Trabalho Programação Paralela e Distribuída: Servidor *Web Multithread*

André Aranda[1], **Caio Lara**[2], **Carlos Pereira**[3]

[1]Departamento de Ciência da Computação
Universidade Federal de Lavras (UFLA) – Lavras, MG – Brasil

{andre,caio_freitas,carlospereira}@computacao.ufla.br

Abstract. *This article discribe the process of development of a aplication that works with a web server in Java. This aplication recive multiple requisitions of clients aplications and process this requisitions in parallel through of Multithreads with the objective of gain performance in relation to a sequencial server. This paper discussed tools used, implementation techniques, tests and results.*

Resumo. *Este trabalho descreve o processo de desenvolvimento de uma aplicação que funcione como servidor web em Java. Essa aplicação recebe várias requisições de aplicações clientes e processa as requisições em paralelo atravez de multithreads com o objetivo de ganhar desempenho em relação a um servidor implementado de forma em sequencial. No trabalho são discutidos as ferramentas utilizadas, as técnicas de implementação, testes e resultados.*

1. Introdução

A *World Wide Web* também conhecida em português como 'Rede Mundial de Computadores' é a combinação de todos recursos e usuários da Internet que estão usando o protocolo HTTP (*Hypertext Transfer Protocol*) [Rouse 2017]. Dois elementos fundamentais da World Wide Web são os servidores web e os clientes (usuários).

Um servidor web é um software que usa o protocolo HTTP (*Hypertext Transfer Protocol*) para servir os arquivos que formam páginas web para usuários, em resposta a suas requisições, que são encaminhados pelos clientes HTTP de seus computadores [Rouse 2015].

O cliente costuma utilizar um navegador como softare cliente para acessar os arquivos de um servidor web. Através deste navegador, é possível, por exemplo, visualizar páginas HTML (*Hypertext Markup Language*) com textos, imagens, sons e vídeos, com estilos definidos em CSS (*Cascading Style Sheets*) e scripts escritos em Javascript. Essas três tecnologias são as mais populares na World Wide Web. [Flanagan 2011]

Em janeiro de 2012, um pouco mais de 2 bilhões de pessoas tinham acesso a Internet o que representa 30% da população mundial. Cinco anos depois metade da população mundial tinha acesso a Internet o que soma 3.7 bilhões de pessoas [Kemp 2017]. Nota-se que há um grande crescimento de usuários na Internet, e para que todos esses usuários consigam acessar os recursos que desejam na Web são necessárias boa infraestrutura de rede e bons servidores web. Este trabalho irá abordar a implementação de um servidor web multithread para atender multiplos clientes de forma mais rápida.

Este trabalho está organizado da seguinte forma, a Seção 2 demonstra as vantagens de servidores web multithread. A Seção 3 explica a implementação do pool de threads com fila sincronizada (produzidor/consumidor) para tratamento de requisições com múltiplas threads. A seção 4 por sua vez descreve como a aplicação anexa a este trabalho deve ser executada e em qual ambiente. A Seção 5 define a metodologia dos testes realizados no servidor. A Seção 6 discute os resultados obtidos com os testes realizados. A Seção 7 conclui as vantagens de servidores *web multithreads*.

2. Servidor Web Multithread

Thread é um pequeno programa que trabalha como um subsistema, sendo uma forma de um processo se autodividir em duas ou mais tarefas. As diversas threads que existem em um programa podem trocar dados e informações entre si e compartilhar os mesmos recursos do sistema, incluindo o mesmo espaço de memória.

Ainda conforme o mesmo autor as threads se beneficiam de equipamentos com CPUs com mais de um núcleo, onde cada núcleo será responsável pela execução de uma *thread*.

Aplicações Web costumam receber inumeras requisições por hora. Ao Processar essas informações de forma paralela pode-se obter uma ganho em relação ao processamento de forma sequencial. Porém, é necessário gerenciar a forma com que as threads trabalharão neste servidor visto que as requisições podem variar muito, em determinado instante o servidor pode receber determinada carga de requisições.

