PUCRS

ESCOLA POLITÉCNICA
PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO
MESTRADO EM CIÊNCIA DA COMPUTAÇÃO

CARLOS ALBERTO FRANCO MARON

PARAMETRIZAÇÃO DO PARALELISMO DE STREAM EM BENCHMARKS DA SUÍTE PARSEC

Porto Alegre
2018

PÓS-GRADUAÇÃO - *STRICTO SENSU*

Pontifícia Universidade Católica
do Rio Grande do Sul

PONTIFÍCIA UNIVERSIDADE CATÓLICA DO RIO GRANDE DO SUL
ESCOLA POLITÉCNICA
PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

PARAMETRIZAÇÃO DO PARALELISMO DE STREAM EM BENCHMARKS DA SUÍTE PARSEC

CARLOS ALBERTO FRANCO MARON

Dissertação apresentada como requisito parcial à obtenção do grau de Mestre em Ciência da Computação na Pontifícia Universidade Católica do Rio Grande do Sul.

Orientador: Prof. Dr. Luiz Gustavo Fernandes
Co-Orientador: Prof. Dr. Dalvan Griebler

Porto Alegre
2018

Ficha Catalográfica

F825p Franco Maron, Carlos Alberto

Parametrização do Paralelismo de Stream em Benchmarks da Suíte PARSEC / Carlos Alberto Franco Maron . – 2018.
95.
Dissertação (Mestrado) – Programa de Pós-Graduação em Ciência da Computação, PUCRS.

Orientador: Prof. Dr. Luiz Gustavo Leão Fernandes.
Co-orientador: Prof. Dr. Dalvan Griebler.

1. Pragramação Paralela. 2. Paralelismo de Stream. 3. PARSEC. I. Leão Fernandes, Luiz Gustavo. II. Griebler, Dalvan. III. Título.

Elaborada pelo Sistema de Geração Automática de Ficha Catalográfica da PUCRS
com os dados fornecidos pelo(a) autor(a).
Bibliotecária responsável: Salete Maria Sartori CRB-10/1363

Carlos Alberto Franco Maron

Parametrização do Paralelismo de Stream em Benchmarks da Suíte PARSEC

Dissertação apresentada como requisito parcial para obtenção do grau de Mestre em Ciência da Computação do Programa de Pós-Graduação em Ciencia da Computação, Escola Politécnica da Pontifícia Universidade Católica do Rio Grande do Sul.

Aprovado em 28 de Agosto de 2018.

BANCA EXAMINADORA:

Prof. Dr. Claudio Fernando Resin Geyer (UFRGS)

Prof. Dr. Tiago Coelho Ferreto (PPGCC/PUCRS)

Prof. Dr. Luiz Gustavo Leão Fernandes (PPGCC/PUCRS – Orientador)

Prof. Dr. Dalvan Jair Griebler (PNPD/PUCRS – Co-Orientador)

DEDICATÓRIA

Dedico esta dissertação aos meus pais, Aldair e Jussara.

“Excelsior!”
(Stan Lee)

AGRADECIMENTOS

Sou grato à Deus por me amparar em todas as situações, estar sempre ao meu lado em minha vida e por tornar tudo isso possível.

Aos meus pais, por todo o suporte, amor, compreensão, carinho, preocupação. Sempre acreditaram em meus sonhos e objetivos, dando o maior apoio possível.

Agradeço à minha namorada, por sua compreensão e seu amor que foram importantes para manter-me fortalecido e motivado em momentos difíceis dessa trajetória.

Meus orientadores Luiz Gustavo Fernandes e Dalvan Griebler, sou grato por todo conhecimento que a mim foi dado e por acreditarem em meu potencial.

Aos colegas de laboratório, pelo conhecimento técnico e teórico que adquiri. Pela paciência, correções e revisões de textos e códigos, traduções, discussões enriquecedoras, meu muito obrigado.

Aos meus amigos da PUC, agradeço pelo companheirismo durante as extensas horas de estudos e trabalho, pelas conversas e trocas de ideias.

Ao Laboratório de Pesquisas Avançadas para Computação em Nuvem (LARCC) e o Projeto HiPerfCloud pelo suporte financeiro.

PARAMETRIZAÇÃO DO PARALELISMO DE STREAM EM BENCHMARKS DA SUÍTE PARSEC

RESUMO

Para o projetista de software paralelo, é importante entender os impactos causados no desempenho a fim de entregar um software escalável e eficiente. Esses impactos podem ser essencialmente causados pelas características comportamentais das aplicações paralelas. Diante disso, elas podem ser representadas em *benchmarks*, uma vez que eles permitem avaliar e entender as suas características de desempenho. Além disso, a literatura indica que aplicações paralelas do mesmo domínio apresentam comportamentos similares. No entanto, os tradicionais *benchmarks* pouco exploram a parametrização das características relativas ao domínio da aplicação (ex. PARSEC). O desafio deste trabalho é permitir a parametrização das características de aplicações do domínio de processamento paralelo de *stream* (ou *stream parallelism* como é conhecido em inglês). Por isso, foram escolhidas duas aplicações (Dedup e Ferret) representativas deste domínio da suite PARSEC. O objetivo é identificar as características do paralelismo de *stream* e implementar o suporte à parametrização de tais características. A partir dos experimentos realizados, constatou-se que a possibilidade de parametrizar a novas características do paralelismo de *stream* implementadas, impactaram significativamente no desempenho dessas aplicações. Na maioria dos casos, a parametrização melhora o *throughput*, a latência, o *service time* e o tempo de execução. Além disso, uma vez que não foram avaliados o desempenho da arquitetura e dos *frameworks* de programação paralela, os resultados obtidos na presente pesquisa motivam novas investigações para compreender outros padrões comportamentais causados pela parametrização das aplicações.

Palavras-Chave: Programação Paralela, Paralelismo de *Stream*, PARSEC.

PARAMETRIZAÇÃO DO PARALELISMO DE STREAM EM BENCHMARKS DA SUÍTE PARSEC

ABSTRACT

The parallel software designer aims to deliver efficient and scalable applications. This can be done by understanding the performance impacts of the application's characteristics. Parallel applications of the same domain use to present similar patterns of behavior and characteristics. One way to go for understanding and evaluating the applications' characteristics is using parametrizable benchmarks, which enables users to play with the important characteristics when running the benchmark. However, the parametrization technique must be better exploited in the available benchmarks, especially on stream processing application domain. Our challenge is to enable the parametrization of the stream processing applications' characteristics (also known as stream parallelism) through benchmarks. Mainly because this application domain is widely used and the benchmarks available for it usually do not support the evaluation of important characteristics from this domain (e.g., PARSEC). Therefore, the goal is to identify the stream parallelism characteristics present in the PARSEC benchmarks and implement the parametrization support for ready to use. We selected the Dedup and Ferret applications, which represent the stream parallelism domain. In the experimental results, we observed that our implemented parametrization has caused performance impacts in this application domain. In the most cases, our parametrization improved the throughput, latency, service time, and execution time. Moreover, since we have not evaluated the computer architectures and parallel programming frameworks' performance, the results have shown new potential research investigations to understand other patterns of behavior caused by the parametrization.

Keywords: Parallel Programming, Stream Parallelism, PARSEC.

LISTA DE FIGURAS

LISTA DE TABELAS

LISTA DE ABREVIATURAS

BSW. – *Basic Slidwing Window*

TTTD. – *Two Thresholds, Two Divisors Algorithm*

SRM. – *Statical Region Merging*

LSH. – *Locality Sensitive Hashing*

EMD. – *Earth Mover's Distance*

FR. – *Fragment Refine*

DD. – *Deduplication*

COMP. – *Compression*

RR. – *Reorder*

DD. – *Deduplication*

SEG. – *Segmentation*

EXT. – *Extraction*

IDX. – *Indexing*

RANK. – *Ranking*

NAS-NPB. – *NASA Parallel Benchmark*

IoT. – *Internet of Things*

SUMÁRIO

1. INTRODUÇÃO

Benchmarks são importantes ferramentas para diversas áreas da computação. Tanto na indústria quanto na academia, eles são utilizados para avaliar e comparar o desempenho de sistemas ou componentes [AHL+15], pois tratam-se de programas sintéticos com características de programas reais. Desse modo, os *benchmarks* conseguem representar os comportamentos das aplicações reais [BLM+09].

Na indústria, os *benchmarks* são úteis em projetos de novos sistemas, dado que seus resultados antecipam comportamentos dos sistemas, antes mesmo de serem colocados em produção [IPE14]. Na academia, os pesquisadores avançam em suas pesquisas com o apoio dos *benchmarks* [SEH03]. Sendo assim, os *benchmarks* contribuem para o surgimento e melhoria de softwares e hardwares. Em função disso, uma grande quantidade de *benchmarks* é criada.

No domínio de *benchmarks* existem diferenças entre os *microbenchmarks* e os *benchmarks*. *Microbenchmark* tem por finalidade testar apenas uma parte do sistema, pois não considera algumas características. Por isso, seu comportamento não é representativo para o sistema completo [BLM+09]. Um exemplo de *microbenchmark* é o IPERF[1], que avalia somente a rede. Já o *benchmark* (*benchmark* de programas, *benchmark* de aplicação, ou *benchmark* sintético) é desenvolvido com traços de aplicações reais, combinando diferentes funções que exigem uma variedade de operações do sistema [Jai90, BLM+09]. Isso permite que os *benchmarks* representem diversos domínios de aplicações, tornando-se mais complexos que os *microbenchmarks*. Um *benchmark* pode ser formado por funções de E/S em disco ou rede, funções com cálculos intensivos que exigem processamento e acesso à memória. Um exemplo de *benchmark* é o LINPACK [DLP03], que avalia a capacidade de supercomputadores. O LINPACK realiza cálculos de equações lineares e pontos flutuantes.

O desempenho de um *benchmark* é um fator relativo [BLM+09]. Inicialmente, um dos fatores é causado pela arquitetura de processamento, pois ela oferece o ambiente necessário para a execução, e dependendo das tecnologias deste ambiente, a execução pode ser mais rápida ou mais lenta. Isso faz com que alguns *benchmarks* sejam utilizados para avaliação do desempenho das arquiteturas. Por outro lado, independentemente da arquitetura, as funções que formam um *benchmark* podem ter comportamentos diferenciados durante a execução, por isso, o desempenho do *benchmark* se torna relativo. Para representar uma aplicação real, o *benchmark* utiliza funções complexas, que podem depender de outros conjuntos de funções e valores. Tal dependência indica que as aplicações também precisam ser avaliadas, pois são formadas por diversas características, que podem: ter diferentes propósitos, executar diferentes instruções, ter inúmeras variáveis de código,

[1] https://iperf.fr/

processar diferentes tipos e tamanho de dados, ser implementadas em diferentes linguagens de programação, *etc*. Ter o entendimento de todas as características que impactam no desempenho pode ser complexo e exigir muito esforço. Por isso, algumas pesquisas sugerem *benchmarks* onde as características das aplicações são parametrizáveis, pois tornam o entendimento das aplicações menos complexo e conseguem oferecer informações mais precisas sobre seus comportamentos [BLM+09].

Em sistemas computacionais e na comunidade científica, é cada vez maior o destaque do domínio de processamento de *stream* [TKA02, AGT14, HSS+14, GFDF18, Gri16], os quais são formados por aplicações de processamento de sinais, banco de dados, imagens, *Internt of Things* (IoT), e outras. Em particular, as aplicações desse domínio de *stream* possuem algumas características como: janela deslizante, canais de comunicação, *buffers*, fluxo de *stream*, entre outras [HSS+14]. Um *benchmark* deste domínio de processamento, deve considerar tais características e, assim, conseguir representar uma aplicação real, tanto paralela quanto sequencial.

Independente da arquitetura, as características do domínio de *stream* têm comportamentos específicos. Tornar ajustáveis as características, pode oferecer informações mais precisas sobre a execução da aplicação real e prever comportamentos sobre diferentes condições. O Eigenbench [HOC+10] é um *microbenchmark* que consegue avaliar o comportamento da aplicação por meio de parâmetros e é um *microbenchmark* do domínio de aplicações de memória transacional. Embora Eigenbench seja para outro domínio de processamento, ele é um exemplo de um *microbenchmark* que consegue avaliar o comportamento de uma aplicação de memória transacional por meio de parâmetros.

A suíte PARSEC é composta por *benchmarks* que representam diferentes domínios de aplicações e tipos de paralelismo, como paralelismo de *stream*, de dados e de tarefas. Porém, a suíte PARSEC não é suficiente para compreender as características do paralelismo de *stream*, pois o cenário de parametrização nos *benchmarks* é limitado. Além disso, os 13 *benchmarks* são caracterizados com um interesse particular em avaliar aspectos das arquiteturas paralelas com processadores *multicores*. Suas aplicações foram implementadas utilizando diferentes modelos paralelos e com os principais *frameworks* de programação paralela, para compreender as características das arquiteturas, que são: paralelização, localidade e conjunto de trabalho, taxa de comunicação para computação, compartilhamento e tráfego *off-chip* [Bie11].

Mesmo que a suíte PARSEC tenha se tornado o estado da arte para pesquisas que envolvem o cenário de arquiteturas de processadores [BM09] e em propostas de novas suítes de *benchmarks* [CBM+09], o PARSEC possui um conjunto reduzido de alternativas para explorar os comportamentos das aplicações. Uma vez que o objetivo é avaliar a arquitetura, os *benchmarks* oferecem poucos parâmetros relacionados às aplicações. Além disso, ainda que pesquisadores implementem as aplicações da suíte com novas estratégias de paralelismo e diferentes *frameworks* de programação paralela [CM08, GFDF18],

estes estudos não são suficientes para compreender os comportamentos do paralelismo de *stream*.

Visto que as aplicações do domínio de *stream* possuem características que não são influenciadas pelas arquiteturas, um *benchmark* parametrizável oferece a possibilidade de explorar diferentes cenários de execuções e auxilia na compreensão de detalhes sobre elas. Ademais, compreender o comportamento de uma aplicação de *stream* específica, contribui para que as outras do mesmo domínio também o sejam. Um exemplo é o Dedup, uma aplicação que pode representar o comportamento de outras aplicações do domínio de *stream*, como: aplicações de controle de versão [Tic85, HVT98], sincronização remota [SHWH12], sistema de backup [BL97], algoritmos de compressão LZ77 [ZL78], *zdelta* [SM02] e *vcdiff* [HVT98].

Um *benchmark* parametrizável também contribui para gerar modelos analíticos de desempenhos mais exatos. Um modelo de desempenho é gerado por meio de resultados das execuções reais de *benchmarks*. Gerar um comportamento diferente durante as execuções dos *benchmarks*, possibilita economia de tempo durante a geração do modelo de desempenho. Tarvo [TR18] demonstra a complexidade de encontrar um modelo de desempenho para aplicações paralelas *multithreads*. Ele afirma que um modelo de desempenho que abrange toda a aplicação pode demorar muito mais tempo para ser gerado que a própria execução da aplicação.

Portanto, nesta dissertação é investigado um cenário não abordado pelos *benchmarks* que estão disponíveis na computação. O objetivo é mostrar que, por meio das parametrizações das características do domínio de *stream* nos *benchmarks*, os comportamentos das aplicações paralelas desse domínio são influenciados. Assim, para tornar isto possível, é proposto neste trabalho versões parametrizáveis do Dedup e do Ferret[2]. Em seguida, na Seção 1.1, descrevemos as principais contribuições deste trabalho.

1.1 Contribuições

O principal objetivo desta dissertação é compreender como as características do processamento paralelo de *stream* influenciam os comportamentos das aplicações deste domínio. Na pesquisa de Hirzel [HSS+14], é evidenciada a importância de gerar novas discussões sobre as aplicações de *stream* e novas ferramentas para avaliar seus desempenhos. Desse modo, foram encontrados diversos *benchmarks* disponíveis para diferentes finalidades na computação, porém, não se tratam de *benchmarks* com características parametrizáveis do processamento de *stream*, pois são utilizados na comparação do desempenho de arquiteturas ou para compreender as características das arquiteturas. Além disso, são *benchmarks* que representam outros domínios de aplicações e também são para

[2]Dedup e Ferret são os *benchmarks* da suíte PARSEC que representam aplicações reais.

ambientes de processamento distribuído. Em vista disso, as principais contribuições para a presente dissertação são as seguintes:

- **Identificação das características do paralelismo de *stream* que influenciam nos comportamentos de *benchmarks* da suíte PARSEC.** Embora existam trabalhos indicando que as aplicações Dedup e Ferret podem ser otimizadas para alcançar melhores resultados de desempenho [NATC09, CCM+15, DSDMT+17], eles se utilizam de diferentes *frameworks* de programação paralela ou estratégias de paralelismo. O estudo nessas pesquisas não é suficiente para propor novos parâmetros ajustáveis nas aplicações. Por isso, nesta dissertação o estudo das aplicações se fez necessário para investigar as características do paralelismo de *stream*. Esta pesquisa realizada se destaca pois a investigação vai além dos *frameworks*; isso tornou possível a identificação de possíveis características que influenciam nos comportamentos das aplicações. Tal contribuição é detalhada no Capítulo 4, onde serão apontados os passos da investigação. Além disso, como parte das investigações dos comportamentos, serão demonstrados os resultados das execuções e as discussões dos comportamentos no Capítulo 5.

- **Suporte à parametrização das características do paralelismo de *stream* em *benchmarks* da suíte PARSEC.** Dedup e Ferret, que fazem parte da suíte, representam o domínio de processamento paralelo de *stream*. Desse modo, é desenvolvido nesses *benchmarks* o suporte para a parametrização das características do processamento paralelo de *stream* estudadas neste trabalho. As parametrizações vão além do que é oferecido pela suíte, como o número de *threads* ou o tamanho de entrada. As parametrizações implementadas nesta pesquisa suportam os tamanhos dos *buffers*, tamanhos das filas, tamanhos dos elementos do *stream* e janela deslizante. Novos tipos e tamanhos diferentes de entradas também são suportados. Outrossim, as métricas de *throughput*, latência e *service time*, que são do processamento de *stream*, também são disponibilizados. No Capítulo 4, será detalhada a mencionada contribuição.

- **Uma análise de desempenho dos efeitos das parametrizações em *benchmarks* da suíte PARSEC baseadas em um conjunto de experimentos.** Esta dissertação não avalia o desempenho das arquiteturas e dos *frameworks*. Os resultados demonstram os comportamentos dos *benchmarks* avaliados com as características parametrizáveis do domínio paralelo de *stream*. Além disso, é discutido detalhadamente como as características impactam nos comportamentos de tempo de execução, *throughput*, latência e *service time*. Os monitoramentos realizados mostram que, parametrizando as características, diferentes comportamentos são gerados nos *benchmarks*. Na maioria dos casos, tempo de execução, *throughput*, latência e *service time* apresentam melhorias. Ressalta-se a viabilidade de os resultados desta disserta-

ção servirem como incentivo para novas investigações de possíveis parametrizações nas aplicações. Os mesmos estão demonstrados no Capítulo 5.

1.2 Estrutura da Dissertação

A dissertação é introduzida e endossa suas motivações no Capítulo 1. Já as aplicações de *stream* e o paralelismo destas aplicações são contextualizados no Capítulo 2. Os trabalhos relacionados estão indicados no Capítulo 3. No Capítulo 4 descreve-se o início da investigação das características do paralelismo de *stream*. Além disso, no mesmo capítulo se esclarecem os passos para as implementações das versões parametrizáveis e os contrastes existentes das versões originais do PARSEC com as versões parametrizáveis implementadas. No Capítulo 5, destacam-se os resultados alcançados com as execuções realizadas, ainda assim, uma visão geral destes resultados também é discutida. O Capítulo 5 faz parte da investigação das características do paralelismo de *stream*. Por fim, a conclusão do trabalho é realizada no Capítulo 6. Além disso, são discutidos os próximos passos que envolvem esta pesquisa.

2. CONTEXTUALIZAÇÃO

Neste capítulo são contextualizadas as aplicações do domínio de *stream*. Na Seção 2.1, a discussão se volta a aplicações reais e suas características. Em seguida, na Seção 2.2, alguns dos principais desafios que envolvem a paralelização das aplicações de *stream* são analisados. Esses desafios fazem parte do contexto desta dissertação. As informações do presente capítulo trazem um embasamento teórico para auxiliar na compreensão dos subsequentes (mais propriamente, Capítulo 4, Capítulo 5 e Capítulo 6).

