereal massively multiplayer online game simulator. Tech. rep., Norges teknisk-naturvitenskapelige universitet, Nov. 2003.

[21] Tveit, A., Øyvind Rein, Iversen, J., and Matskin, M. Scalable agent-based simulation of players in massively multiplayer online games. In *Proceedings of the 8th Scandinavian Conference on Artificial Intelligence* (Bergen, Norway, Nov. 2003), IOS Press, pp. 80–89.

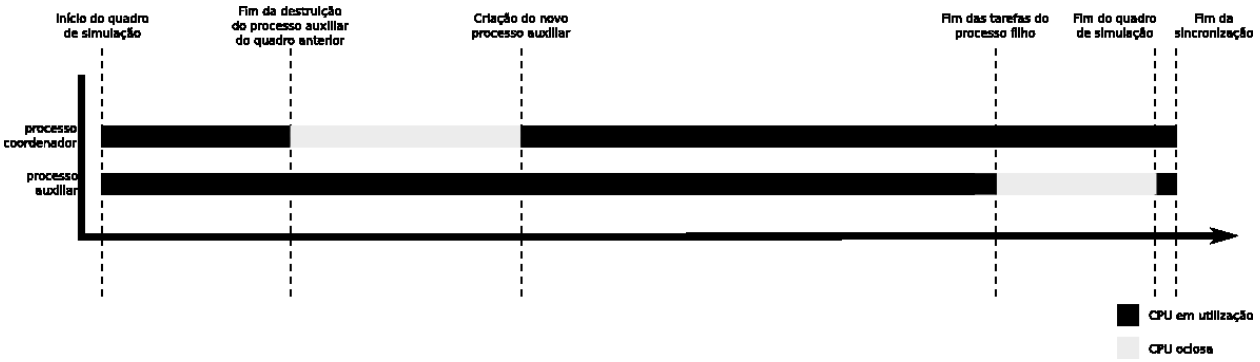
[22] Valiant, L. A bridging model for parallel computation. *Communications of the ACM 33*, 8 (1990), 103–111.

[23] Wisniewski, R. W., and Rosenburg, B. Efficient, unified, and scalable performance monitoring for multiprocessor operating systems. In *Proceedings of the 2003 ACM/IEEE conference on Supercomputing* (Washington, DC, USA, 2003), IEEE Computer Society, p. 3.

[24] Zeldovich, N., Yip, A., Dabek, F., Morris, R., Mazieres, D., and Kaashoek, F. Multiprocessor support for event-driven programs. In *Proceedings of the 2003 USENIX Annual Technical Conference* (June 2003), pp. 239–252.



(a) Exemplo de quadro onde o processo coordenador termina primeiro de processar seus eventos



(b) Exemplo de quadro onde o processo filho termina primeiro de executar seus eventos

Figura 5: Distribuição do processamento entre as CPUs disponíveis