Ao lidar com *threads*, encontramos alguns problemas como lidar com dados compartilhados de uma thread com a outra.

Neste contexto temos a abordagem da técnica produtor/consumidor, onde uma thread produz dados para serem consumidas por outra thread, onde é necessário discutir como transferir dados de forma que o produtor e o consumidor nunca fiquem inativos sem necessidade e evitar problemas como produtor insere dados em um local que já possuia dados que não foram consumidos, acontecendo uma sobrescrita inadequada. Outra situação que podemos citar é que o consumidor busque o dado em um local que já foi consumido.

3. Pool de threads com fila sincronizada (produtor/consumidor)

Na implementação do servidor web multithread foi utilizado um componente chamado Executor que gerencia um pool de threads que são responsáveis por executar as tarefas submetidas ao Executor. Para o gerenciamento das tarefas no Executor é utilizada uma fila sincronizada no modelo produtor/consumidor.

O Algoritmo 1 demonstra o comportamento que as work threads do pool de threads executam. Nota-se que essas threads estão em um loop infinito de remover tarefas da fila de tarefas e executar essas tarefas.

As operações da fila de tarefas sincronizada no modelo produtor/consumidor são executadas de forma bloqueante, para isso utiliza-se dois semáforos. Um destes semafóros, $fillCount$, controla a quantidade de tarefas enfileiradas. A cada tarefa produzida (enfileirada) libera um passe no semáforo $fillCount$ e antes de consumir (desenfileirar) uma tarefa é preciso adquirir um passe desse semáforo. Portanto, enquanto

não houver tarefas para consumir não haverá passes no semáforo $fillCount$, e assim os consumidores ficarão bloqueados.

O outro semáforo é o $emptyCount$ que controla a quantidade de espaço na fila, este semáforo é inicializado com a capacidade máxima da fila em passes. Antes de produzir (enfileirar) cada tarefa deve adquirir um passe desse semáforo e a cada tarefa consumida (desenfileirada) libera um passe nesse semáforo. Portanto, quando a fila está na sua capacidade máxima, não existiram passes no semáforo $emptyCount$, então a operação de produzir uma tarefa fica bloqueada até que um consumidor libere um passe por consumir uma tarefa.

Para entender melhor o uso dos semáforos para controlar as operações de inserção(produção)/remoção(consumo) da fila de tarefas, os Algoritmos 2 e 3, respectivamente, demonstram essas operações passo-a-passo.

Algoritmo 1 Comportamento work threads

1: $queue \leftarrow$ fila de tarefas
2: **loop**
3: $\quad tarefa \leftarrow$ remove tarefa da $queue$
4: $\quad$ executa $tarefa$

Algoritmo 2 Inserção de tarefa da fila de tarefas

1: $elem \leftarrow$ elemento a ser inserido
2: $emptyCount \leftarrow$ semáforo da capacidade da fila
3: adquire passe $emptyCount$
4: insere $elem$ de forma sincronizada na fila
5: $fillCount \leftarrow$ semáforo de elementos produzidos
6: libera passe $fillCount$

Algoritmo 3 Remoção de tarefa da fila de tarefas

1: $fillCount \leftarrow$ semáforo de elementos produzidos
2: adquire passe $fillCount$
3: $elem \leftarrow$ remove elemento da fila de forma sincronizada
4: $emptyCount \leftarrow$ semáforo da capacidade da fila
5: libera passe $emptyCount$
6: **return** $elem$

Na Figura 1 pode-se observar o diagrama de classes da implementação do servidor web multithread.

- **ArgsHandler** é responsável por tratar os argumentos da aplicação e realizar a configuração do servidor.
- **MultithreadedWebServer** é responsável por configurar e inicializar o servidor web.
- **MyHttpServer** é responsável por aceitar as conexões HTTP dos clientes e submetê-las para que um executor trate essas requisições.