2.1 Domínio de Aplicações de *Stream*

O processamento de vídeos, imagens, dados de Internet das Coisas (IoT) e de grandes volumes de dados representam algumas das aplicações de *stream* que estão presentes no cotidiano das pessoas. Essas aplicações são processadas em diferentes arquiteturas, que vão desde *data center* até dispositivos móveis. Pelo contexto da aplicação, existem características particulares e comportamentos que definem esse domínio de aplicação [Thi10, AGT14, Gri16]. O *stream* processado pelas aplicações é formado por um fluxo contínuo de elementos, que são dados, instruções ou tarefas. Além disso, os elementos podem ser quaisquer tipos de dados computacionais, como texto, imagem, vídeo, estrutura, *etc*. Os elementos do *stream* seguem uma sequência ordenada de operações que computam cada um dos elementos. Ou seja, as operações são organizadas e formam a aplicação, assim, cada operação consome uma entrada e produz uma saída. Tal comportamento assemelha-se a uma linha de produção automobilística, onde os carros precisam passar por processos industriais até serem finalizados. Nessa analogia, na aplicação de *stream*, o carro é o elemento e os processos industriais são as operações (computações).

Normalmente, o *stream* não possui um fim estabelecido, por isso estas aplicações executam infinitamente. No entanto, também existem aplicações onde o final do *stream* é conhecido ou previsto. Por esses motivos, na maioria das vezes o processamento do *stream* acontece em tempo real, com a possibilidade deste *stream* ser irregular, com oscilações na taxa de entrada, diferentes tipos de elementos e diferentes origens (entradas).

Aplicações de processamento multimídia como vídeos, imagens e áudios são populares no domínio de *stream* e estão mais próximas tanto do usuário comum quanto do avançado. A aplicação VLC *media player*[1] é utilizada pelo usuário comum para reproduzir conteúdo multimídia. O VLC é capaz de processar a maioria dos codificadores, tais como: MPEG-2, MPEG-4, H.264, MP3 e outros. Além disso, o VLC é compatível para todas as plataformas de sistemas operacionais, inclusive para dispositivos móveis. Também é uma

[1] http://www.videolan.org/vlc/

aplicação que consegue processar tanto o *stream* infinito quanto o *stream* com final conhecido/previsto. Por exemplo, ao reproduzir um arquivo de vídeo ou áudio, o *stream* terá o tamanho do arquivo reproduzido pelo VLC. Por outro lado, um *stream* infinito, será o vídeo capturado por uma câmera. Isso indica que o VLC também processa *stream* de diferentes origens, desde o arquivo em disco até uma câmera. VLC é uma aplicação de *stream* simples e que consegue representar com clareza as principais características do domínio de processamento de *stream*.

Aplicações de *stream* estão presentes em áreas importantes, como a da saúde. Nesse cenário, as aplicações processam grandes volumes de dados, ajudando na prevenção e planejamento de medidas contra epidemias, bem como na qualidade do monitoramento de exames [NBSV13]. Na área da inteligência artificial, o processamento de *stream* também é encontrado nas aplicações. Dentre elas, YOLO [RDGF16] é uma aplicação de *Deep Learning* que detecta e classifica objetos em imagens, determinando a localização dos mesmos e como eles interagem entre si. Por meio de uma única rede neural durante a execução do YOLO, uma imagem é dividida em segmentos de uma matriz *n X n*, enquanto elementos probabilísticos são delegados a cada região segmentada. Assim, as identificações dos objetos ocorrem de acordo com as somas dessas probabilidades. Esse tipo de processamento permite que uma máquina realize tarefas extremamente complexas, como dirigir sem muito pensamento consciente, como ocorre nos carros autônomos operados por computadores. Tais aplicações processam eficientemente sem a necessidades de sensores especializados, melhorando a interação entre o homem e a máquina.

As aplicações de *stream* estão se tornando recorrentes em muitas áreas, sendo utilizadas com o intuito de processar diferentes tipos de informações para obter diversos resultados. Essas aplicações têm exigido o processamento paralelo, por isso diversos desafios são encontrados ao paralelizar o processamento de *stream*. Portanto, na Seção 2.2, são demonstradas algumas das características do processamento paralelo nas referidas aplicações e quais seus principais desafios.

2.2 Paralelismo de *Stream*

Nas aplicações de *stream*, as principais características são o fluxo contínuo do *stream* e o processamento próximo ao tempo real dos elementos. Essas características impõem alguns desafios ao paralelizar aplicações desse domínio. Em certos cenários, tais aplicações enfrentam variações imprevisíveis nas taxas que os elementos do *stream* chegam para serem processados. Assim, paralelizar essas aplicações para melhorar o desempenho deve levar em conta *throughput* e latência. Desse modo, entender estes comportamentos para que alguma estratégia de paralelismo seja tomada, exige do programador um conhecimento aprofundado do cenário envolvido. O programador deve conhecer a es-

Figura 2.1 – Aplicações do domínio de *stream*. Adaptado de [AGT14].

trutura da aplicação com detalhes e os tipos de dados que serão processados, a arquitetura que será utilizada, a linguagem e o *framework* para o desenvolvimento (compatíveis com o cenário de aplicação de *stream*), entre outros.

A paralelização das aplicações de *stream* é um tema recorrente em diversas pesquisas. Em algumas delas, *frameworks* foram desenvolvidos para ajudar os programadores a paralelizar essas aplicações. Para ambientes de memória compartilhada, em destaque estão os *frameworks* FastFlow [ADKT17] e SPar [GF17]. *Frameworks* para processamento paralelo distribuído também são utilizados, porém, detalhá-los não faz parte do contexto desta dissertação. Na paralelização, utilizar somente o *framework* não resolve todos os desafios que as aplicações impõem. O programador precisa conhecer os detalhes das aplicações para garantir o processamento eficiente, o desempenho e a integridade dos resultados do processamento. Os referidos desafios das aplicações são discutidos em seguida.

Normalmente, uma aplicação paralela de *stream* é implementada com o modelo *pipeline*. Cada estágio do *pipeline* é processado por uma *thread* e, em uma arquitetura *multicore*, as *threads* são alocadas nos núcleos do processador. Esse é o modelo básico para a implementação do paralelismo nas aplicações de *stream* e pode ser observado na Figura 2.2. Embora o *pipeline* seja o modelo clássico, o *Farm* também é uma estratégia que pode ser utilizada pelo programador (desmonstrado na Figura 2.2).

No entanto, paralelizar uma aplicação de *stream* com o modelo *farm* exige da aplicação alguns requisitos. Inicialmente, é necessário compreender que no *farm* os estágios são replicados. Se o estágio pode ser executado em paralelo, são criadas as réplicas, que também são executadas por *threads*. Todavia, para replicar um estágio, os elementos do *stream* não podem conter dependências entre si. Por exemplo, na Figura 2.2, os elementos processados pelo estágio dois do *farm* não têm dependências com os elementos

processados no mesmo instante pelas outras réplicas. Isso garante que os elementos sejam processados por réplicas distintas. No paralelismo de *stream*, estágios paralelos são chamados de *stateless*, pois os elementos não possuem dependência. Já estágios *statefull*, são os que não podem ser paralelizados.

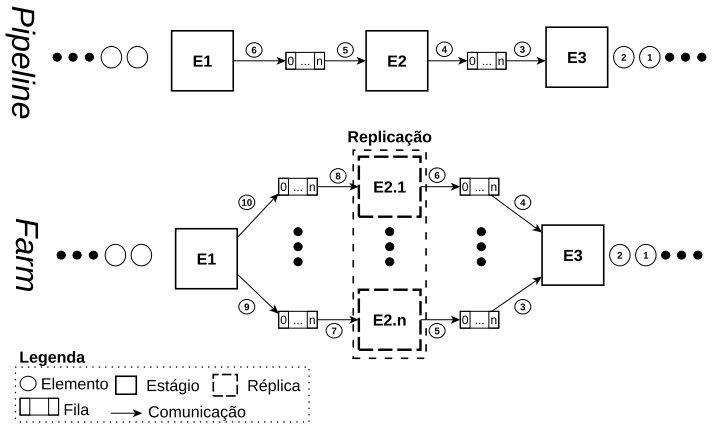

Figura 2.2 – Modelos de paralelismo em aplicações de *stream*. Os modelos demonstrados são o *pipeline* e o *farm*. As caixas indicam os estágios que realizam as operações. Os objetos circulares indicam os elementos do *stream* e estão identificados por uma ordem numérica. As setas indicam a comunicação ordenada. No *farm*, a região em destaque do estágio 2 representa o estágio *stateless*, o estágio que pode ser replicado. Entre os estágios de ambos os modelos existem as filas com tamanho de 1 até *n*.

FastFlow e SPar são *frameworks* que abstraem o paralelismo das aplicações de *stream* e ajudam na produtividade do programador, pois diversos mecanismos são oferecidos implicitamente ao programador que utiliza esses *frameworks*. No FastFlow e na SPar, as filas entre os estágios não precisam ser implementadas pelo programador.

POSIX *threads* também é utilizado na paralelização de aplicações de *stream*. Porém, ao contrário do FastFlow e SPar, *pthreads* demanda um esforço maior do programador, pois exige a implementação de alguns mecanismos, como filas entre os estágios, *locks*, entre outros. Implementar tais mecanismos exige conhecimentos avançados de programação paralela. Sendo assim, a discussão no decorrer dos capítulos desta dissertação traz pontos que ajudam na compreensão destes detalhes nas aplicações paralelas de *stream*. Além disso, minuciosamente são discutidos os comportamentos das aplicação e como as características delas, descritas anteriormente, impactam no comportamento.

2.3 Métricas de Desempenho

Na computação paralela clássica, as principais métricas de desempenho podem ser o tempo de execução, *speed-up* e eficiência. No entanto, nas aplicações paralelas de *stream*, o cenário é um pouco diferente e, nesse caso, as métricas mais utilizadas são a latência, *throughput* e *Service Time*. Como existem aplicações de *stream* que nunca

terminam, as métricas tradicionais de tempo de execução, *speed-up* e eficiência não podem ser obtidas.

$$Throughput = \frac{Trabalho\ em\ processo}{Ciclo\ de\ tempo} \tag{2.1}$$

Na Equação 2.1 é demonstrada a fórmula para o cálculo de *throughput*. Essa métrica de desempenho indica a quantidade de trabalhos realizados em uma determinada janela de tempo. No contexto do processamento de *stream*, o *throughput* pode ser obtido em tempo de execução com base em um intervalo mínimo de tempo. Esse cenário é necessário para obter o *throughput* quando as aplicações têm o comportamento infinito de processamento. O objetivo com essa métrica é alcançar o maior valor possível, pois quanto maior o *throughput*, melhor é o desempenho da aplicação. Por isso essa métrica é utilizada em métodos que adaptam automaticamente o grau de paralelismo em aplicações de *stream*.

Embora a métrica de latência exista em outros domínios de aplicações (ex: rede), no processamento de *stream* ela indica o custo de comunicação dos elementos entre os estágios da aplicação. Na maioria dos casos, a latência significa o tempo que um elemento aguardou nas filas até ser retirado para o processamento. O objetivo com essa métrica é obter o menor valor possível, pois valores altos podem indicar problemas de desempenho na aplicação de *stream*. A latência também pode ser utilizada em métodos de adaptação do grau de paralelismo.

3. TRABALHOS RELACIONADOS

Neste capítulo são apresentados alguns trabalhos relacionados e que estão organizados em seções. Os trabalhos que envolvem a avaliação de desempenho utilizando benchmarks e técnicas de modelagem analítica [Tar14, TR18] se encontram na Seção 3.1. Os trabalhos que abrangem diferentes domínios de aplicação, como Big Data [LWXH14], mineração de dados [GQS10] e memórias transacionais [HOC+10] estão na Seção 3.2. Na Seção 3.3, está o trabalho que propõe melhorias em suíte de avaliação de desempenho [MS17]. Ao final, na Seção 3.4, a Tabela 3.2 mostra uma visão geral dos trabalhos relacionados.

3.1 Técnicas de Modelagem Analítica

Tarvo [Tar14, TR18] apresenta um *framework* para modelagem automática de desempenho de programas *multithreads*. O *framework* combina métodos de análise estática e dinâmica para construir um modelo de desempenho do programa. O modelo é gerado a partir de informações extraídas de uma execução do programa. Essas informações são a sua estrutura, semânticas de interação entre *threads* e demanda de recursos do componentes individuais que compõem o programa.

O trabalho mostra as dificuldades encontradas para criar um modelo de predição para aplicações desse cenário. Muitos comportamentos são complicados de avaliar, como a sincronização de *threads* ou o acesso das *threads* aos dados compartilhados. O mesmo ocorre com o uso compartilhado de recursos pelas *threads*, como CPU, disco, memória.

Aplicações *multithreads* são extensas e complexas. Um modelo de desempenho que busca cobrir toda a aplicação pode ser complicado e sua execução levar mais tempo que a aplicação real. Por isso, os autores implementam sua abordagem sobre o PERSIK - *Performance Simulation Kit*. PERSIK foi utilizado para construir modelos de programas Java *multithreads*, que foram submetidos a cargas de trabalho ligadas a CPU e I/O e executadas em diferentes infraestruturas de 4 e 16 *threads*. Para todos os casos, os modelos gerados foram construídos a partir de uma única execução do programa, onde a predição de desempenho variou configurações do tipo: número de *threads*, cores da CPU e intensidade da carga de trabalho. Esses modelos incluíram métricas de desempenho como *throughput*, tempo de resposta, utilização de recursos de CPU e armazenamento.

Foram modeladas aplicações pequenas e médias dos domínios de aplicações científicas, financeiras, multimídia e *web servers*, que variam de 1006 a 3207 linhas de código (LOC). Modelos de grandes aplicações industriais, como Apache Tomcat e Sunflow 3D *renderer*, variam de 21.987 a 28.314 LOC.

Os modelos de desempenho são construídos respeitando uma hierarquia de 3 níveis. Modelo de **Alto Nível** simula os fluxos das tarefas e como são processadas no programa. O modelo de **Médio Nível** simula os atrasos que acontecem enquanto as *threads* processam o programa. O modelo de **Baixo Nível** simula os atrasos que acontecem quando existe a concorrência de recursos das *threads*, como CPU, disco ou sincronização. Os modelos de desempenho gerados pelo PERSIK tiveram um erro médio para CPU entre 0.032 a 0.134 e para E/S 0.262 a 0.269

Os autores concluem que os modelos de predição para essas aplicações são difíceis de modelar, devido a quantidade de parâmetros que possuem. Vários destes influenciam diretamente no desempenho, como *locks* ou sincronização. Os *locks* ajudam a coordenar os trabalhos de cada *thread* na aplicação, e muitos programas utilizam diversas abordagens de implementação. Em âmbito geral, pode ser difícil para um *framework* descobrir todas elas. A modelagem de *hardware* também é essencial para a precisão dos resultados. Cada modelo de arquitetura (CPU, memória, disco, entre outros) pode desempenhar um comportamento diferente na aplicação.

A precisão em tais modelos é garantida pela correta medição do uso de recursos dos componentes de um programa. Pequenos erros durante a medição, podem impactar em grandes erros durante a predição. Isso deve ser evitado, pois a depuração de modelos de desempenho é uma tarefa difícil. Quando acontecem erros, se demanda um grande esforço para localizá-los, além do uso de ferramentas ou outros métodos.

Métodos de predição de desempenho são utilizados em muitas áreas da computação, pois alcançam um modelo de desempenho de forma rápida e, às vezes, muito precisa. No entanto, para que isso seja possível, muitos fatores estão envolvidos antes da construção do modelo e do *framework* que irá gerá-lo. Por exemplo, Tarvo menciona a importância de conseguir gerar outros comportamentos da aplicação *multithread* antes de ser utilizado um *framework* de predição. Tais comportamentos podem variar quando diferentes cargas de trabalho são utilizadas, ajustes de parâmetros que afetam um aspecto particular da aplicação, entre outros. Na maioria das vezes, os benchmarks para este domínio de processamento auxiliam nessa etapa. Porém, poucos deles permitem que o usuário explore outros comportamentos - alguns também não representam o domínio de aplicação que o usuário busca avaliar.

Com as versões parametrizáveis do Dedup e Ferret se consegue derivar diferentes comportamentos com o *benchmark*, pois são oferecidos além dos parâmetros básicos de uma aplicação paralela. Normalmente, o usuário encontra nos *benchmarks* parâmetros relacionados com o número de *threads* e tamanho de entrada, por exemplo. Com as parametrizações implementadas, o cenário é expandido e o usuário consegue ajustar o tamanho dos *buffers*, filas, fragmentação, segmentação e diferentes tipos de entradas. Além disso, outros tipos de dados de entrada podem ser testados. Essa possibilidade de ajustes é determinante para avaliar o comportamento de uma aplicação como o Dedup. Isso porque

manipular dados nos *buffers* exige o uso de *locks* e semáforos, características sensíveis para os métodos de predição de aplicações *multithreads*.

3.2 Domínios de Big Data, Mineração de Dados e Memórias Transacionais

StreamBench [LWXH14] é um *benchmark* para avaliar o processamento de *data stream* em *big data*. Além de avaliar a computação de *stream* distribuída, o *benchmark* também busca encontrar as necessidades de melhorias dos *frameworks* do cenário de *big data*.

Os autores declaram que o *benchmark* é pioneiro no cenário de computação de *stream* distribuída. Desse modo, inicialmente foi necessária uma classificação em três dimensões das cargas de trabalho que envolvem esse cenário, conforme descrito a seguir:

- **tipos de dados alvos:** texto ou números;
- **complexidade da computação:** as operações de *stream* básicas nas aplicações;
- **envolvimento da atualização do estado armazenado:** o processamento em tempo real pode exigir informações históricas;

A classificação foi importante para definir as aplicações necessárias para compor o *benchmark* e que cumprissem os requisitos mencionados anteriormente. Desse modo, sete aplicações foram selecionadas e são listadas a seguir:

1) **Benchmark de Identidade:** lê uma entrada e não aplica nenhuma operação. Usado como *baseline* para outros *benchmarks*.
2) **Benchmark Simples:** analisa a entrada do *stream* conforme uma probabilidade específica. Operação básica muito utilizada.
3) **Benchmark de Projeção:** extrai um certo campo do *stream*. Utilizado também como um pré-processamento em aplicações reais.
4) **Benchmark Grep:** checa se a entrada do *stream* contém *strings* específicas.
5) **Word Count:** aceito como um *micro benchmark* para *big data*.
6) **DistinctCount:** extrai um campo alvo de um registro armazenado em um conjunto e o tamanho desse conjunto; realiza a contagem das palavras também.
7) **Benchmark de Estatística:** calcula máximo, mínimo, soma e a média de um campo de entrada.

O *benchmark* utiliza quatro tamanhos de entradas que foram baseadas no volume de computação de *stream* do Twitter e do Facebook. A escala em tamanho dessas entradas são: 5 milhões de gravações(*baseline*); 10 milhões de gravações(2x); 20 milhões de gravações(4x); 50 milhões de gravações(10x). As métricas de processamento do *benchmark* são: ***throughput*** - a contagem média dos registros com os *bytes* processador por segundo; **latência** - a média do tempo, medido quando um registro chega até o fim do seu processamento. Essas métricas também são obtidas em um cenário de tolerância a falhas, onde o *benchmark* analisa o tempo de resposta do *framework* após uma quantia de gravações por minutos.

Os resultados mostram comportamentos diferentes nos *frameworks* Storm e Spark utilizados nos testes. O primeiro tem um *throughput* maior e sofre menos impacto durante os testes de tolerância a falhas. No Storm, a latência é menor com cargas de trabalho menos complexas.

StremBench foi proposto para um cenário de computação distribuída e não considera sistemas de processamento multi-cores. Isso exige uma grande quantidade de recursos para execução dos benchmarks e frameworks para o processamento de *big data*. A suíte é limitada nas questões que envolvem a parametrização da aplicação e no conjunto de entrada para o benchmark. O principal objetivo do benchmark é analisar os *frameworks* para o processamento de *big data* e um usuário não consegue explorar os comportamentos da aplicação.

Nosso trabalho oferece a versão parametrizada do Dedup e Ferret para que os comportamentos da aplicação sejam avaliados. Os comportamentos são derivados por meio de parâmetros definidos pelo usuário antes da execução do benchmark. Esses parâmetros foram discutidos na Seção 4.

No trabalho de Geisler *et.al* [GQS10] é proposto um *framework* para identificar parâmetros de configuração para mineração de dados em *data stream*. O *framework* serve para identificar a influência dos parâmetros na precisão e confiabilidade dos resultados no processo de mineração. A validação é feita no cenário simulado pelos próprios autores, seguindo as características de um cenário real.