- **HttpSocketHandler** é responsável por tratar as requisições HTTP do servidor web, entregando os recursos requisitados ao cliente.
- **ServerConfiguration** representa a configuração do servidor web.
- **MyBlockingQueueImp** implementação da fila sincronizada produtor/consumidor.
- **MyThreadPoolExecutor** implementação do Executor que gerencia o pool de threads.
- **MyThreadPoolRunnable** representa o comportamento de execução das work threads.

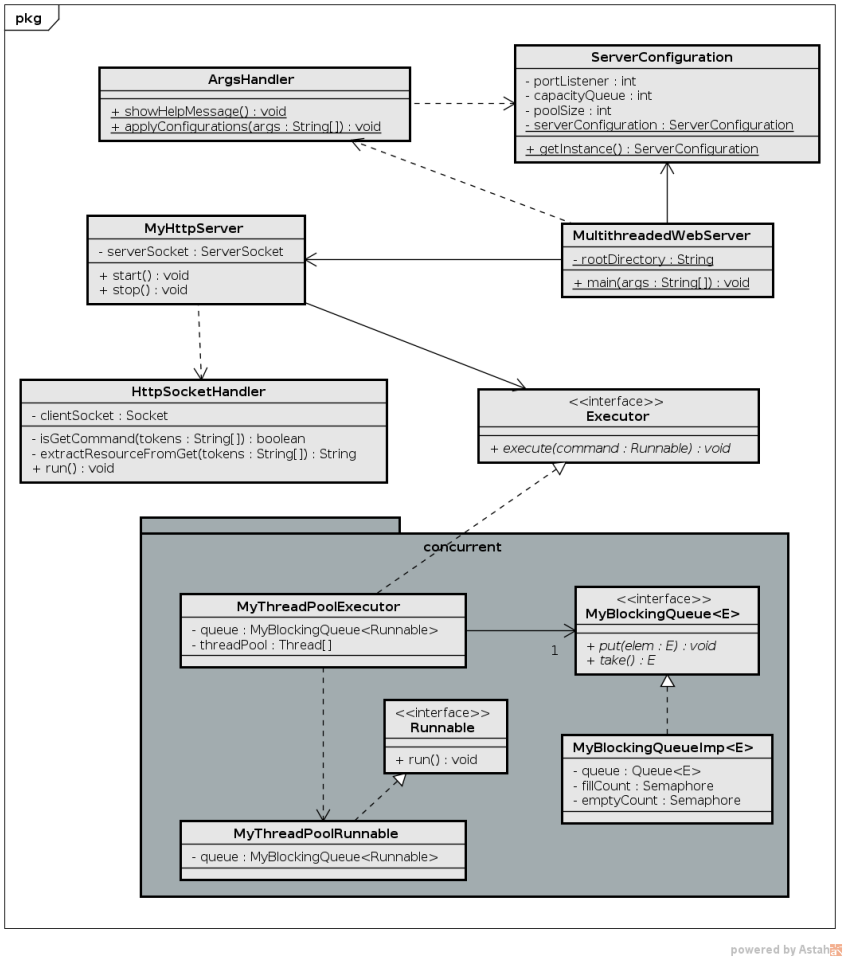

Figura 1. Diagrama de classes da implementação do servidor web multithread

4. Instruções para execução da aplicação

Nesta seção será informado como executar as aplicações do projeto, que estão disponíveis no GitHub, o link é https://github.com/CarlosPereira27/MultithreadedWebServer.

É recomendável a execução da aplicação em ambiente Linux, pois todos o experimento foi realizado neste ambiente.

4.1. Executando o servidor

O jar executável do projeto está no diretório dist e chama `multithreaded-web-server.jar`.

Para executar, acesse o diretório dist pelo terminal e então execute com o comando:

```
# java -jar multithreaded-web-server.jar -p <port> -c
<queue_capacity> -s <poll_size> -l
```

Todos parâmetros no comando acima são opcionais, por padrão, a aplicação será executada na porta 8008 com um pool de threads com 3 threads e uma fila de tarefas com a capacidade máxima de 10 tarefas. Para ajustar as configurações defina os parâmetros da aplicação, segue o que cada parâmetro define:

- `-p --port=PORT` Porta em que o servidor está ouvindo
- `-c --queue_capacity=CAPACITY` Capacidade máxima da fila de tarefas
- `-s --poll_size=SIZE` Tamanho do pool de threads
- `-l --log` Ativa o log das requisições
- `-h --help` Imprime mensagem de ajuda

Exemplos:

- `java -jar multithreaded-web-server.jar -p 9005` - A aplicação é executada na porta 9005 com um pool de 3 threads e uma fila de tarefas com a capacidade máxima de 10 tarefas.
- `java -jar multithreaded-web-server.jar -p 9005 -s 8 -c 20` - A aplicação é executada na porta 9005 com um pool de 8 threads e uma fila de tarefas com a capacidade máxima de 20 tarefas.

A aplicação estará disponível na porta especificada, basta acessar no seu navegador de preferência:

http://127.0.0.1:¡port¿/, substituindo ¡port¿ pela porta especificada.

4.2. Executando o teste de carga

Foi desenvolvida uma aplicação multithread em python para realizar teste de carga. O arquivo da aplicação é o `load_test.py` e está no diretório raiz do projeto.

Para executar o teste de carga execute:

```
# python3 load_test.py -r <request_file> -o <host> -p
<port> -n <qty_each_request> -c <num_clients>
```

onde,

- `-r --request_file=REQUEST_FILE` Arquivo com a lista de recursos a ser requisitado ao servidor web
- `-o --host=HOST` Host do servidor web. Padrão: http://localhost
- `-p --port=PORT` Porta em que o servidor web está ouvindo. Padrão: 8008
- `-n --qty_each_request=QTY_EACH_REQUEST` Quantidade de requisições para cada recurso. Padrão: 1
- `-c --num_clients=NUM_CLIENTS` Quantidade de clientes fazendo requisições. Padrão: 1
- `-h --help` Imprime mensagem de ajuda

5. Metodologia

Nesta seção é descrito a metodologia dos experimentos realizados. Identificando o ambiente onde os experimentos foram executados, as bases de dados utilizadas no experimento e o procedimento do experimento.

5.1. Ambiente do experimento

Para realização do experimento foi alocada duas máquinas virtuais na nuvem, atravéz da plataforma Google Cloud, ambas na região *us-east1-b*. Para executar o servidor web multithreads vou alocada uma máquina com um processador Intel Xeon de 24 núcleos e 24 GB de memória RAM, o sistema operacional utilizado foi a distribuição Linux Ubuntu 16.04.3 LTS x86_64 GNU/Linux (xenial) com o kernel do linux versão 4.13.0-1008-gcp. Para simular múltiplos clientes foi alocada uma máquina com um processador Intel Xeon de 8 núcleos e 8,5 GB de memória RAM, o sistema operacional utilizado foi a distribuição Linux Ubuntu 16.04.3 LTS x86_64 GNU/Linux (xenial) com o kernel do linux versão 4.13.0-1008-gcp.

5.2. Base de dados

Para a realização do experimento foi utilizada a base de dados *Syskill and Webert Web Page Ratings* do repositório [**?**]. Essa base de dados contém arquivos HTML de páginas web e também avaliações de um usuário nessas páginas. A base de dados está divida em quatro assuntos, que são: bandas, cabras, ovelha, e biomedicina.

A base de dados *Syskill and Webert Web Page Ratings* contém 345 arquivos, que são usados como recursos do servidor web, esses arquivos somam 1,5MB de dados.