O *framework* é formado por três componentes: a **fonte dos dados**, **o sistema de gerenciamento de *data stream*** e o **banco de dados espacial**. Em ambiente simulado, os dados processados pelo *framework* são coletados em tempo real e enviados por sensores acoplados em veículos que percorrem uma rodovia específica. A composição desses dados enviados é a posição atual do veículo, a velocidade e sua aceleração. O cenário proposto serve para identificar engarrafamentos em rodovias por meio da mineração de *data stream*. Para os testes, uma configuração padrão dos parâmetros no *framework* é definida:

1 taxa de penetração. Quantidade dos veículos que enviam ao banco de dados informações sobre o tráfego;

2 volume do tráfego. Medido em carros por hora;

3 tamanho da seção. Uma pequena área do total da rodovia;

4 *window size*. Tempo em segundos para considerar os dados do passado na mineração;

Alguns resultados sobre a sensibilidade não foram conclusivos no trabalho. No entanto, percebe-se que a parametrização do *framework* permitiu aos autores definir os detalhes cruciais para a mineração. Por exemplo, o volume do tráfego teve um forte impacto na precisão do resultados, enquanto a taxa de penetração não teve um efeito crucial.

Geisler mostra a importância de antecipar comportamentos nas aplicações antes de serem colocadas em produção. Os testes foram importantes para determinar os parâmetros que impactam na precisão dos resultados. Porém, seu cenário de aplicação é voltado para a mineração de dados e utiliza um ambiente simulado. Além disso, a paralelização não é levada em conta no processamento.

Eigenbench [HOC+10] é um *microbenchmark* para a avaliação de sistemas de memória transacionais. Sua proposta é que, por meio de parâmetros definidos antes da execução do *microbenchmark*, o usuário consegue reproduzir um determinado comportamento de aplicações de memória transacional. Tal comportamento é formado por características encontradas em cenários reais desses sistemas. São elas: **(I) simultaneidade**, **(II) tamanho do conjunto de trabalho**, **(III) comprimento da transação**, **(IV) contaminação**, **(V) localidade temporal**, **(VI) contenção**, **(VII) predominância** e **(VIII) densidade**.

As características que Eigenbench avalia são derivadas por meio do conjunto de parâmetros vistos na Tabela 3.1. Eles são informados pelo usuário em um arquivo de configuração ou por argumentos.

Tabela 3.1 – Parâmetros Eigenbench. Adaptado de Oguntebi *et.al*[HOC+10]

Nome	Objetivo		Nome	Objetivo
N	Nº threads		R_1	Leituras/tx do Array1
S	Random Seed		W_1	Escritas/tx do Array1
tid	Thread ID		R_2	Leituras/tx do Array2
loops	Nº de TX por thread		W_2	Escritas/tx do Array1
A_1	Tamanho Array1 (quente)		R_3i	Leituras do Array3 dentro de TX
A_2	Tamanho Array2 (suave)		W_3i	Escritas do Array3 dentro de TX
A_3	Tamanho Array3 (frio)		R_3o	Leituras do Array3 entre TXs
W_3o	Escritas do Array3 entre TXs		K_o	Escalonador para out-TX
Nop_i	Nº de operações entre acesso TM		persist	Restaurar Random Seed se violada
Nop_o	Nº operações fora de TX		lct	Probabilidade de repetição de endereços
K_i	Escalonador para in-TX		-	—

Os resultados do *microbenchmark* ajudam a entender a patologia das alterações feitas nos parâmetros. Com esses resultados, o comportamento de uma aplicação pode ser previsto e analisado por meio dos testes. O controle das *threads* e a paralelização do *microbenchmark* é feita com *pthread*.

Nosso trabalho foi inspirado pelo Eigenbench. Por meio dele, vimos que o comportamento de uma aplicação de memória transacional pode ser expressada por vários parâmetros. Porém, esse *benchmark* não consegue representar uma aplicação real, como o Dedup ou Ferret, e por isso não pode ser utilizado. Desse modo, procuramos estudar aplicações mais representativas de um cenário real de processamento.

3.3 Melhorias em Suítes de *Benchmarks*

O trabalho de Maleszewski [MS17] desenvolve recursos extras para o controle de execuções de *benchmarks*, que combina novas alternativas de medições com os *benchmarks*. Para isso, os autores utilizam a suíte Phoronix Test[1].

Segundo os autores, é mais apropriada a utilização de *benchmarks* no processo de avaliação de desempenho e em testes que envolvem esse cenário. Isso é defendido porque os *benchmarks* podem ser representados de forma sintética ou baseados em aplicações, permitindo que um cenário de processamento próximo ao real seja explorado. Tal questão se torna importante durante a etapa de desenvolvimento de um sistema. Com a execução do *benchmark*, os comportamentos de uma aplicação podem ser representados no sistema e o monitoramento de métricas de desempenho é um indicativo para esses comportamentos.

No entanto, problemas envolvendo o monitoramento do desempenho têm sido demonstrado pelos autores. Além disso, com vários experimentos no decorrer de suas pesquisas, foi necessário compreender o impacto geral do desempenho em diferentes configurações do sistema. Em consequência disso, os autores buscaram combinar técnicas de *benchmarking* com indicadores de desempenho. Foram desenvolvidas sob a plataforma de testes Phoronix Test Suíte (PTS) ferramentas suplementares para atender os requisitos que antes eram negligenciados pela literatura.

As modificações foram realizadas em três aspectos, *I) integração de indicadores de desempenho na PTS; II) integração do PTS com GNU/Linux* ***cgroups****; III) melhoria do módulo MATISK (My Automated Test Infrastructure Setup Kit) e na geração de plots*.

Utilizando a plataforma Phoronix com as modificações, os experimentos foram executados com os *benchmarks* disponíveis na plataforma de testes. As modificações foram possíveis, pois a plataforma PTS é *open source* e foram aprovadas pelo líder do projeto, Michael Larabel - sendo incluídas nas novas versões do PTS. Os resultados são obtidos por cinco conjuntos de testes organizados. Diferentes infraestruturas foram utilizadas entre os testes, e unicamente nos testes T5, quatro infraestruturas. Os testes são:

[1] https://www.phoronix-test-suite.com/

- **T1**: avaliando o impacto das *flags* de otimização do compilador GCC e o número de *threads* utilizado. Benchmark C-Ray com geração de imagem.

- **T2**: teste da velocidade de leitura e escrita em disco utilizando o benchmark Flexible I/O em três modelos de discos.

- **T3**: teste da eficácia do processador gráfico com o *benchmark* Unigine Valley.

- **T4**: teste de compressão com gzip, xz, bzip2 utilizando diferentes dados com diferentes níveis de compressão.

- **T5**: codificação de vídeos com *benchmark* x264.

Os autores concluem que a avaliação de desempenho depende dos *benchmarks*, da suíte de testes e das técnicas de coleta escolhidas para a avaliação. Com os testes realizados, foi possível coletar informações que representam o comportamento dos sistemas ou de componentes avaliados. Além disso, foi possível analisar e caracterizar as cargas de trabalho e o impacto delas no desempenho. Nem sempre a arquitetura de processamento é um estímulo para o desempenho. Porém, uma medição apropriada de parâmetros de desempenho, combinada com monitoramento, também é fator que ajuda a compreender o desempenho. Portanto, a modificação do PTS elimina a necessidade de instrumentações para alcançar os requisitos que melhoram uma avaliação de desempenho.

Maleszewski demonstrou a importância dos *benchmarks* para a avaliação de desempenho. Além disso, propõe uma discussão mostrando as limitações encontradas nos *benchmarks* existentes. A principal delas foi a falta de parametrização dos *benchmarks*. O trabalho mostra a importância dessa característica, pois com isso o cenário de testes se torna mais flexível e mais abrangente.

Os autores corrigiram algumas limitações implementando melhorias na suíte de testes Phoronix, que exigiu grande esforço e tempo. Porém, as melhorias ainda não atendem alguns aspectos em específico, por exemplo, o paralelismo. Os resultados das execuções são apresentados em forma de gráficos ou tabelas, e pouca discussão técnica é feita sobre eles. Isso mostra que o comportamento da aplicação não é o importante para o trabalho, e sim a oferta de parâmetros e diferentes métricas de desempenho que podem ser obtidas.

Nosso trabalho vai além disso, pois oferecemos parâmetros que estão diretamente relacionados com a aplicação, procurando explorar diferentes desempenhos. O foco de Maleszewski está na melhoria de uma suíte de testes para conseguir mais métricas de desempenho em experimentos e otimizar execuções em larga escala. Desse modo, ele implementa recursos em uma suíte consolidada que gerencia testes com os *benchmarks*. Porém, as melhorias não fazem parte das aplicações.

3.4 Visão Geral dos Trabalhos Relacionados

A Tabela 3.2 mostra a visão geral dos trabalhos discutidos nas seções anteriores.

Tabela 3.2 – **Visão geral dos trabalhos relacionados.** Na tabela são descritos os trabalhos discutidos anteriormente. O trabalho do PARSEC é discutido no Capítulo 1 e no Capítulo 4.

Trabalho	Objetivo	Contexto	Aplicações ou benchmarks	Parâmetros	Resultados
StreamBench [LWXH14]	Encontrar melhorias em *frameworks* para *big data*	Paralelismo distribuído de *stream* em *big data*	*Word Count. DistinctCount. Benchmarks*: de identidade, simples, projeção, *grep*, estatística	*Benchmarks* e conjuntos de entradas	*Storm* e *Spark* apresentam diferentes desempenhos
Framework [GQS10]	Identificar a influência de parâmetros na confiabilidade e precisão na mineração de dados	Mineração de *Data stream*	Aplicação simulada para identificar engarrafamentos	Taxa de penetração, volume do tráfego, tamanho da seção, tamanho da janela	Resultados sobre sensibilidade não conclusivos. Volume do tráfego impacta na precisão. Taxa penetração sem efeito crucial
Eigenbench [HOC+10]	Reproduzir comportamentos de aplicações de memórias transacionais	Sistemas de memória transacional	Genome, *Vacation-low labyrinth*, SSCA2, *Intruder*	21 parâmetros. Descritos na Tabela 3.1	Eigenbench permitiu variar cada características de forma independente. Forneceu uma visão sobre como o sistema é afetado por estas características. As características reproduzidas foram precisas ao comportamento de aplicações reais
Maleszewski [MS17]	Melhorar a plataforma *Phoronix Test Suíte* para oferecer novos parâmetros	Avaliação e coleta de resultados de desempenho	*Benchmarks:* C-Ray, *Flexible I/O*, Unigine Valley, Testes de compressão do *gzip,xz,bzip2*, codificação de vídeo com x264	Diferentes parâmetros são utilizados para cada testes. Para todos os *benchmarks* e testes, no total são 46	Cada *benchmark* pode ser avaliado no sistema com diferentes parâmetros. As melhorias facilitaram a coleta e o monitoramento dos resultados
Tarvo [TR18]	*Framework* para modelo de previsão automática de desempenho	Modelagem de desempenho de aplicações *multithreads*	Raytracer, Moldyn e Galaxy, Tornado, Sunflow 0.07 3D e Apache Tomcat 7.0	Diferentes parâmetros são utilizados para gerar as simulações	Os modelos gerados foram precisos. Gerar modelos completos para aplicações *multithreads* é complexo
PARSEC [Bie11]	Avaliar arquiteturas paralelas com processadores *multicores*	Características da arquitetura	*blacksholes, bodytrack*, *canneal, dedup, facesim*, *ferret, fluidanimate, freqmine*, *streamcluster, swaptions, vips* ,*raytrace, x264*	Tipos de configurações dos *benchmarks* (paralelo ou sequencial), entradas. Parâmetros de gerenciamento da suíte	Diversas características da arquitetura foram analisados
Este trabalho	Avaliar as características do paralelismo em aplicações de *stream*	Paralelismo de *stream*	Dedup e Ferret	14 classes de entradas. 2 classes de entradas ajustáveis. Janela Deslizante, tamanho do *buffer*, tamanho das filas, elementos do *stream*	Parametrização das características apresentou diferentes comportamentos nos *benchmarks*

4. CARACTERÍSTICAS DO PARALELISMO DE *STREAM*: PROJETO E IMPLEMENTAÇÃO

Este capítulo discute e especifica as características do processamento paralelo de *stream*. Detalhadamente, expandimos as seguintes contribuições desta dissertação, encontradas na Seção 1.1.

- **Identificação das características do paralelismo de *stream* que influenciam no comportamento de *benchmarks* da suíte PARSEC.**

- **Suporte à parametrização das características do paralelismo de *stream* em *benchmarks* da suíte PARSEC.**

Entender com clareza o domínio de processamento de *stream* foi importante para determinar suas principais características. Esse domínio de aplicação, possui processamento de dados que é constantemente assimilado com *data flow*, *data stream* e sistemas reativos. Na suíte PARSEC, o processamento de *stream* é abordado em seus *benchmarks*, que foram criados por meio de uma metodologia específica [Bie11]. Os *benchmarks* da suíte representam aplicações reais e emergentes, que foram desenvolvidas para estudar as características de arquiteturas paralelas com processadores *multicores*. Por esses motivos, o PARSEC foi o melhor cenário para explorar as características do processamento paralelo de *stream*, pois oferece aplicações reais que incluem versões paralelas delas em arquiteturas *multicores*.

Os *benchmarks* Dedup e Ferret do PARSEC foram selecionados por se encaixarem no domínio de aplicações de *stream*. Tais *benchmarks* representam tipos de aplicações diferentes, como compressão de dados e busca de similaridade em imagens. A partir disso, os esforços foram direcionados para a melhoria desses *benchmarks*, modificados a partir das versões paralelas originais da suíte PARSEC. A referida estratégia é semelhantemente observada no trabalho de Maleszewski [MS17] explicada no Capítulo 3, que, por sua vez, utilizou a suíte Phoronix.

O contexto da parametrização para os *benchmarks* foi desenvolvido com base nas características das aplicações de *stream*. A Janela Deslizante[1](Figura 4.1) e as filas são algumas destas características compartilhadas entre os *benchmarks* Dedup e Ferret do PARSEC. A Figura 4.1 é a representação mais detalhada de um estágio da Figura 2.2 na Seção 2.1, sendo possível entender como a janela deslizante atua sobre as filas. Com os *benchmarks* escolhidos é possível analisar como a parametrização dessas características influenciam em suas execuções.

[1] Sliding Window (SW)

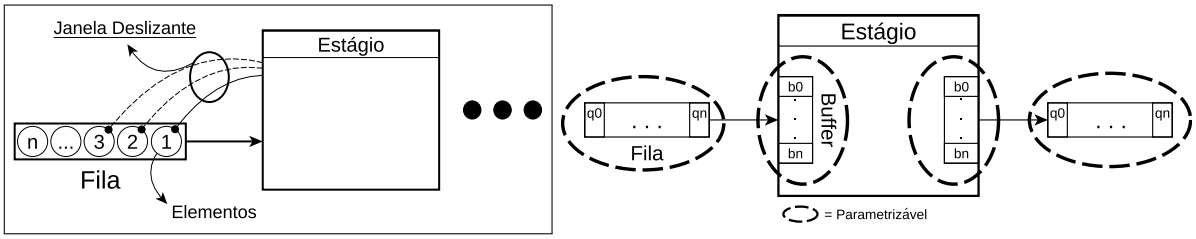

Figura 4.1 – Visão geral das parametrizações no estágio. A figura com o destaque (esquerda) exemplifica a janela deslizante baseada em elementos. As regiões em destaque na figura da direita, representam as características parametrizáveis na aplicação paralela de *stream*. Para as aplicações utilizadas neste trabalho, somente o Dedup implementa *buffers* nos estágios.

A Janela Deslizante representa a quantidade de elementos manipulados nas filas entre os estágios [AGT14]. Essa característica também está vinculada aos *buffers*. Um estágio manipula as filas quando algum tipo de computação precisa retirar ou colocar elementos na fila. Portanto, as regiões em destaque na figura representam tais características parametrizáveis. Normalmente, quando um estágio possui *buffers*, o tamanho dele é determinado pelo valor da janela deslizante e as filas serão maiores que os *buffers*. Por exemplo, a janela deslizante com valor 20 no Dedup, irá manipular 20 posições da fila, retirando ou colocando elementos um por um até que o limite 20 seja atingido.

As filas entre os estágios são utilizadas para comunicação entre os estágios vizinhos. Normalmente, réplicas de um estágio compartilham entre si essas filas e o acesso a elas exige um controle para evitar conflitos. O controle mais usual na computação paralela são os *locks*, que garantem o acesso exclusivo para as réplicas. Porém, se muitas réplicas são utilizadas, os *locks* podem gerar um gargalo para a aplicação, pois enquanto uma réplica está manipulando, as outras precisam aguardar sua vez. Esse é um processo sequencial e os *buffers* são utilizados para amenizar isso, armazenando de forma exclusiva os elementos nos estágios antes de serem manipulados nas filas. Os mencionados atributos do domínio de *stream* variam de acordo com a aplicação. Nesse caso, o *benchmark* Dedup utiliza filas e *buffers* e o Ferret utiliza somente as filas.

As métricas de desempenho são um reflexo do comportamento da aplicação. Um *benchmark* parametrizável deve possuir as métricas necessárias para demonstrar como as características parametrizáveis atuam no comportamento da aplicação. Essa foi uma questão mencionada em Maleszewski [MS17] no Capítulo 3. Na maioria das vezes, o tempo de execução não é suficiente para as aplicações de *stream*. Por exemplo, muitas aplicações desse domínio são dependentes de *throughput*, latência e *service time*, que são indicadores de desempenho para tais aplicações. No PARSEC essas métricas não são oferecidas, por isso foram implementadas.

Na aplicação, o *throughput* é a quantidade máxima de itens processados em um intervalo de tempo. Latência é o tempo de comunicação entre os estágios, indicando o tempo gasto por um elemento ao trafegar de um estágio ao outro. *Service Time* é o tempo

total do elemento na aplicação, desde o momento que entrou para ser computado até a sua saída. Essa métrica contabiliza toda a computação aplicada ao elemento, incluindo a comunicação entre os estágios, processamento, operações de escrita, entre outros. No Capítulo 3, o *benchmark* StreamBench [LWXH14] define isso como latência. *Throughput*, latência e *service time* são três métricas essenciais para um *benchmark* de aplicações de *stream* [HSS+14].

Para um *benchmark* tradicional de outro domínio de aplicação, as métricas de desempenho são informadas ao final da execução. Isso é diferente para os *benchmarks* de *stream*, pois algumas aplicações não possuem um critério de parada e seu processamento pode ser infinito. Desse modo, as métricas devem ser obtidas em tempo de execução para que seja possível acompanhar seu desempenho.

Os parágrafos seguintes detalham as modificações realizadas no Dedup e no Ferret, bem como de que forma as características de *stream* foram abordadas nos *benchmarks*.

4.1 Dedup

A versão original POSIX Threads do Dedup é formada por um *pipeline* de cinco estágios, explicados a seguir. O primeiro estágio processa a leitura dos arquivos e executa sua fragmentação em uma granularidade grande. O segundo estágio realiza uma fragmentação refinada dos dados recebidos do primeiro estágio, criando *chunks* de tamanho menor que são enviados ao próximo estágio. Usando um algoritmo SHA-1, o terceiro estágio gera uma identificação *hash* para cada *chunk* criado no segundo estágio. Cada chave *hash* é armazenada em uma tabela global, que posteriormente é utilizada para comparação com todas as outras chaves. Quando um *chunk* duplicado é encontrado, ele é enviado para o último estágio e somente o ponteiro para o dado duplicado é gravado em uma região especial do arquivo de saída. O quarto estágio recebe os *chunks* não duplicados para realizar a compressão local dessa quantia de dados utilizando o GZIP. O resultado dessa compressão é enviado ao estágio final em forma de ponteiro. Por fim, o último estágio realiza a montagem do arquivo de saída, gravando o resultado em um arquivo comprimido. Além disso, o estágio final deve ordenar todos os *chunks* gerados pela aplicação, pois a fragmentação em dois estágios diferentes gera o desordenamento no estágio final, mesmo sem haver estágios replicados.

Os conjuntos de entrada oferecidos pelo PARSEC para tal implementação são seis arquivos no formato TAR, que contêm unicamente arquivos de extensão DAT ou ISO, variando de 10 KB até 705 MB. As entradas só podem ser utilizadas a cada nova execução do *benchmark*, pois processam de maneira independente os arquivos de entrada (somente um por execução).

Na Figura 4.2 é possível visualizar a nova implementação do Dedup. Os parágrafos seguintes explicam como o processamento acontece em cada um dos estágios.

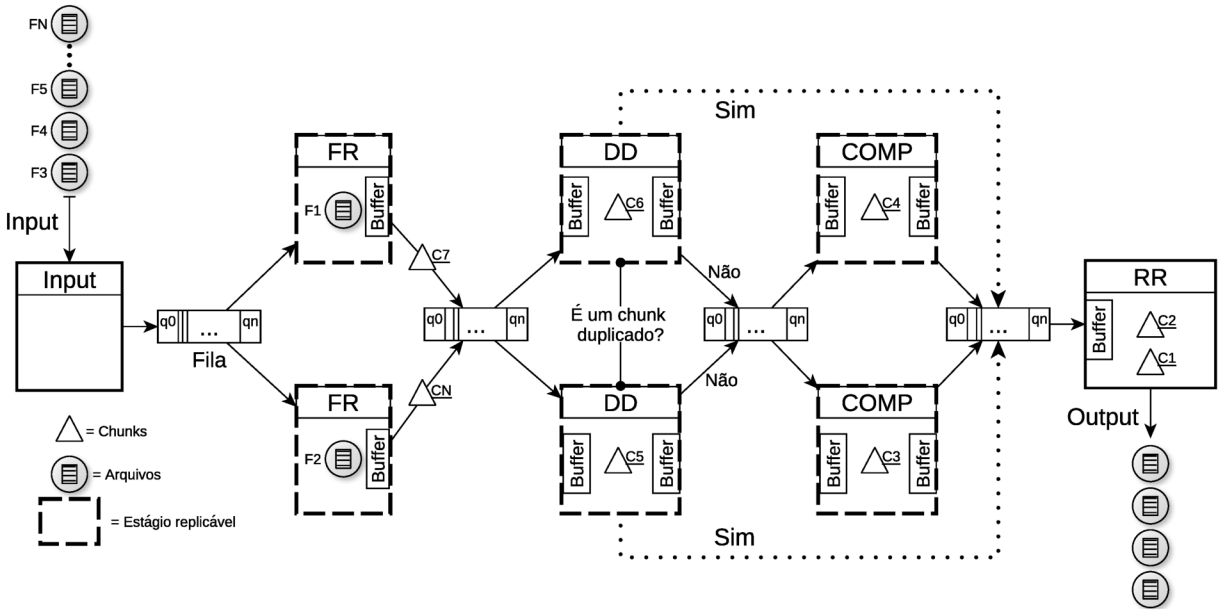

Figura 4.2 – Visão geral do Dedup modificado. Primeiro estágio *Input*. Segundo estágio *Fragment Refine* (FR). Terceiro estágio *Deduplication* (DD). Quarto estágio *Compression* (COMP). Quinto estágio *Reorder* (RR). As setas indicam o fluxo ordenado do *stream*. Estágios representados com as linhas tracejadas indicam estágios paralelos. Como forma de ilustração, a figura representa o Dedup com grau 2 de paralelismo.

A nova versão do Dedup foi modificada para poder processar arquivos maiores que 1GB e conseguir indicar diretórios para o *benchmark* processar mais arquivos por execução. Esses recursos não eram possíveis utilizando a versão original. Nessa implementação realizada, a leitura dos arquivos acontece no primeiro estágio, que recebe um diretório como parâmetro de entrada e verifica a quantidade de arquivos e o formato deles. Na figura, esse processo é identificado no estágio *input*. Foi mantido o comportamento da versão original de processar somente arquivos no formato TAR e excluída a primeira fragmentação que acontecia no primeiro estágio. De acordo com Bienia [BKSL08], a fragmentação do primeiro estágio acontecia para minimizar os custos de leitura em disco, na tentativa de criar *chunks* estáticos com até 128 MB de dados. Se essa estratégia de fragmentação do Bienia fosse mantida, seria uma tarefa difícil identificar quais *chunks* pertenceriam a determinado arquivo.

Tal dificuldade ocorre devido o fluxo desordenado na aplicação, representado pela linha pontilhada na Figura 4.2. Por exemplo, em um cenário com a versão original do Dedup, o escalonamento de trabalho entre o estágio 1 e 2 é baseado pelo método *round-robin*, onde cada réplica disponível do segundo estágio processa um dos *chunks* estáticos gerados no primeiro estágio. Quando é processado somente um arquivo, fica evidente que todos os *chunks* gerados no Dedup, tanto no primeiro quanto no segundo estágio, serão deste único arquivo. Assim, com o ordenamento de dois níveis no estágio final, estes *chunks*

seriam corretamente identificados. Porém, se mais arquivos fossem utilizados, devido ao desordenamento, haveria uma sobrecarga no estágio final para identificar de qual arquivo os *chunks* se originaram, podendo ser necessário um outro nível de ordenação e controle para os *chunks* gerados no segundo estágio.

Com a nova estratégia, cada arquivo lido no primeiro estágio é inteiramente enviado ao segundo estágio para a fragmentação refinada ser realizada, gerando os *chunks* para o Dedup. O *chunk* é equivalente ao elemento de *stream* no fluxo. Na Figura 4.2 o estágio de fragmentação é identificado como *FR*. Essa estratégia faz com que estes arquivos tenham um tamanho aproximado dos *chunks* estáticos que eram gerados no primeiro estágio na versão original. Além disso, com a fragmentação acontecendo em uma só réplica, a quantidade de *chunks* criados em cada arquivo é controlada internamente no estágio. Isso evita que o estágio *fragment refine* seja transformado em um estágio com estado (*statefull*). Posteriormente, no estágio final, o processo de ordenação faz uso do controle do número de *chunks* que o segundo estágio gerou. A descrição detalhada sobre os estágios com estado está no Capítulo 2.

O processo para verificar a deduplicação é feito de maneira individual para cada arquivo no estágio *DD* na Figura 4.2. Em outras palavras, significa que uma tabela HASH é exclusiva para cada arquivo e garante que *chunks* de um arquivo não sejam comparados com *chunks* de outros arquivos. É nesse estágio que acontece o desordenamento do fluxo de *stream*, identificado pela linha pontilhada na figura entre o estágio *DD* e *RR*, por isso dificultaria o controle dos *chunks* se a fragmentação do primeiro estágio fosse mantida, conforme explicado anteriormente. Quando o *chunk* é duplicado, ele é enviado diretamente ao estágio final e não passa pelo estágio de compressão. Por outro lado, os *chunks* não duplicados são enviados ao estágio de compressão e são processados normalmente.

O estágio final de montagem dos arquivos de saída recebe os *chunks* processados nos estágios de deduplicação e compressão. Devido a esse desordenamento no processamento explicado anteriormente, foram mantidos os dois níveis de ordenação dos *chunks* que o estágio *reorder* realiza. Em seguida é explicado como a ordenação garante que os *chunks* sejam escritos corretamente no arquivo final.

No Dedup, o *chunk* é representando por uma estrutura que contém as informações necessárias para o processamento de cada estágio. Uma das informações armazenadas é a identificação dos *chunks* no Dedup. O Código 4.1 representa parte da estrutura do *chunk* e a *Struct* `sequence_t sequence`, na linha 4, é a estrutura que armazena essas identificações. Elas são representadas em dois níveis, como observado na declaração da *struct* `sequence_t` no Código 4.2, nas linhas 2 e 3. Portanto, com o `l1num`, o algoritmo *Search Tree* posiciona a ordem dos arquivos lidos. A identificação desse primeiro nível acontece no estágio Load, na Figura 4.2. Com o `l2num`, o algoritmo Heap ordena os *chunks* de cada arquivo gerado no segundo estágio. Dessa maneira, é possível identificar os *chunks* que

pertencem a cada arquivo para escrevê-los corretamente no arquivo de saída, pois cada um carrega consigo sua origem e identificação.

```
1 typedef struct _chunk_t {
2   /* . . . */
3 #ifdef ENABLE_PTHREADS
4   sequence_t sequence;
5   int isLastL2Chunk;
6 #endif //ENABLE_PTHREADS
7 } chunk_t;
```

Código 4.1 – *Struct* do *chunk*. Nesse código são representados sete, das 49 linhas de código da *struct*. `isLastL2Chunk` (linha 5) é utilizado para identificar o último *chunk* do arquivo no estágio *fragment refine*.

```
1 typedef struct _sequence_t {
2   sequence_number_t l1num;
3   sequence_number_t l2num;
4 } sequence_t;
```

Código 4.2 – *Struct* para armazenar a identificação dos *chunks*.

Para avaliar a nova implementação do Dedup, foram criados novos conjuntos de entrada, diferente dos oferecidos pelo PARSEC. Esse conjunto é dividido em quatro classes que são chamadas de *Test, Light Duty Class, Heavy Duty Class* e *Free Class*. As classes *Light Duty* e *Heavy Duty* são divididas em outras quatro sub-classes, identificadas como L1 e H1, L2 e H2, L3 e H3, e LS e HS (*shuffled*). As classes *Light Duty* e *Heavy Duty* foram criadas para que o Dedup seja executado em infraestruturas com diferentes capacidades de processamento. A primeira é destinada para infraestruturas com processadores e memórias mais limitados, como em *desktops*, por exemplo. Já a segunda, pode ser utilizada em servidores com alto poder computacional, normalmente utilizados em testes de alto desempenho.

A numeração das sub-classes indica os tipos de arquivos que o Dedup irá processar. Essas sub-classes foram criadas com a intenção de avaliar o impacto no processamento de tipos específicos de elementos do *stream* que são processados. Nesse caso, o elemento de *stream* será formado por partes (que são os *chunks*) do conjunto de entrada. A sub-classe 1 são arquivos do tipo fotos no formato JPG e BMP. A sub-classe 2 são arquivos do tipo texto, nos formatos TXT, PDF e EPUB. Vários desses arquivos foram selecionados de uma base de artigos acadêmicos e diversos livros de literaturas. Outros arquivos TXT foram intencionalmente criados com *strings* randômicas e sequências contínuas de um único caractere. Na sub-classe 3, são os arquivos de áudio e vídeo, nos formatos MP3, AVI, AAC, WAV, AVI, FLV e MP4.

A sub-classe *shuffled* contém arquivos das sub-classes anteriores. O conteúdo dos arquivos varia em quantidade e tamanho. A sub-classe foi criada dessa forma para avaliar como o Dedup se comporta quando existe uma variação nas taxas de entrada do

stream, que é influenciada pelo tamanho do arquivo, e quando este *stream* é composto por diferentes tipos de dados. Por exemplo, um dos arquivos criados para a sub-classe *shuffled* contêm imagens reaproveitadas da sub-classe 1. Assim, nesse arquivo existem imagens que partem de 60 KB e podem alcançar até 30 MB. Esse comportamento de variação semelhante se repete para outros arquivos originados das sub-classes H2 e H3.

Uma coleção de arquivos serve como base em todas as sub-classes, formando um conjunto de entrada para o *benchmark*. Esses arquivos estão no formato TAR e variam de 50 MB até 550 MB. Por exemplo, um teste inicial com sub-classe H1 iria processar cinco arquivos que, juntos, totalizam 1.5 GB de dados. Para uma teste com maior escala, tais arquivos podem ser replicados dentro do diretório. Ao criar todas as classes, intencionalmente duplicou-se 10% do conteúdo em cada arquivo. A Classe *Test* é utilizada em execuções rápidas que não possuem um propósito de avaliação. A *Free Class* fica a critério do usuário definir os tipos de arquivos e a quantidade deles. Com as modificações apresentadas anteriormente, o usuário consegue montar seu conjunto de entrada ou escolher alguma das classes oferecidas.

No *benchmark*, os tamanhos das filas são fixas com mais de 1 milhão de posições. Foi mantido este valor da versão original devido à característica do Dedup, porque o *benchmark* gera muitos *chunks*, e nessa situação, o tamanho dos *buffers* teve um impacto maior para a análise do comportamento do *benchmark*. Desse modo, o tamanho dos *buffers* é parametrizável, indicando também o valor da janela deslizante.

O *chunk* do Dedup também é uma característica particular das aplicações do domínio de compressão de arquivos. Por outro lado, no domínio de processamento de *stream*, o *chunk* equivale ao elemento *stream*. No Dedup, essa é outra característica que também pode ser parametrizada. Esse parâmetro atua no algoritmo de fragmentação no segundo estágio e terá impacto na quantidade e no tamanho dos *chunks*.

O *Basic Slidwing Window* (BSW) [MCM01], juntamente com o algoritmo Rabin-Karp *Fingerprint* [Rab81], realizam a fragmentação. O BSW utiliza três parâmetros: um tamanho fixo de janela **(W)**, um valor pré-determinado **(D)** e um restante inteiro **(R)**, onde $R < D$. Na configuração padrão do Dedup no PARSEC, o valor **W** é definido em 12 (*bits*), o valor **D** em 32 (bytes), e $0 \leq R \leq 32$. Para encontrar um ponto de divisão (*anchor*[2]), o algoritmo percorre os dados do arquivo com uma janela de tamanho 32 *bytes*. Para cada *byte* dessa janela, o algoritmo calcula uma chave *hash* com o Rabin-Karp. Um ponto de divisão é encontrado quando os últimos 12 *bits* dessa chave somam 0. Esse processo de fragmentação é detalhado na Figura 4.3.

O processo de fragmentação foi mantido, pois alterar essa técnica não estava no escopo do presente trabalho. No entanto, é importante mencionar que o BSW apresenta algumas limitações que alteram significativamente a quantidade e o tamanho dos *chunks*. Isso acontece por dois problemas [ET05]. O primeiro é que o algoritmo pode en-

[2]Nomenclatura utilizada no Dedup do PARSEC para indicar um ponto de divisão no dado.

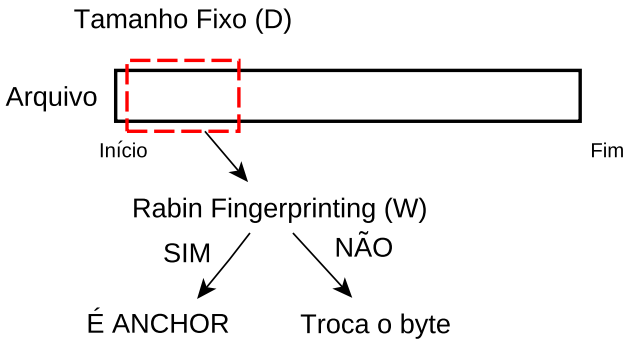

Figura 4.3 – *Basic Sliding Window* (BSW) [MCM01, Cha09].

contrar um ponto de divisão a cada troca de byte. Por exemplo, quando uma sequência de "*11111111*" ou "*aaaaaaaaa*" compõe os dados do *chunk*. Outro é que um ponto de divisão pode não ser encontrado. Eshghi [ET05] resolve essas limitações com o algoritmo *Two Thresholds, Two Divisors Algorithm* (TTTD). Outros trabalhos também mostram que arquivos empacotados no formato TAR impactam na fragmentação, variando o tamanho e a quantidade de *chunks* [WDQ+12].

A quantidade de filas entre os estágios é escalável de acordo com o número de réplicas. Essa quantidade é obtida com a divisão do número de réplicas passado como argumento na linha de comando pela quantidade máxima de réplicas por fila (definido em 4). Tal estratégia foi mantida e, para esse caso, não foram exploradas estas características no Dedup. O argumento que define o grau de paralelismo na linha de comando indica a quantidade de réplicas para os estágios paralelos. Na Figura 4.2, os estágios replicáveis estão em destaque (linhas tracejadas).

4.2 Ferret

A versão original do Ferret é paralelizada com POSIX Threads e possui seis estágios, sendo representada na Figura 4.4. O primeiro e o último estágio são sequenciais. O primeiro, lê as imagens do disco e realiza a descompressão JPEG para extrair o cabeçalho das imagens, para obter RGB, HSV e a resolução. Para todas as imagens lidas, estas informações são armazenadas em uma estrutura e colocadas na fila do próximo estágio. Essa estrutura pode ser observada no Código 4.3 em linguagem C, nas linhas 1 até 8. Na Figura 4.4, a computação do primeiro estágio acontece no estágio *Load*.

No Ferret, o elemento de *stream* sempre será uma estrutura utilizada para armazenar os dados para o processamento em cada estágio. Desse modo, as informações dessa estrutura mudam de acordo com a computação realizada nos estágios. No decorrer dos próximos capítulos, os estágios e as estruturas são discutidas.

```
1 /* LOAD Stage */
2 struct load_data{
3   int width, height;
4   char *name;
5   unsigned char *HSV, *RGB;
6   float size_file;
7   int idx;
8 };
```

Código 4.3 – *Struct* para o estágio *load*

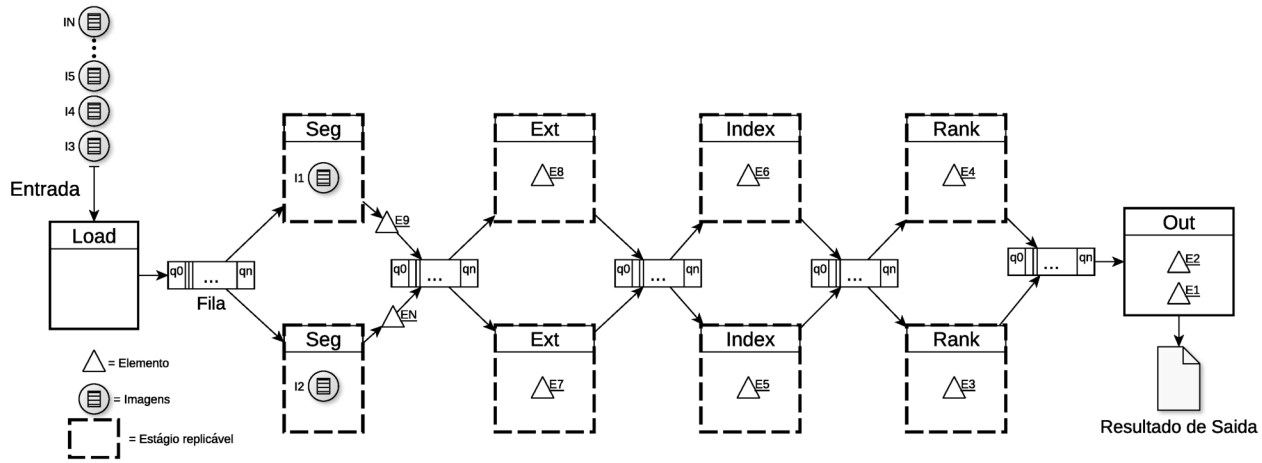

Figura 4.4 – Visão geral do Ferret. Primeiro estágio *Load*. Segundo estágio *Segmentation* (Seg). Terceiro estágio *Extraction* (Ext). Quarto estágio *Indexing* (Index). Quinto estágio *Rank* (Rank). Sexto estágio *Output* (Out). As setas indicam o fluxo ordenado do *stream*. Estágios representados com as linhas tracejadas são paralelos. Como forma de ilustração, a figura representa o Ferret com grau 2 de paralelismo.

O segundo estágio recebe as informações das imagens para realizar a segmentação delas. Ele é identificado como *Seg* na Figura 4.4. A principal computação desse estágio é realizar a segmentação utilizando o algoritmo *Statical Region Merging* (SRM), que organiza os pixels em conjuntos, e cada conjunto representa uma região da imagem [NN04]. O Código 4.4 representa a estrutura que armazena essas informações, onde o número de regiões originado pela segmentação é indicado na variável `nrgn` (linha 3). Essa segmentação inicia com uma decomposição fina (pequenos conjuntos) e vai unindo eles até o final da segmentação. Esse resultado é organizado em uma estrutura e enviado ao terceiro estágio para extrair 14 características de cada região nas imagens. Na Figura 4.4 é o estágio *Ext*.

```
1 /* Segmentation Stage */
2 struct seg_data{
3   int width, height, nrgn;
4   char *name;
5   unsigned char *mask;
6   unsigned char *HSV;
7   int idx;
8 };
```

Código 4.4 – *Struct* para o estágio *segmentation*

As 14 características são as delimitações (*bounding box*) e as cores de cada segmento. No Código 4.5, a estrutura `cass_dataset_t ds` agrupa estas características. Tal estrutura é particular da biblioteca utilizada no desenvolvimento do Ferret. Outras informações sobre essa biblioteca são abordadas nos próximos parágrafos.

```
1 /* Extraction Stage */
2 struct extract_data{
3   cass_dataset_t ds;
4   char *name;
5   int idx;
6 };
```

Código 4.5 – *Struct* para o estágio *extraction*

Com as características de cada imagem agrupadas em estruturas, o quinto estágio faz uma pré-indexação com possíveis imagens que irão aparecer no resultado final. Esse estágio é identificado na Figura 4.4 como estágio *Index*. O estágio *Index* aplica o método de indexação *Locality Sensitive Hashing* (LSH) [LJW+07] para consultar em um banco de dados as imagens para comparação da similaridade e o resultado dessa consulta é armazenado em uma estrutura. No Código 4.6, o resultado é armazenado na *struct* `cass_result_t result`, particular da biblioteca do Ferret.

```
1 /* Indexing Stage */
2 struct vec_query_data{
3   char *name;
4   cass_dataset_t *ds;
5   cass_result_t result;
6   int idx;
7 };
```

Código 4.6 – *Struct* para o estágio *indexing*

O banco de dados utilizado para consulta é alocado em memória antes da computação ser iniciada e contém tabelas *hash* de 59.695 imagens, também indexadas pelo método LSH.