5.3. Procedimento do experimento

Para elaboração do experimento foram definidos os seguintes números de threads para o servidor multithread: 1, 2, 3, 4, 6, 8, 12, 16, 20, 24, 32, 40, a capacidade da fila do servidor foi mantida em 1000, a quantidade de clientes simulados no cliente foram 8. Por fim, cada cenário realiza 100 vezes as requisições da base de dados definida, onde o primeiro cliente realiza 16 vezes as requisições de recursos da base de dados e os outros clientes 12 vezes. Para cada cenário foram executados dez vezes o experimento. O motivo de se executar dez vezes foi para ganhar confiança, caso em algum momento, algo atrapalhe o processamento da máquina. Utilizou-se dois cenários de números de threads maiores do que a capacidade da máquina, para analisar o quanto a troca de contexto de processos pode atrapalhar no processamento.

Após a definição desse procedimento criou-se um script para executar o experimento e salvar os tempos em uma planilha. O script, executado na máquina cliente, executa a aplicação do servidor web multithread com as diferentes configurações do experimento na máquina servidor através de ssh. Depois disso, o script executa a aplicação que simula os clientes fazendo requisições na máquina cliente e por fim gera um relatório.

6. Resultados

Na Tabela 1 está o relatório da média de tempo para as dez execuções do experimento para cada quantidade de thread, o intervalo de confiança dessa média para um grau de confiança de 95% , e também o *speedup* e a eficiência alcançada pelo processamento paralelo.

Tabela 1. Resultados do experimento

nthreads	tempo médio	intervalo de confiança	speedup	eficiência
1	72717321	[71828129 - 73606513]	1.000000	1.000000
2	38098933	[37749163 - 38448703]	1.908645	0.954322
3	26842818	[26620439 - 27065197]	2.709005	0.903002
4	23531019	[23242052 - 23819986]	3.090275	0.772569
6	21747735	[21425487 - 22069983]	3.343673	0.557279
8	21537061	[21239348 - 21834774]	3.376381	0.422048
10	21497078	[21146156 - 21848000]	3.382661	0.338266
12	21651260	[21424577 - 21877943]	3.358572	0.279881
16	22024614	[21686103 - 22363125]	3.301639	0.206352
20	22002722	[21460659 - 22544785]	3.304924	0.165246
24	22637821	[22350533 - 22925109]	3.212205	0.133842
32	23367986	[23162119 - 23573853]	3.111835	0.097245
40	23427009	[23143331 - 23710687]	3.103995	0.077600

Na Figura 2, o gráfico mostra o logaritmo do tempo médio gasto para realizar as requisições em relação ao aumento de threads no poll de threads do servidor. Nota-se que até 6 threads o tempo está diminuindo de forma mais significativa, com 8 e 10 threads o tempo abaixa muito pouco, já de 12 threads até 20 o tempo aumento bem pouco e depois disso ele aumenta um pouco mais.