O resultado da computação do quinto estágio é enviado ao sexto estágio, *Rank* na Figura 4.4, para realizar a busca de similaridade detalhada das imagens. A métrica *Earth Mover's Distance* (EMD) é utilizada para classificar as imagens com base na sua similaridade para todas as imagens lidas no primeiro estágio. As estruturas dos estágios *Index* e *Rank* são similares, porém, o resultado armazenado na *struct* `cass_result_t result` do Código 4.7 terá o resultado final da busca de similaridade. Por fim, o estágio *Out* recebe essa mesma estrutura do estágio *Rank* para salvar em um arquivo o resultado final das imagens processadas com sua respectiva classificação. No PARSEC, o Ferret pode ser processado com seis conjuntos de entrada, variando entre uma imagem até 3.500 imagens que não ultrapassam 120 X 120 pixels. O banco de dados para consulta também varia, partindo de uma até 59.695 imagens.

Compreender os detalhes que envolvem o Ferret foi importante para torná-lo parametrizável. A medida que o *benchmark* foi sendo explorado, diversas limitações surgiram. Portanto, antes de apresentar a versão parametrizável do Ferret, essas limitações são discutidas a seguir, pois impactaram em nossas decisões de projeto.

```
1 /* Ranking Stage */
2 struct rank_data{
3   char *name;
4   cass_dataset_t *ds;
5   cass_result_t result;
6   int idx;
7 };
```

Código 4.7 – *Struct* para o estágio *rank* e *out*

O *benchmark* Ferret foi implementado utilizando a biblioteca CASS-image (*Content Aware Similarity Search for Images*)[LCL04, LJW+06, Lv06], projetada para processar a busca de similaridade em imagens, vídeos e áudios. A documentação sobre essa biblioteca é insuficiente para que novas estratégias de implementações sejam avaliadas. Por isso, o avanço para a melhoria do *benchmark* foi lento e limitado, pois o objetivo não está na criação de um novo *benchmark*.

O Ferret trabalha com dois tipos de entradas, que são as imagens passadas como argumento de entrada no Ferret e o *dataset* para a consulta de similaridade. A metodologia para criação desses dois conjuntos de entradas não é suficientemente detalhada nas documentações disponíveis. As informações encontradas indicam que o *dataset* contém dados de diversas imagens e sua estrutura é semelhante a um banco de dados tradicional. As imagens agrupadas neste *dataset* estão representadas por um conjunto de *sketch*. O *sketch* é a representação compacta das regiões de uma imagem que foi segmentada [Lv06]. O problema foi compreender como o *dataset* de consulta foi projetado e quais eram as características das imagens utilizadas para criá-lo.

A complexidade que envolve a biblioteca *CASS-image* e a documentação pouco detalhada não permitiu a criação de um novo *dataset* para o cenário da presente dissertação. Por exemplo, a computação do estágio 3 cria as *sketchs* das imagens recebidas como entrada no Ferret. No Código 4.5, 4.6 e 4.7, as *sketchs* destas imagens são armazenadas na estrutura `cass_dataset_t ds`. No *dataset* disponibilizado pelo PARSEC para o Ferret, as *sketchs* já foram criadas e agrupadas, por isso são diferentes das geradas no estágio 3. Portanto, como não foram encontradas instruções claras nas documentações, não foi possível criar um novo *dataset* para o cenário de testes dessa dissertação.

A versão parametrizável do Ferret precisou ser adaptada, considerando as limitações anteriormente discutidas. Com a nova versão é possível definir o tamanho das filas entre os estágios e diferentes entradas. O maior *dataset* do PARSEC foi utilizado para a consulta das imagens, já que não foi possível criar um novo. Devido ao comportamento da aplicação, a janela deslizante é fixada em 1 e somente o tamanho das filas pode ser para-

metrizado. O algoritmo de segmentação também pode ser parametrizado e são necessários os parâmetros *Q Value* e *threshold* para o SRM segmentar as imagens. O primeiro representa o tamanho máximo de uma região da imagem; o segundo indica o tamanho mínimo de uma região que deve ser unida com outras regiões menores ou de mesmo tamanho.

O novo conjunto de entrada criado para o Ferret é dividido em quatro classes, que são chamadas de *Test, Light Duty Class, Heavy Duty Class* e *Free Class*. As classes *Light Duty* e *Heavy Duty* são divididas em outras três sub-classes, identificadas como L1 e H1, L2 e H2, LS e HS (*shuffled*). As classes *Light Duty* e *Heavy Duty* foram criadas para que o Ferret seja executado em infraestruturas com diferentes capacidades de processamento. A primeira é destinada para infraestruturas com processadores e memórias mais limitados, como em *desktops*. A segunda pode ser utilizadas em servidores com alto poder computacional, normalmente para testes de alto desempenho.

A numeração nas sub-classes indicam o tamanho da imagem, tanto em resolução quanto tamanho do arquivo. A sub-classe 1 compõe as imagens com resoluções pequenas, variando entre 800x600, 678x435, 128x96[3] e cada imagem tem aproximadamente 1 MB. Na sub-classe 2, compõe imagens com resoluções maiores que as da sub-classe 1. As imagens variam entre 14400x7200, 3000x2000, 5472x3648[4] e não ultrapassam 14 MB. A sub-classe *shuffled* contém imagens das sub-classes anteriores. Nas classes de entradas todas as imagens são do formato de arquivo JPEG.

O Ferret do PARSEC é uma aplicação específica para o processamento de imagens, configurada para busca de similaridade em imagens. Nessa configuração, a qualidade da busca é estimada. Portanto, o conjunto de entrada criado serve para avaliar o comportamento do Ferret quando processa imagens com diferentes resoluções, que nesse caso irá influenciar apenas no tamanho do elemento de *stream*. Não foi possível avaliar diferentes tipos de dados do elemento de *stream*, pois é necessária a reimplementação do Ferret, discutida em seguida.

Mantendo o processamento com imagens, o elemento do *stream* poderia ser variado quando outros formatos de imagens fossem testados, como BMP ou PNG. Para isto, seria necessário reimplementar os estágios *load* e *segmentation*, para reconhecer os cabeçalhos de diferentes formatos e adequá-los ao processo de segmentação. Outra abordagem poderia incluir áudio, vídeo, ou objetos 3D além das imagens[LJW+06]. Porém, na versão original do PARSEC, isso não foi considerado. Desse modo, devido às documentações disponíveis serem limitadas, nenhuma das referidas abordagens puderam ser avaliadas, por isso não foram consideradas no escopo deste trabalho.

[3]Outras resoluções de imagens também estão inclusas, mas não ultrapassam a maior resolução de 800x600.

[4]Outras resoluções de imagens também estão inclusas, mas não ultrapassam a maior resolução de 14400x7200.

A sub-classe *shuffled* foi criada para avaliar como o Ferret se comporta quando existe uma variação nas taxas de entrada do *stream*, que é influenciada pelo tamanho dos arquivos e das resoluções das imagens.

4.3 Parametrização, Simplicidade e o Processamento Paralelo de *Stream*

Nas seções anteriores foram apresentadas as modificações necessárias para tornar parametrizável os *benchmarks* Dedup e Ferret. O embasamento para alcançar estes requisitos tem a intenção de avaliar algumas características do paralelismo de *stream* das aplicações. A Tabela 4.1 resume as características parametrizáveis no Dedup e Ferret implementadas para este trabalho. Outras características do paralelismo de *stream* poderiam ser avaliadas no Dedup e no Ferret, ou em outras aplicações, mas não foram consideradas nesta pesquisa.

Tabela 4.1 – Resumo da parametrização no Dedup e no Ferret.

	Características Parametrizáveis do *Stream*				
	Janela Deslizante	**Fila**	***Buffer***	**Tamanho do Elemento do *Stream***	**Tipo de Dado do *Stream***
Dedup	Parametrizável	Parametrizável*	Parametrizável	Fragmentação do *Chunk*	Texto, imagem, vídeo, áudio
Ferret	Estático	Parametrizável	Não utiliza	Segmentação da Imagem**	Somente imagens
*Parametrização não foi explorada devido ao escalonamento de trabalho, discutido na Seção 4.1. **De acordo com o tamanho da imagem					

Explorar as características de *stream* depende das características dos *benchmarks*. No caso do Ferret, a janela deslizante e o *buffer* não podem ser parametrizáveis devido às limitações de implementação. As limitações também impactaram na criação dos conjuntos de entrada do Ferret, por isso a variação foi somente no tamanho das imagens, na resolução e no tamanho do arquivo. Para processar outros formatos seria necessária uma reimplementação do *benchmark*. Detalhes sobre isso são discutidos na Seção 4.2.

Dentre vários critérios que um *benchmark* deve seguir, simplicidade é o que torna ele compreensível [Gra92]. Alguns trabalhos utilizam usabilidade para descrever tal critério [AHL+15]. No PARSEC, a usabilidade é garantida por um *shell script* (*parsecmgmt*) para gerenciar os *benchmarks* da suíte. Com esse *script*, o usuário pode instalar, executar e remover os *benchmarks* de maneira simples, por linha de comando. A seguir é mostrado um exemplo de utilização do *script* original do PARSEC. O exemplo se dá com o Dedup e o primeiro comando demonstra a instalação. Os argumentos `build`, `-p dedup` e `-c gcc-pthreads` indicam que o Dedup será instalado com a configuração POSIX Threads.

```
$ parsecmgmt -a build -c gcc-pthreads -p dedup
```

Após a instalação, é necessário executar o Dedup com a configuração desejada. O comando a seguir irá executar o Dedup com quatro *threads*[5] e a entrada *native*. É necessário definir a configuração de execução, que nesse caso é *gcc-pthreads*.

```
$ parsecmgmt -a run -c gcc-pthreads -p dedup -n 4 -i native
```

Para remover, é necessário indicar o Dedup e a configuração que foi utilizada na instalação, como mostra o comando em seguida.

```
$ parsecmgmt -a uninstall -c gcc-pthreads -p dedup
```

Em se tratando do Ferret, os procedimentos são os mesmos mostrados anteriormente. Porém, no argumento `-p` coloca-se `ferret`. Esses foram alguns dos comandos mais básicos para se utilizar esses *benchmarks*. Diversas opções estão disponíveis, como configurações, entradas e *benchmarks*, e outras[6].

No entanto, com o *parsecmgmt*, as parametrizações desenvolvidas nos *benchmarks* não podem ser utilizadas. Em razão disso, foi necessário desenvolver um novo *script* e integrá-lo com o *parsecmgmt*. No *parsec_stream* desenvolvido nessa dissertação, os *benchmarks* são instalados e executados com as parametrizações desejadas. Para tornar simples a parametrização, em cada *benchmark* existe um vocabulário para indicar onde a parametrização será aplicada. O vocabulário foi criado com base nas características dos *benchmarks* discutidas anteriormente. Além da parametrização, também é possível gerenciar o monitoramento do desempenho dos *benchmarks* e executar rapidamente quando uma parametrização já foi escolhida. Todas essas questões são apresentadas a seguir.

A parametrização no Dedup é feita da seguinte forma:

- `parsec_stream -p dedup -i [classe de entrada] \`
  `-r [n_réplicas] -a [ação] ‘‘VALOR"`
  As ações podem ser:
  - `-fr`: janela deslizante e *buffer* do estágio *fragment refine*.
  - `-dd`: janela deslizante e *buffers* do estágio *deduplication*.
  - `-comp`: janela deslizante e *buffers* do estágio *compression*.
  - `-rr`: janela deslizante e *buffer* do estágio *reorder*.
  - `-chunk`: fragmentação dos dados.
  - `-trace`: habilita o modo *tracing* de desempenho.

[5] *Threads* é a nomenclatura que o PARSEC utiliza. Nesta dissertação isto é diferente e o termo réplicas é utilizado.

[6] Mais detalhes podem ser obtidos acessado o site oficial da suíte PARSEC. `http://parsec.cs.princeton.edu/index.htm`

- `-notrace`: desabilita o modo *tracing* de desempenho.

Os parâmetros das ações podem ser feitos em diferentes ordens e nem todos eles precisam ser utilizados. Quando algum parâmetro não é utilizado, um valor padrão é automaticamente atribuído pelo *script*[7]. No entanto, para os *buffers FR* e *RR* existem algumas particularidades. No *FR*, apenas o *buffer* de saída do estágio é ajustado. No *RR*, apenas o de entrada, pois é um estágio da extremidade do pipeline e não precisa de *buffer* de saída, já que o resultado é gravado diretamente em arquivo. No *FR*, o *buffer* de entrada é utilizado para o escalonamento dos arquivos para as réplicas do estágio. Na tentativa de garantir um balanceamento de carga uniforme, quando uma réplica está disponível, este *buffer* recebe somente um arquivo por vez. Esse arquivo tem origem do estágio *input*. Por exemplo, caso sejam utilizados 10 arquivos e duas réplicas no estágio FR, é esperado que cada réplica processe cinco arquivos. Porém, isso não é totalmente garantido por causa do algoritmo de fragmentação, que pode processar mais rapidamente algum arquivo.

Com o Ferret, a parametrização segue deste modo:

- `parsec_stream -p ferret -i [classe de entrada] \`
  `-r [n_réplicas] -a [ação] ‘‘VALOR"`
  As ações podem ser:
  - `-seg`: tamanho da fila entre estágios *Load* e *Segmentation*.
  - `-ext`: tamanho da fila entre estágios *Segmentation* e *Extraction*.
  - `-idx`: tamanho da fila entre estágios *Extration* e *Indexing*.
  - `-rank`: tamanho da fila entre estágios *Indexing* e *Ranking*.
  - `-out`: tamanho da fila entre estágios *Ranking* e *Out*.
  - `-precision`: precisão para segmentação das imagens.
  - `-ranking`: quantidade máxima para encontrar as imagens similares.
  - `-trace`: habilita o modo *tracing* de desempenho.
  - `-notrace`: desabilita o modo *tracing* de desempenho.

Assim como no Dedup, a parametrização das ações também pode ser feita em diferentes ordens e nem todos os parâmetros precisam ser definidos. Para os parâmetros não utilizados, um valor padrão é automaticamente atribuído pelo *script*.

Ao utilizar qualquer combinação dos parâmetros do Dedup e do Ferret mostrados anteriormente, por padrão o *scritp parsec_stream* trata isso como uma execução e não é necessária a instalação. No entanto, quando são utilizados parâmetros como `-chunk` no Dedup e `-precision` no Ferret, por exemplo, a modificação acontece nos códigos fonte onde

[7]Os valores padrões atribuídos pelo *script* são os mesmos valores encontrados nas implementações originais do PARSEC. Para o Dedup, os valores estão descritos na Seção 5.1.1. Para o Ferret, os valores estão descritos na Seção 5.1.2

estão implementados os algoritmos. Em razão disso, é obrigatório que os *benchmarks* sejam compilados novamente. Essas questões são totalmente controladas pelo novo *script* *parsec_stream*, que verifica como os parâmetros devem ser aplicados, com base somente na linha de comando que foi utilizada. Em execuções consecutivas que utilizam os mesmos parâmetros ou que variam apenas alguns deles, o *script* verifica a necessidade da compilação, que torna o processo de utilização mais rápido.

As parametrizações mostradas até o momento refletem em diferentes comportamentos no Dedup e no Ferret. Para ajudar a entendê-las está disponível no Dedup e no Ferret o modo *tracing*, que coleta as métricas de *throughput* e latência. Esse modo é habilitado utilizando a *flag* `-trace`, que irá coletar as métricas calculadas com base em intervalos mínimos de um segundo (tempo padrão). O intervalo de tempo para a coleta pode ser informado após a *flag trace*, no formato de tempo em segundos. No entanto, coletar essas métricas pode gerar um *overhead* na execução, e, por isso, pode ser desabilitado com a *flag* `-notrace`. As parametrizações não acontecem em tempo de execução e devem ser realizadas antes da execução do Dedup e do Ferret.

5. RESULTADOS

Este capítulo expande detalhadamente as contribuições descritas na Seção 1.1, que são:

- **Identificação de características do paralelismo de *stream* que influenciam no comportamento de aplicações da suíte PARSEC.**
- **Uma análise de desempenho dos efeitos da parametrização em aplicações da suíte PARSEC baseada em um conjunto de experimentos.**

A proposta é apresentar os resultados obtidos com os *benchmarks* Dedup e Ferret, modificados a partir das versões POSIX Threads da suíte PARSEC. Todos os códigos estão em linguagem C. Estes *benchmarks* foram modificados para alcançar os requisitos de parametrização, com o objetivo de investigar diferentes comportamentos que a parametrização pode gerar no *benchmark*. Além disso, com as modificações realizadas, foi necessário obter diferentes métricas de desempenho. A seguir são demonstrados os resultados alcançados com este cenário modificado. São avaliados nos *benchmarks* Dedup e Ferret, as métricas de tempo de execução, *throughput*, latência e *service time*.

Inicialmente, é mostrado o cenário padrão de execução dos *benchmarks* na Seção 5.1. Em seguida, o cenário parametrizável de execução é apresentado na Seção 5.2. Em cada uma das seções, a metodologia utilizada nos testes é indicada. Para medir o tempo de execução foram obtidas 10 amostras para o cálculo de média e desvio padrão. Diferentemente, a caracterização do desempenho propõe representar uma execução real das aplicações, por isso uma única execução foi realizada para coletar as métricas de desempenho. Para a caracterização do desempenho, os *benchmarks* foram executados uma única vez, onde o *throughput*, a latência e o *service time* foram coletados.

A classe de entrada *Heavy Duty* foi utilizada para as execuções dos *benchmarks* (Capítulo 4). Detalhes das sub-classes do Ferret estão na Seção 4.2. Para as sub-classes do Dedup descritas na Seção 4.1, foram duplicados os arquivos em vista de aumentar a escala de entrada, mantendo um tamanho total aproximado entre as sub-classes. As classes de entradas para as execuções dos testes são apresentados na Tabela 5.1. O principal objetivo com a criação das sub-classes nessa proporção é aproximar o tamanho total dos conjuntos em cada sub-classe dos *benchmarks* Dedup e Ferret. No Dedup, foi possível manter um equilíbrio entre quantidade de arquivos e tamanho total do conjunto. Na sub-classe HS, para conseguir atingir aproximadamente 6 GB, foi necessário colocar mais arquivos, uma vez que o objetivo era reproduzir uma variação em taxa e tipo de entrada no Dedup. No Ferret, por causa das características das imagens, existe uma grande variação na quantidade de imagens em cada sub-classe.

Uma questão sobre as discussões dos resultados no decorrer deste capítulo deve ser enfatizada: as classes de entradas criadas são utilizadas para avaliar o impacto sobre o comportamento dos *benchmarks* ao processá-los. Portanto, o objetivo não é definir aspectos qualitativos dessas classes. Nosso intuito é representar diferentes cenários com tais classes e conseguir compreender o comportamento da aplicação ao processá-los.

Todos os testes foram executados em um servidor com dois processadores Intel Xeon E5-2620 v3 2.40 GHz, 12 núcleos e 24 *threads* (*Hyper-Threading*), 32 GB de memória RAM, sistema operacional Ubuntu Server 64 bits (kernel 4.4.0-121-generic). Essa infraestrutura é oferecida pelo LAD[1] (Laboratório de Alto Desempenho) da PUCRS. Os *benchmarks* foram compilados com o GCC 5.4.0 utilizando a *flag* -Os[2].

Tabela 5.1 – Configuração da Classe *Heavy Duty* para Dedup e Ferret.