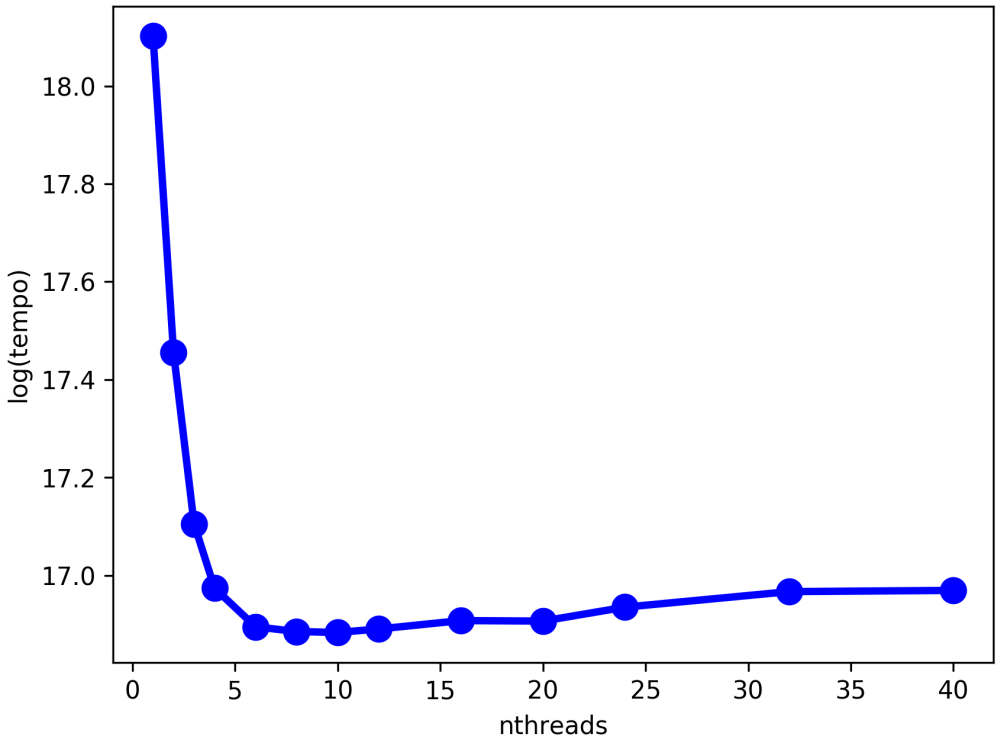

Figura 2. Logaritmo do tempo médio de execução

O gráfico, na Figura 3, mostra o *speedup* do servidor multithread ao aumentar a quantidade de threads no poll de threads. Observa-se que o *speedup* aumenta até 10 threads, apesar de nos dois últimos pontos aumentar bem pouco. Portanto, o speedup atinge seu melhor valor com 10 threads e depois disso ele só vai diminuindo.

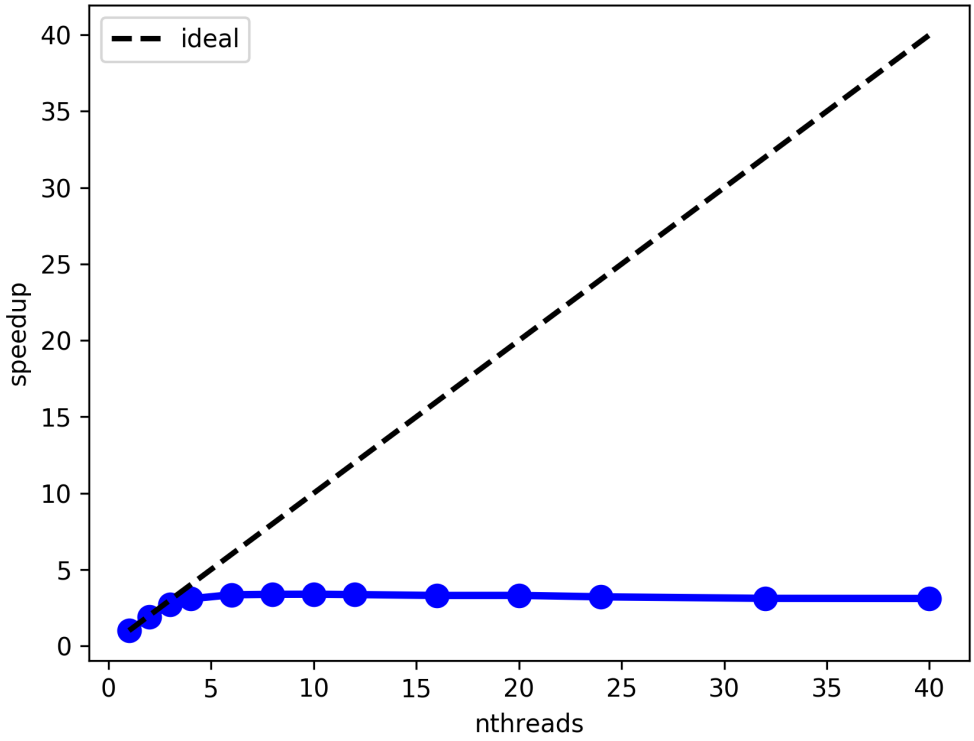

Figura 3. *Speedup* do servidor multithread

O gráfico, na Figura 4, mostra a eficiência do servidor multithread ao aumentar a quantidade de threads no poll de threads. Nota-se que a eficiência aumenta sua queda a partir de 4 threads, chegando a uma eficiência extremamente baixa com 40 threads.

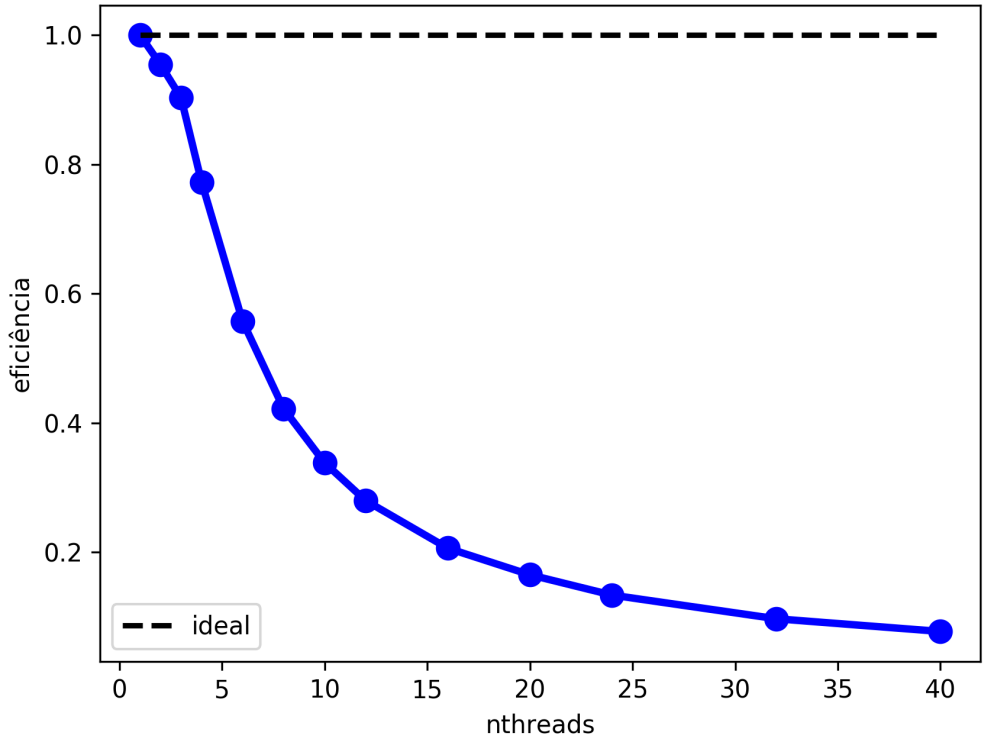

Figura 4. Eficiência do servidor multithread

7. Conclusão e trabalhos futuros

Por fim, conclui-se que o servidor multithreads têm uma melhora de desempenho até 6 threads, com mais de 6 threads o desempenho do servidor fica estagnado, chegando ao ponto de que com mais de 24 threads fique pior. No entanto, a perda de desempenho por ter mais threads do que a capacidade da máquina não causa uma perda tão significativa, visto que a troca de contexto de threads é mais eficiente do que de processos.

Uma das causas de o desempenho não melhorar após 6 threads é que o experimento simula apenas 8 clientes, provavelmente se simulasse mais clientes, poderia ter uma melhora de desempenho acima de 6 threads.

Referências

Flanagan, D. (2011). *JavaScript: The Definitive Guide Activate Your Web Pages*. O'Reilly Media, Inc., 6th edition.

Kemp, S. (2017). The incredible growth of the internet over the past five years – explained in detail. https://thenextweb.com/insider/2017/03/06/the-incredible-growth-of-the-internet-over-the-past-five-years-explained-in-detail/. Acesso em: 18/01/2018.

Rouse, M. (2015). Web server. http://whatis.techtarget.com/definition/Web-server. Acesso em: 18/01/2018.

Rouse, M. (2017). World wide web (www). http://whatis.techtarget.com/definition/World-Wide-Web. Acesso em: 18/01/2018.