	H1			**H2**			**H3**			**HS**	
	Número de Arquivos	Tamanho Total		Número de Arquivos	Tamanho Total		Número de Arquivos	Tamanho Total		Número de Arquivos	Tamanho Total
Dedup	15	6.2 GB		15	6.5 GB		15	6.3 GB		34	6.0
Ferret	13.750	719 MB		236	752 MB		*	*		6.906	717 MB
*Ferret não possuí conjunto H3. Formato dos arquivos: TAR para o Dedup. JPEG para as imagens no Ferret											

5.1 Cenário de Execução Padrão

O cenário de execução padrão utiliza os valores dos parâmetros equivalentes ao cenário de *benchmarks* do PARSEC (sem qualquer modificação). Os resultados destas execuções serão a linha de base para investigar os comportamentos nas execuções parametrizadas.

5.1.1 Dedup

O cenário padrão do Dedup foi executado com os seguintes parâmetros: Janela Deslizante (SW) 20 e *buffer* de tamanho 20. Os *buffers* com esse tamanho são de entrada e saída dos estágios. No estágio *fragment refine* (FR), apenas o *buffer* de saída é ajustado[3]. No *reorder* (RR), apenas o *buffer* de entrada, pois é um estágio da extremidade do *pipeline*. No algoritmo de fragmentação é utilizado o *Window Size* (WS) 32. Não foram encontradas explicações nas documentações do PARSEC sobre o motivo para a utilização desses valores.

[1] http://www.pucrs.br/ideia/laboratorios/lad/

[2] Otimização para tamanho. -Os ativa todas as otimizações de -O2 que normalmente não aumentam o tamanho do código. Ele também executa otimizações adicionais projetadas para reduzir o tamanho do código.

[3] Detalhes sobre esta configuração foram discutidos na Seção 4.3

A Figura 5.1 mostra a média do tempo de execução e o desvio padrão com as classes de entrada H1, H2, H3 e HS, executadas até 12 *threads*. Os resultados apontam o contraste do tempo de execução para cada tipo de entrada utilizada, indicando que o tipo de arquivo processado gera um comportamento diferente no Dedup.

O Dedup se comporta de forma escalável, processando todas as classes de entradas criadas. À medida que mais *threads* são alocadas, o tempo de execução diminui. Nas execuções com seis *threads*, o uso de *hyper-threading* penaliza o tempo de execução do Dedup em todas as classes.

Nas execuções com oito *threads*, existe uma variação no tempo de execução do Dedup. Essa variação é resultado da intensa comunicação entre as *threads* em execução, afetando, desse modo, o desempenho do Dedup. Com seis *threads*, esse impacto também é possível, devido às divergências no grau de paralelismo e *threads* alocadas, como explicado a seguir.

Passar uma grande quantidade de *chunks* entre os estágios exige uma comunicação entre as *threads*. Quando o grau de paralelismo é 1, cada estágio é executado por uma *thread*. Com esse grau de paralelismo, a comunicação tende a ser mais trivial para o *benchmark*. Porém, quando se aumenta esse grau de paralelismo, a comunicação pode ser um fator complicador durante a execução. Isso ocorre porque réplicas são criadas para os estágios que podem ser paralelizados[4] e cada réplica equivale a uma *thread*. Portanto, o grau de paralelismo definido como argumento antes da execução é menor que as *threads* que serão alocadas. No Dedup isso é obtido com o seguinte cálculo: *3 + 3n*, onde *n* é o número mínimo de *threads* definido pelo usuário. Por exemplo, com o grau de paralelismo em 12 *threads*, serão alocadas 39 *threads* no sistema operacional.

O comportamento do *benchmark* também é influenciado pelo conjunto de entrada utilizado. Um dos motivos está ligado ao algoritmo de fragmentação, pois ele define a granularidade de trabalho no Dedup, fragmentando o conjunto de entrada. Esse tipo de computação gera todos os *chunks* que serão processados durante a execução. Como existe uma imprecisão na fragmentação dos dados (problema discutido na Seção 4.1), os tamanhos dos *chunks* não serão estáticos e podem ter uma grande variação. Esse comportamento afeta principalmente o desbalanceamento de carga entre os estágios. O desbalanceamento é demonstrado também no trabalho de Navarro [NATC09]. Seus resultados com o Dedup do PARSEC revelam que nem todos os estágios processam os *chunks* gerados no estágio *fragment refine*. Por outro lado, os arquivos do tipo TAR também agravam tal problema. Esse tipo de formato influencia em quantidade e tamanho dos *chunks* quando são fragmentados [WDQ+12].

Os resultados também revelaram que os tipos de dados (nesse caso, as classes de entradas) impactam na quantidade dos *chunks* e no tamanho deles, devido ao algoritmo de fragmentação. Na Tabela 5.2, são mostrados os resultados do processamento do Dedup,

[4] *Fragment Refine (RR). Deduplication (DD), Compression (COMP)*

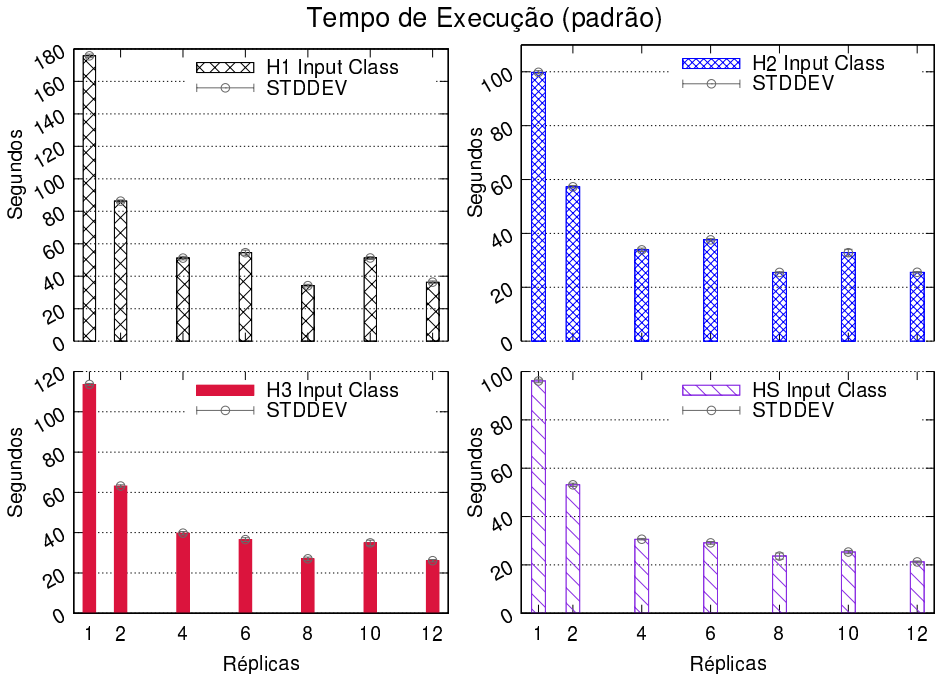

Figura 5.1 – Média do tempo de execução do Dedup.

sendo possível ver o tamanho médio dos *chunks*, desvio padrão e quantidade de *chunks* gerados para cada classe de entrada.

A caracterização do desempenho permite analisar com mais profundidade os aspectos discutidos nos parágrafos anteriores. A Figura 5.2 mostra a caracterização do desempenho com as métricas de *throughput* e *service time* com as entradas H1, H2, H3 e HS. No Dedup, o *throughput* é calculado com base no tamanho dos dados de cada *chunk*. Obter essa métrica com base na quantidade de *chunks* pode gerar uma informação errada sobre o desempenho, devido às questões associadas ao algoritmo de fragmentação mostradas anteriormente. No cenário de execução parametrizado (Seção 5.2), isso será evidenciado com mais detalhes.

Antes de compreender os resultados de caracterização, é necessário examinar os dados contidos na Tabela 5.2. Os valores da tabela não fazem distinção entre os arquivos e representam valores absolutos de uma execução do Dedup. Por exemplo, a quantidade de *chunks* duplicados na classe H1 exibido na tabela, é o total dos *chunks* duplicados dos 15 arquivos. Essa mesma regra vale para todas as classes de entrada e métricas da tabela.

Com base no número de *chunks*, observa-se que para cada tipo de entrada no Dedup, haverá um comportamento diferente de fragmentação. Com a classe H2, que é maior em relação às outras, a quantidade de *chunks* foi inferior. Na execução com o HS, o tamanho da entrada é menor que H2, porém, percebe-se uma quantidade de *chunks* superior. Para as classes que representam dados multimídias (imagens com H1, vídeos e áudios com H3), o comportamento de fragmentação foi semelhante entre eles, considerando a quantidade de *chunks*. No entanto, é importante observar a existe diferença de

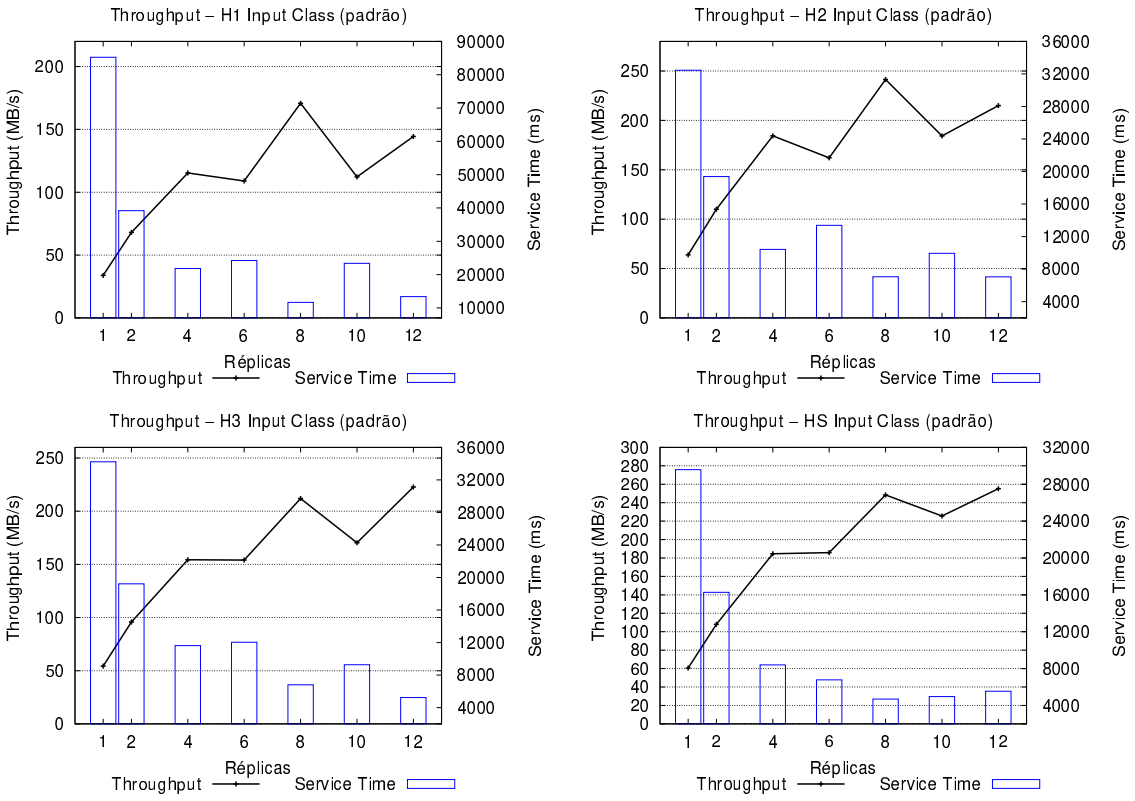

Figura 5.2 – Caracterização do desempenho do Dedup com as sub-classes H1, H2, H3 e HS. As métricas de desempenho são *throughput* e *service time*.

aproximadamente 100 MB no tamanho total das classes que pode ter causado a variação na quantidade de *chunks*.

As diferenças na quantidade de *chunks* é explicada pelo tamanho médio dos *chunks*. Na classe H2, os *chunks* foram maiores e com uma variação pequena. Tal comportamento ocorre com a classe HS. Na H3 existe uma grande variação no tamanho dos *chunks*, demonstrada pelo desvio padrão.

Anteriormente, destacaram-se a granularidade de trabalho do Dedup e, a partir disso, foi possível compreender o impacto que tal fator gera sobre o *throughput* e o *service time*. O tamanho da granularidade está associada a aspectos como a taxa de comunicação e a eficiência do processamento. Granularidades pequenas exigem mais comunicações entre os estágios, gerando uma condição de *locks contention* entre as *threads* das réplicas. Além disso, pouco tempo de processamento é investido sobre cada *chunk*, por serem processados mais rapidamente. Por esses motivos, as taxas de *throughput* foram menores e *service time* mais alto com H1 e H3. Por outro lado, com H2 e HS, que alcançaram tamanhos de *chunks* maiores, seu desempenho nessas métricas foram melhores.

O processamento de deduplicação demonstrou ser mais eficiente com a subclasse H3. A porcentagem de *chunks* duplicados foi maior em relação as outras subclasses. Isso ocorre por causa do tamanho dos vídeos utilizados para criar os arquivos na H3. Um desses arquivos, possui apenas dois vídeos empacotados para alcançar 500

MB de tamanho, aproximadamente. Desse modo, como 10% do conteúdo desses arquivos foram duplicados, resultaram somente em dois arquivos. Portanto, o Dedup considera todos os *chunks* duplicados para tal arquivo, pois uma tabela *HASH* é exclusiva para cada arquivo do conjunto [5].

Tabela 5.2 – Resultado do processamento com o Dedup no cenário de execução padrão.

Conjunto de Entrada	Tamanho da Entrada	*Chunks*	*Chunks* Duplicados	Tamanho Médio (KB) / Desvio Padrão (KB)	Tamanho da Saída: Após deduplicação (GB) / Após compressão (GB)
H1	6.2 GB	1653279	96915(5.86%)	3.91 / 5.11	5.98 / 5.86
H2	6.5 GB	1436500	321811 (22.40%)	4.70 / 4.95	4.29 / 3.69
H3	6.3 GB	1645041	518832 (31.54%)	3.98 / 5.13	4.34 / 4.31
HS	6.0 GB	1477071	389801(26.39%)	4.23 / 4.93	4.19 / 3.98

A Figura 5.2 mostra que processar as entradas H1 e H3 (arquivos de mídia) influencia no comportamento do *benchmark*, onde as taxas de *throughput* são menores que o processamento com entrada H2 (arquivos de texto). A fragmentação de arquivos de vídeos e grandes imagens é custosa para o Dedup e gera uma quantidade maior de *chunks*. Além disso, isso impacta no processo de verificação dos *chunks* duplicados, pois *chunks* de um único vídeo fragmentado nem sempre são duplicados. Analisar com profundidade tais questões de fragmentação não faz parte do escopo do presente trabalho.

Com a métrica de latência, o impacto da comunicação no Dedup é analisado com mais detalhes. Na Figura 5.3 é mostrada a latência entre os estágios em escala logarítmica, com exceção da latência entre o estágio *fragment refine* e *deduplication* (FR->DD).

A latência alta entre o estágio *deduplication* e *compression* (DD->COMP) indica que o tempo de computação nesse estágio é maior do que nos outros. Mesmo com o aumento do número de réplicas, a maior parte da computação do Dedup se mantém no estágio de compressão, pois existe pouco impacto na latência à medida que mais réplicas são utilizadas. Isso mostra que existe um desbalanceamento de carga entre os estágios do Dedup.

O problema de desbalanceamento é abordado no trabalho de Navarro [NATC09], que afirma que o desbalanceamento de carga no Dedup é causado pelo estágio *fragment refine*, que processa todos os *chunks*. No entanto, a implementação do Dedup nesta dissertação é diferente do cenário adotado por Navarro. Com a nova implementação do Dedup, o problema que acontecia no estágio de fragmentação foi amenizado, porém surge no estágio de compressão. Os resultados de latência mostraram que a maior computação está no estágio de compressão para a nova implementação do Dedup.

[5] Mais detalhes sobre o processo de Deduplicação estão no Capítulo 4, na Seção 4.1.

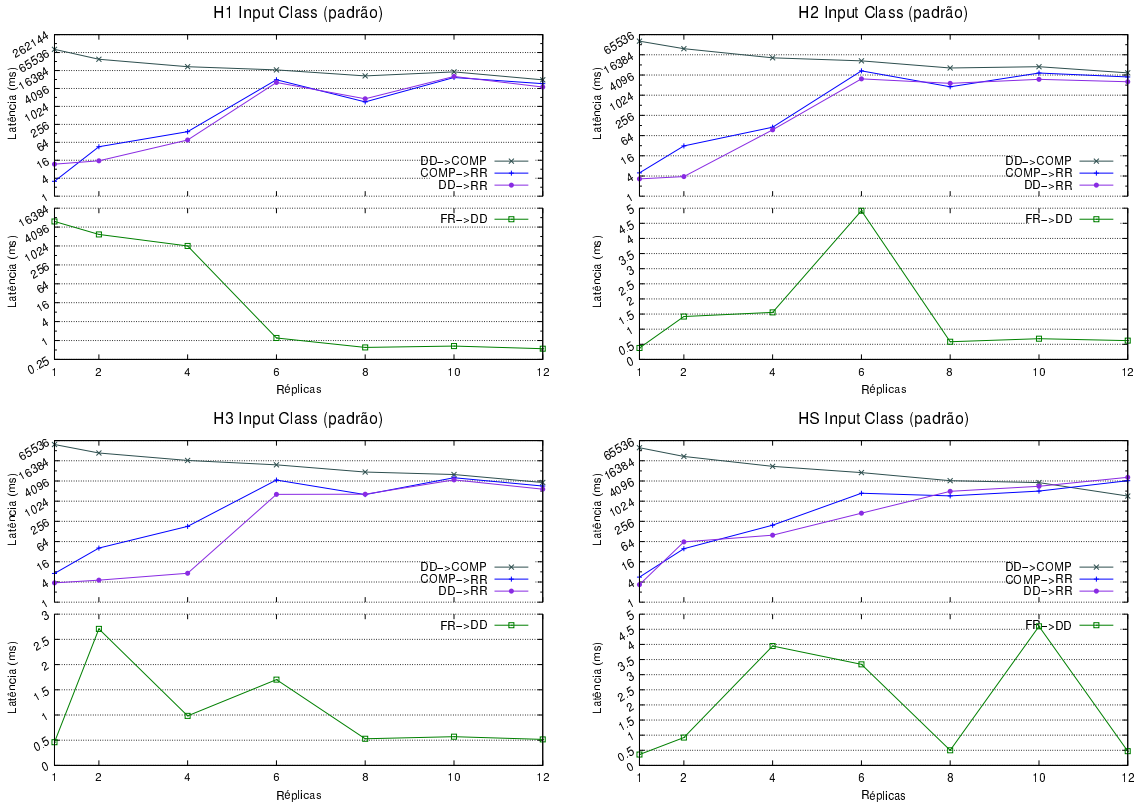

Figura 5.3 – Caracterização do desempenho com as sub-classes H1, H2, H3 e HS.

5.1.2 Ferret

Para a execução do Ferret, foram definidas as configurações padrões utilizadas nos testes com a entrada *native* do PARSEC [Bie11]. As filas são de tamanho 20 (posições) e a quantidade de imagens similares classificadas é 50. No algoritmo de segmentação, o parâmetro *Q value* é definido em 128 e o *threshold* em 0.005.

Nas documentações do PARSEC, os tamanhos das filas e os valores do algoritmo de segmentação não são justificados. A quantidade de imagens similares é um valor particular da entrada *native* do Ferret no PARSEC. Essa entrada foi criada na suíte para as execuções dos *benchmarks* representarem um cenário computacional real, sendo o conjunto mais complexo que os restantes (*test,simdev,simsmall,simmedium* e *simlarge*).

A Figura 5.1 mostra os resultados de tempo de execução e desvio padrão com as novas classes de entrada, todos processados com as configurações padrões do Ferret. Os resultados mostram que o Ferret tem um comportamento escalável com as classes H1 e HS até oito réplicas. Com a classe H2, o tempo de execução é irregular, onde o melhor caso de execução foi com uma *thread*. Antes de entender essa irregularidade, discutiremos os comportamentos com as classes H1 e H2.

Alguns trabalhos na literatura debatem o problema da escalabilidade existente no Ferret. Por exemplo, em [CCM+15] é avaliado o impacto de paralelismo de tarefas

em 11 *benchmarks* da suíte PARSEC. Os resultados das execuções mostram a melhora no desempenho em alguns *benchmarks*, com exceção do Ferret que manteve o mesmo *speed-up* e escalabilidade da implementação original. Outras pesquisas buscam contornar esse problema de escalabilidade ajustando diferentes graus de paralelismo em cada estágio [PGB11]. Todos esses trabalhos executaram seus testes com a maior entrada oferecida pelo PARSEC para o Ferret.

Com as três classes de entradas, criadas para este trabalho, o problema de escalabilidade se manteve. As classes foram criadas com a intensão de aumentar o conjunto de trabalho no Ferret e atuar principalmente no tamanho da granularidade. Foi discutido anteriormente na Seção 4.2, que a quantidade de trabalho oferecida pelo PARSEC pode ser insignificante para as atuais arquiteturas de processamento. Portanto, com as novas classes foi possível compreender que fatores adicionais comprometem o desempenho do Ferret e, com a caracterização do desempenho, os problemas ficaram mais evidentes, como mostrado nos parágrafos seguintes.

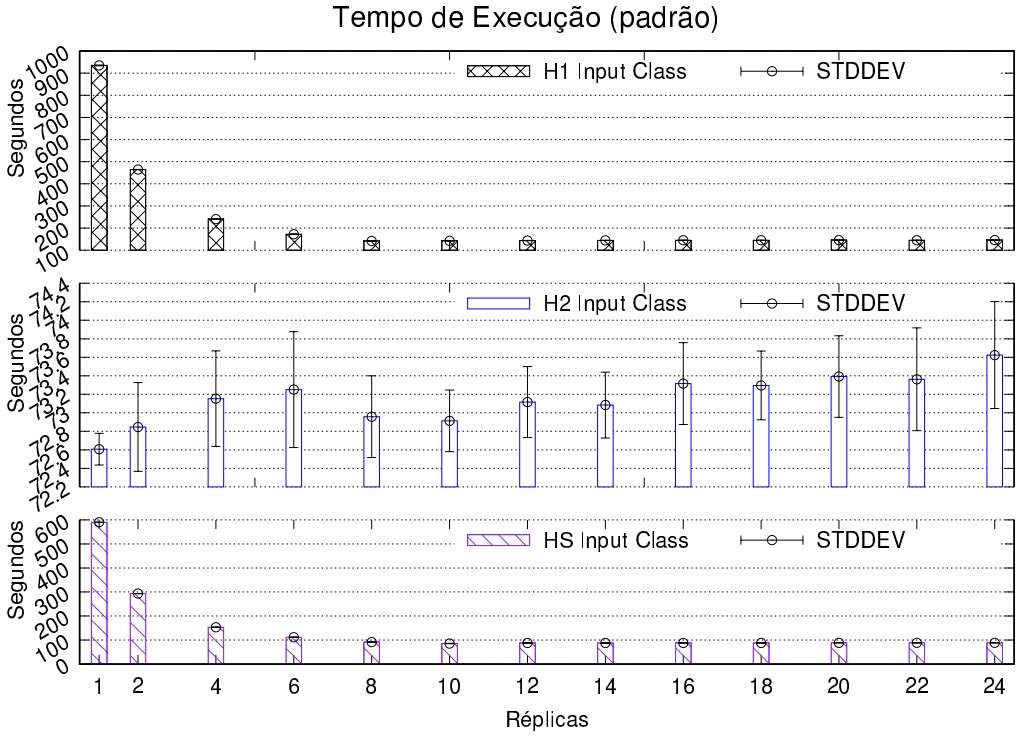

Figura 5.4 – Média do tempo de execução do Ferret.

A caracterização do desempenho forneceu mais detalhes para investigar o comportamento do Ferret. A Figura 5.5 mostra os resultados do *throughput* (imagens por segundo) e *service time*. Com as classes H1 e H2, essas métricas permanecem lineares a partir de oito e 10 réplicas. Tal comportamento linear do *throughput* é semelhante ao encontrado em [RKO+11], que utiliza essa métrica para adaptar o grau de paralelismo nas aplicações utilizando um orquestrador chamado DoPE.

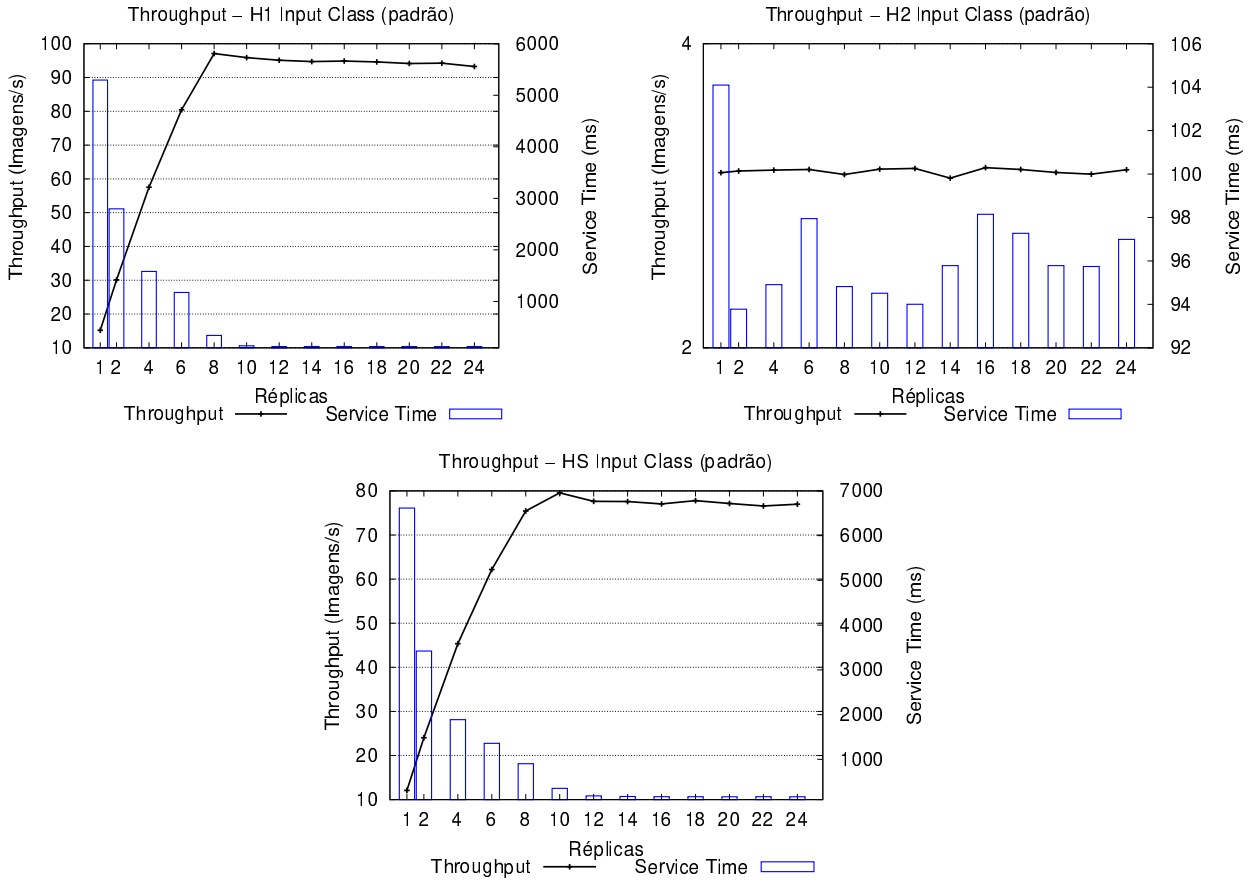

Figura 5.5 – Caracterização do desempenho com as sub-classes H1, H2 e HS no Ferret.

Na Figura 5.6 pode ser observada a latência medida com as execuções das classes H1, H2 e HS, plotada em escala logarítmica, com exceção da latência entre estágio *rank* e *out* (*Rank->Out*). Com a métrica de latência, é possível analisar com mais detalhes o comportamento de comunicação do Ferret e compreender o problema de escalabilidade envolvido no *benchmark*. O número de réplicas é um fator que está associado à latência e pode ser observado nas execuções com H1 e HS. A latência tende a um comportamento sempre linear, sendo maior com poucas réplicas e menor quando mais réplicas são utilizadas. As oscilações aparecem a partir de oito réplicas, por causa das questões que envolvem a escalabilidade do Ferret.

Diferentes fatores estão envolvidos com o problema da escalabilidade do Ferret. Um deles é a quantidade de computação que o estágio *rank* realiza. A maior parte da computação se concentra nesse estágio e, desse modo, o ganho de desempenho no Ferret fica limitado a ele. Outro fator está no lento processo de segmentação das imagens do estágio *segmentation*. Esse segundo motivo fica mais evidente quando as imagens com resoluções maiores (sub-classe H2) são utilizadas para o processamento. Os problemas envolvidos no estágio *rank* e na segmentação que acontece no estágio *seg* também foram demonstrados em outras pesquisas que envolvem o Ferret [PGB11, Lv06]. Embora

os trabalhos utilizem um cenário diferente, os resultados da caracterização com a versão parametrizável demonstra detalhadamente que os problemas se mantêm.

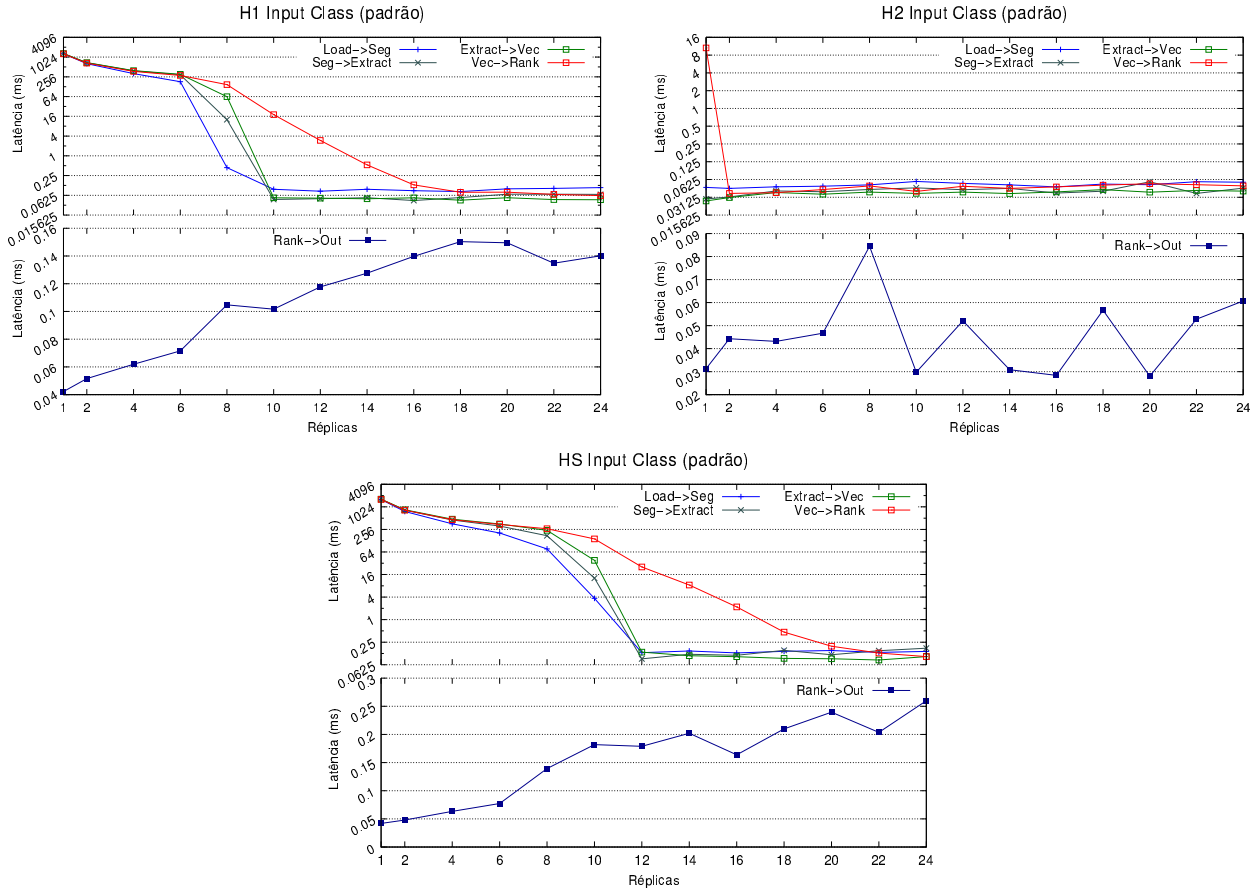

Figura 5.6 – Latência do Ferret com as sub-classes H1, H2 e HS.

Dadas as demonstrações anteriores envolvendo as classes H1 e HS, os problemas relacionados à classe H2 ficam mais evidentes. Inicialmente, o tempo de execução tem um desempenho irregular. Por outro lado, ao observar os resultados de latência, percebe-se que o tempo de comunicação não foi influenciado pelo número de réplicas como acontece com as outras classes; inclusive o tempo de comunicação foi mais rápido. Porém, esses resultados de latência indicam uma falsa impressão de desempenho. Isso acontece porque é alto o tempo de segmentação das imagens dessa classe em relação às imagens das demais classes. Lv [Lv06] afirma que o tempo de segmentação é alto também para imagens com resoluções pequenas.

Portanto, a baixa latência indica que os elementos não aguardam um longo período nas filas, uma vez que no instante que são inseridos, existem réplicas aguardando por computação. A Figura 5.7 confirma esse problema mostrando as *threads* das réplicas que foram alocadas durante um teste com o Ferret. O teste foi executado utilizando a classe H2

com grau de paralelismo 8[6]. Nesse teste, o grau de paralelismo definido foi oito *threads*, pois foi o melhor cenário de escalabilidade alcançado com as outras classes.

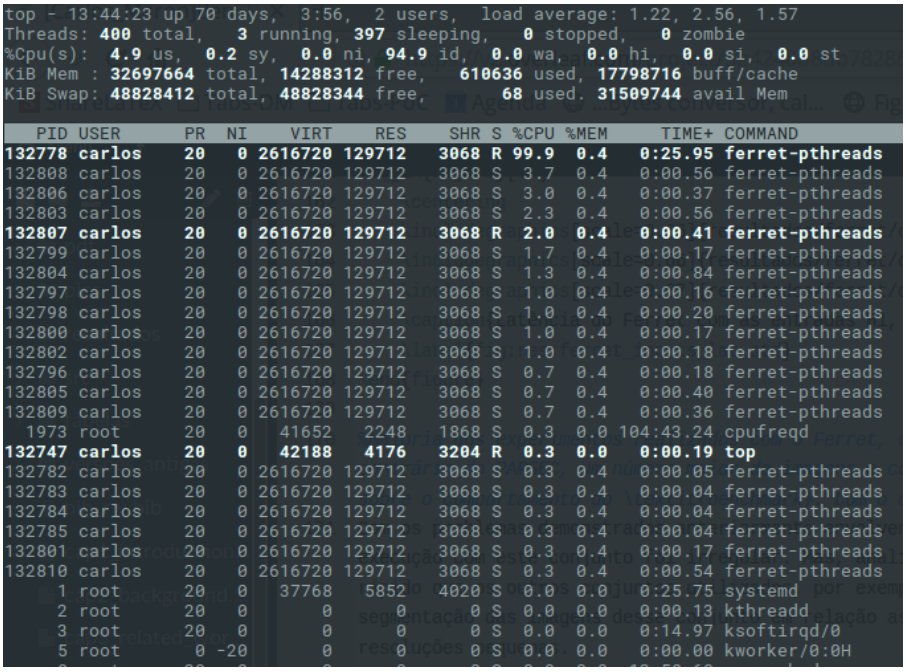

Figura 5.7 – Lista de réplicas do Ferret alocadas no sistema operacional. A coluna **S** indica o estado (*State*) dos processos/*threads* alocados no sistema. Nesta coluna, réplicas com a letra **S** indicam *sleeping* (aguardando por computação). Réplicas com a letra **R** indicam *Running* (executando alguma computação).

5.2 Cenário de Execução Parametrizado

Nas seções anteriores foram apresentados os resultados dos *benchmarks* Dedup e Ferret, executados em seu cenário padrão, que são os valores dos parâmetros empregados pelo PARSEC em seu formato original. Nesse sentido, discutimos os comportamentos dos *benchmarks*, mostrando os impactos e as limitações para essas execuções.

Nesse trabalho foram realizadas diversas modificações nos *benchmarks* Dedup e Ferret. Além disso, novas classes de entradas foram criadas para cada um deles, diferentes das existentes na suíte PARSEC para tais *benchmarks*. Essas questões são discutidas detalhadamente no Capítulo 4.

Nesta seção é explorado o cenário parametrizado para avaliar o comportamento desses *benchmarks*, executando com as novas entradas e utilizando diferentes valores para os parâmetros. Será mostrado como o ajuste de certos parâmetros impactam no tempo de

[6] A execução do Ferret foi monitorada com o utilitário *top* do Linux, um visualizador de processos interativo, que mostra em tempo real os processos em execução. A figura representa uma captura de tela durante um teste do Ferret. O utilitário foi executado com o seguinte comando: `$ top -H`.

execução, *throughput*, latência e no *service time*. Além disso, ressalta-se a facilidade para parametrizar as execuções e alcançar diferentes comportamentos.

Na Seção 5.2.1, os resultados com o Dedup são apresentados, seguido pelos resultados do Ferret na Seção 5.2.2. O cenário de execução padrão (Seção 5.1) são a base para a discussão dos resultados desta seção.

5.2.1 Dedup

O Dedup foi executado com as seguintes características parametrizadas: janela deslizante (SW) com valor 40; *buffer* com valor 40, e *Window Size* (WS) com tamanho 2048. Os conjuntos de entrada foram os mesmos utilizados no cenário padrão de execução, descritos na Seção 5.1.1.

Para chegar a tais valores, o Dedup foi executado variando esses parâmetros diferentes vezes. A partir disso, foram analisados cada um dos cenários, levando em conta principalmente a caracterização de desempenho (latência e *throughput*) e selecionando apenas um dos cenários para a discussão.

Na Figura 5.8 é mostrada a comparação dos resultados de tempo de execução entre os cenários padrão e o parametrizado (destacado). Para esse teste, o Dedup foi executado com a linha de comando a seguir:

```
$ parsec_stream -p dedup -t 12 -i h2 -chunk 2048 \
              -fr 40 -dd 40 -comp 40 -rr 40 \
              -notrace
```

A parametrização teve efeito positivo no tempo de execução, garantindo uma execução mais rápida para o Dedup. Isso acontece devido ao processamento mais eficiente do algoritmo de fragmentação e aos *buffers* maiores. Os *buffers* podem ser vistos como “filas” exclusivas de cada réplica, e aumentar o tamanho deles evita o acesso frequente nas filas que são compartilhadas entre as réplicas do estágio. O acesso às filas é controlado por *locks* (primitivas de sincronização) para garantir o ordenamento de cada *thread* na manipulação de dados em uma região de memória compartilhada. Por outro lado, não é necessário esse controle para acessar os *buffers*.

A diminuição do tempo de execução também está associada à diminuição dos *chunks*, devido à parametrização do algoritmo de fragmentação no estágio FR. O *chunk* representa a granularidade de trabalho para cada *thread*. Reduzindo a quantidade de trabalho, consequentemente, irá reduzir aos estágios. Na Tabela 5.3, é mostrado o resultado do processamento com o Dedup parametrizado, que revela uma redução significativa na quantidade dos *chunks*. Essa redução é resumida na Tabela 5.4.

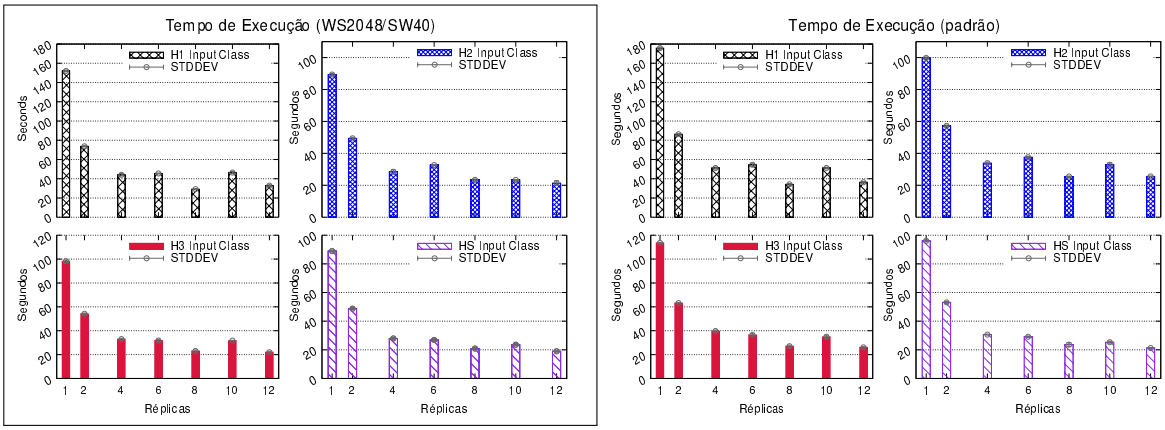

Figura 5.8 – Tempo de execução do Dedup. A figura em destaque representa os resultados do cenário de execução parametrizado.

Tabela 5.3 – Resultado do processamento com o Dedup no cenário de execução parametrizado.

Conjunto de Entrada	Tamanho da Entrada	*Chunks*	*Chunks* Duplicados	Tamanho Médio (KB) / Desvio Padrão (KB)	Tamanho da Saída: Após deduplicação (GB) / Após compressão (GB)
H1	6.2 GB	1078412	12566(1.17%)	6.00 / 4.13	6.10 / 5.97
H2	6.5 GB	1156488	410805 (35.52%)	5.84 / 3.76	4.28 / 3.65
H3	6.3 GB	1092186	309279 (28.23%)	6.00 / 4.16	4.48 / 4.44
HS	6.0 GB	1055706	294413(27.89%)	5.92 / 3.87	4.37 / 4.15

No Dedup existe uma relação entre a quantidade e o tamanho dos *chunks*, que pode ser explorada parametrizando o algoritmo de fragmentação (BSW). Essa relação é observada nas Tabelas 5.3 e 5.2. Quando o tamanho da janela no algoritmo é maior, a quantidade de *chunks* diminui, porém aumenta o tamanho de cada *chunk*. Essa relação de quantidade *versus* tamanho do *chunk* também está associada à redução do tempo de execução. Isso acontece porque a computação fica mais eficiente devido ao aumento do tamanho do *chunk*. Por exemplo, o *chunk* maior exige mais tempo do processador para gerar as chaves HASH no estágio *deduplication* ou para comprimir os dados do *chunk* no estágio *compression*. Além disso, outras operações de processamento também diminuem, como acesso à memória, acesso às filas, e em especial as operações de E/S do disco.

O Dedup é um *benchmark* com um comportamento de E/S intensivo. No primeiro estágio existem as requisições de acesso ao disco para ler os arquivos de entrada e alocá-los em memória. No último estágio, as requisições de acesso ao disco ocorrem para cada *chunk* gerado na aplicação. Desse modo, as requisições acompanham essa diminuição da quantidade de *chunks* e reduz a sobrecarga de E/S no estágio *reorder*.

As Figuras 5.9, 5.10, 5.11 e 5.12 mostram os resultados de *throughput* e *service time* com as classes de entrada H1, H2, H3 e HS, comparados com os resultados do cenário padrão. Em todos os cenários comparados, a parametrização contribuiu para o *benchmark* alcançar maiores taxas de *throughput* e a diminuição do *service time*. Para essa caracterização do desempenho, um exemplo da linha de comando utilizada é mostrado a seguir. Os

Tabela 5.4 – Redução da quantidade de *chunks* no cenário parametrizado

Conjunto de Entrada	Redução do Número de *Chunks*
H1	34.77%
H2	19.49%
H3	33.60%
HS	28.52%

parâmetros são: *benchmark* Dedup executando com 12 *threads*, *chunk* 2048 e os *buffers* com valor 40. Para esta execução o modo *trace* está habilitado e o tempo é 1 segundo.

```
$ parsec_stream -p dedup -t 12 -i h2 -chunk 2048 \
                -fr 40 -dd 40 -comp 40 -rr 40 \
                -trace 1
```

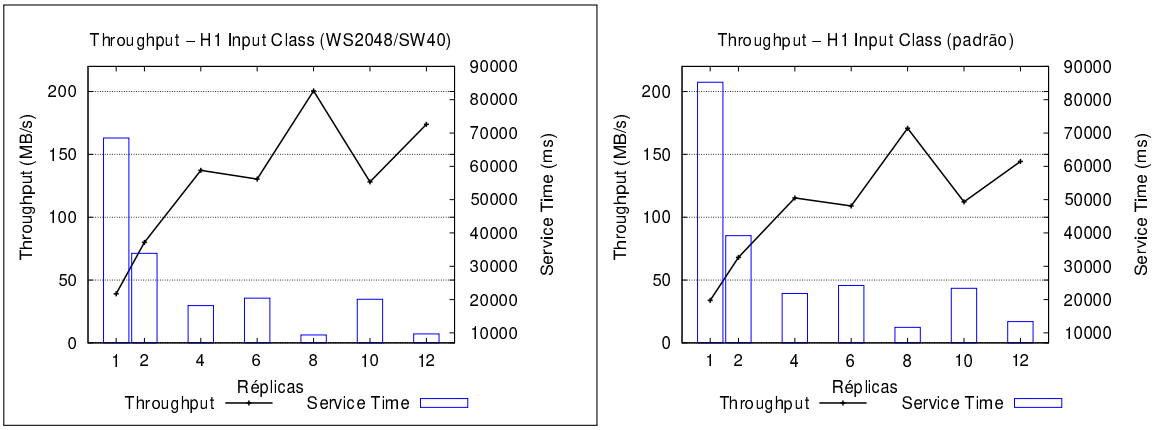

Figura 5.9 – Caracterização do desempenho com a sub-classe H1 no Dedup.

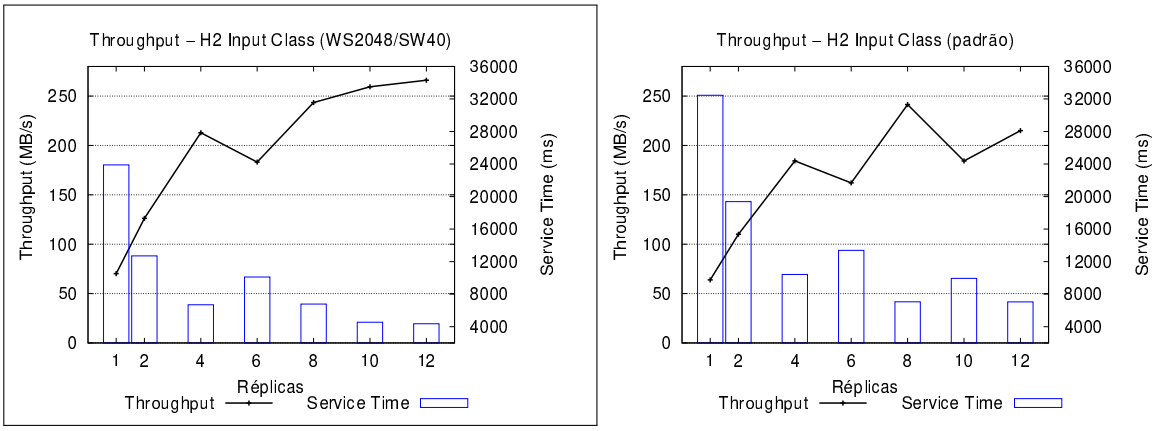

Figura 5.10 – Caracterização do desempenho com a sub-classe H2 no Dedup.

Nos resultados do processamento com a classe H3, o tamanho total da saída (soma do tamanho de todos os arquivos) no cenário parametrizado foi maior que no cenário

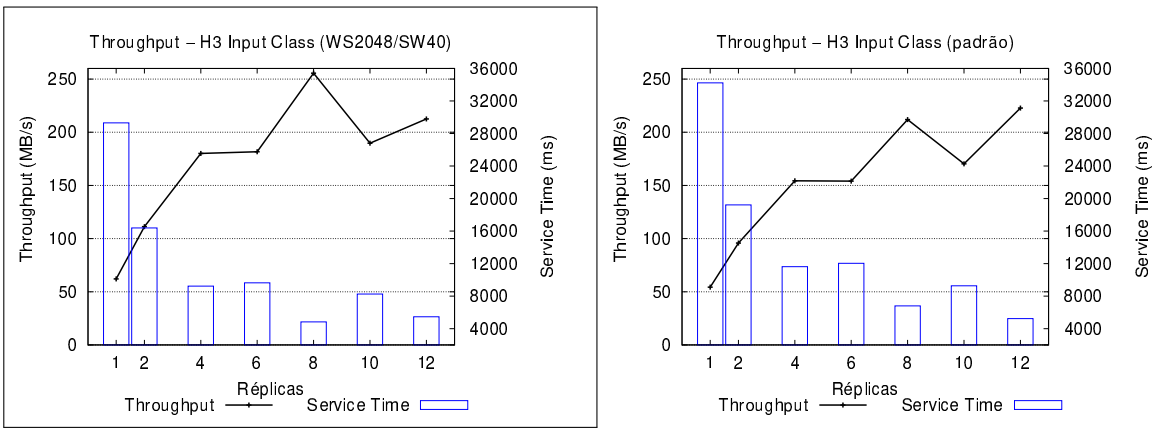

Figura 5.11 – Caracterização do desempenho com a sub-classe H3 no Dedup.

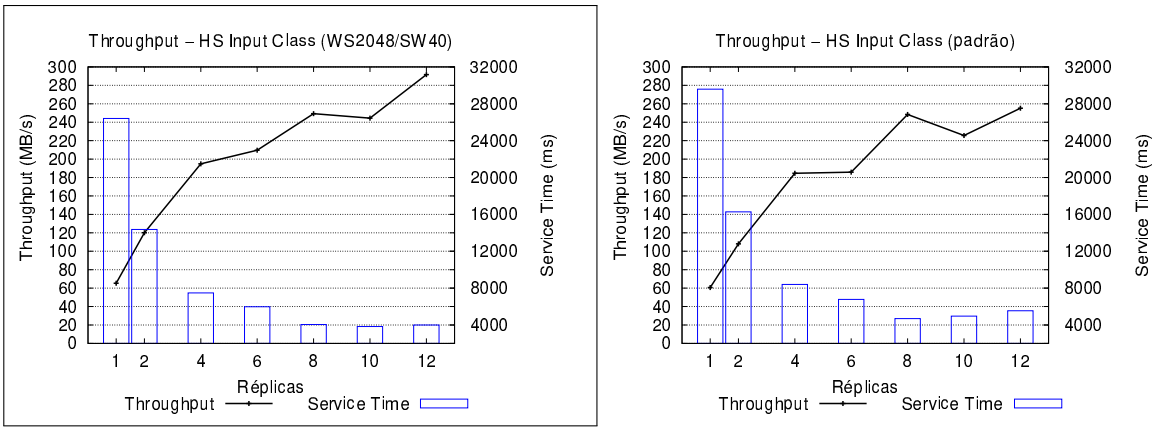

Figura 5.12 – Caracterização do desempenho com a sub-classe HS no Dedup.

padrão. Isso ocorre porque houve a diminuição dos *chunks* duplicados no cenário parametrizado (31.23% no cenário padrão contra 28.23% no cenário parametrizado). Quando um *chunk* é identificado como duplicado, apenas sua referência ao *chunk* original é gravado em disco e essa referência ocupa um espaço menor que o *chunk* original.

Os arquivos gerados ao final das execuções foram validados com o próprio Dedup. Essa validação é o processo inverso da deduplicação, que gera o arquivo original baseado no arquivo já processado (com a deduplicação realizada). Essa validação mostrou que a implementação garantiu o processamento íntegro e não gerou arquivos corrompidos.

As Figuras 5.13, 5.14, 5.15 e 5.16 mostram a comparação dos resultados de latência entre o cenário parametrizado (destacado) e o padrão. A métrica é plotada em escala logarítmica, com exceção da latência entre os estágios *fragment refine* e *deduplication* (FR->DD). O mesmo se aplica para a classe H1 do cenário padrão na Figura 5.13.

Os resultados de latência indicaram um gargalo durante a comunicação entre os estágios *fragment refine* e *deduplication* (FR->DD) no cenário de execução padrão, mostrando que o estágio de fragmentação realiza uma intensa computação no *benchmark*. Esse resultado confirma mais uma vez o que foi defendido por Navarro [NATC09] e explicado an-

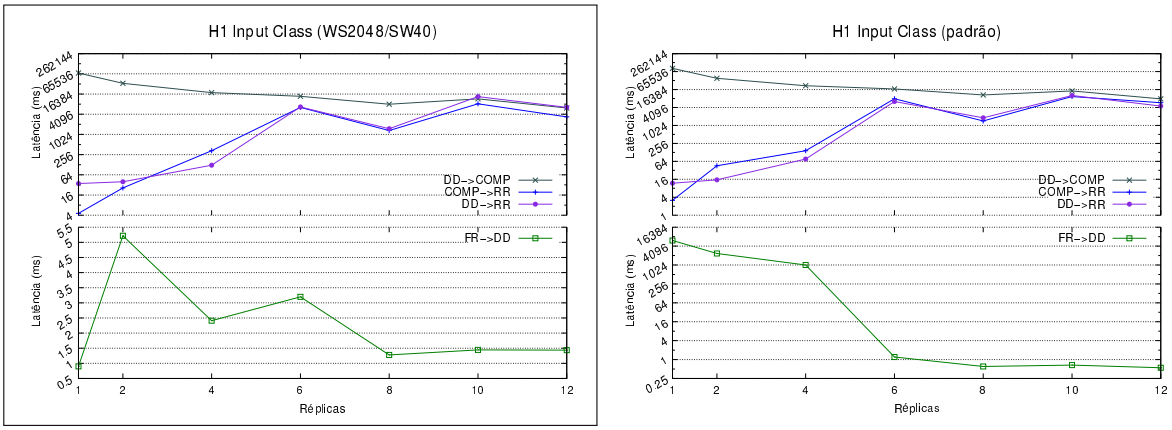

Figura 5.13 – Caracterização do desempenho com a sub-classe H1 no Dedup.

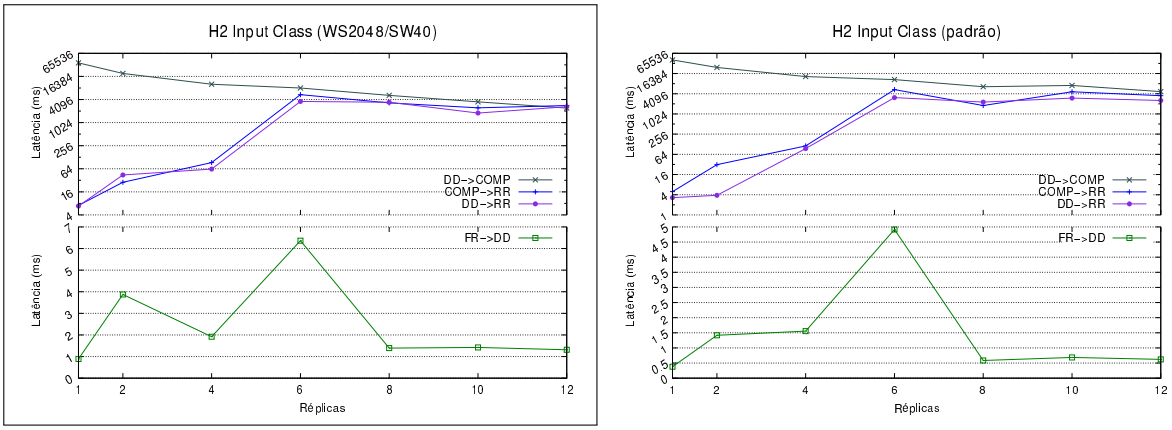

Figura 5.14 – Caracterização do desempenho com a sub-classe H2 no Dedup.

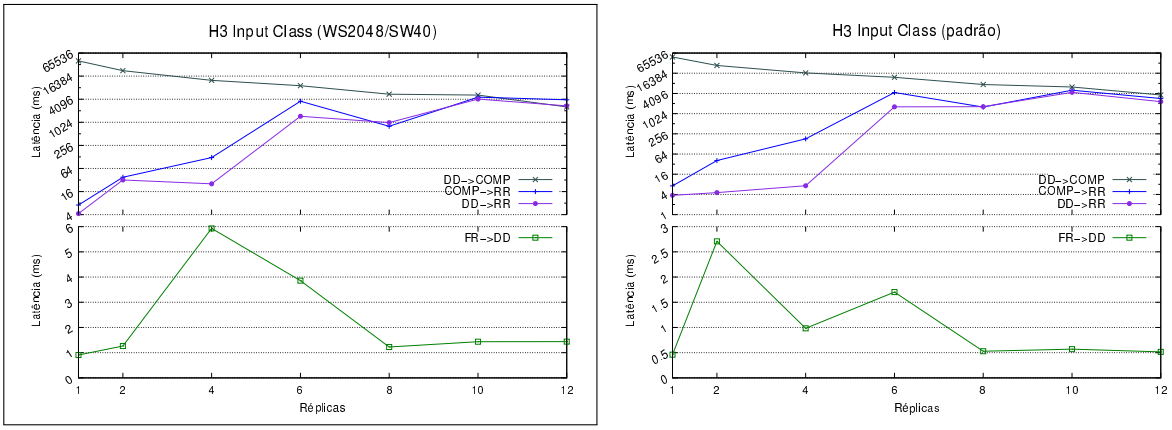

Figura 5.15 – Caracterização do desempenho com a sub-classe H3 no Dedup.

teriormente na Seção 5.1.1. Com a parametrização dos *buffers*, o tempo de comunicação foi melhorado nos testes do cenário parametrizado.

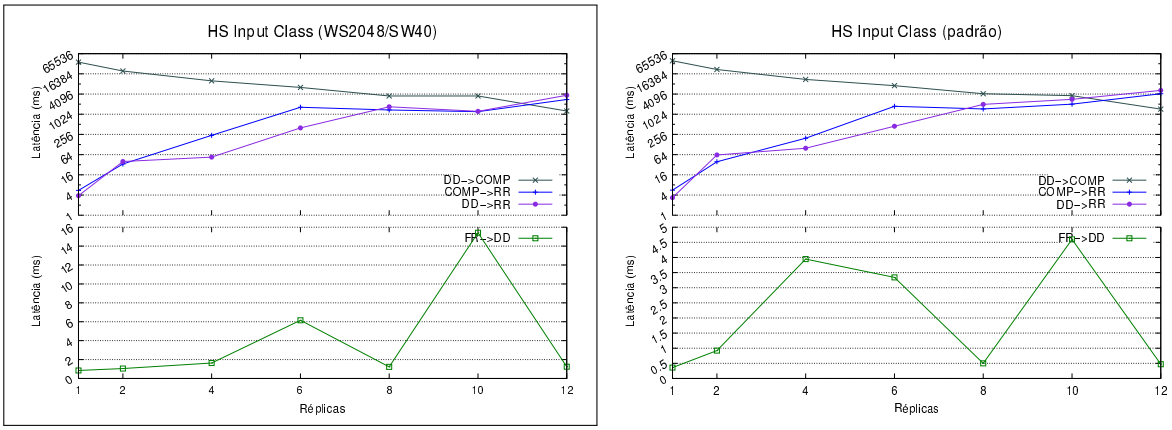

Figura 5.16 – Caracterização do desempenho com a sub-classe HS no Dedup.

5.2.2 Ferret

No cenário parametrizado foi modificado o tamanho das filas para 5. Assim como no cenário padrão, também foram utilizadas 50 imagens para classificar as mais similares e os mesmos valores para o algoritmo de segmentação, *Q value* 128 e *threshold* 0.005.

Várias execução foram realizadas para definir o tamanho das filas. Foi observado que a modificação do algoritmo de segmentação tem impacto significativo no tempo de execução e na precisão da busca de similaridade. Quando o parâmetro *Q value* aumenta, mais regiões são geradas durante a segmentação, assim mais regiões são comparadas durante a busca de similaridade. Desse modo, o tempo é penalizado durante a segmentação no estágio *segmentation*, na pré-classificação do estágio *indexing* e na classificação do estágio *rank*. Por esse motivo, a qualidade da classificação das imagens não será analisada e pela grande quantidade de imagens executadas (20.892 somando as classes H1, H2 e HS).

Os resultados de latência no cenário de execução padrão (Seção 5.1) indicaram que os elementos aguardam determinado período nas filas. Com base nisso, o Ferret foi executado com filas maiores, acreditando que o tempo de execução teria algum impacto positivo. Nessa tentativa, a espera das réplicas por posições livres nas filas seria menor. Com essa situação, a seguinte analogia pode ser feita: um estágio não consegue colocar elementos na fila do próximo estágio, pois não existem posições livres. Da mesma forma que não consegue colocar, o estágio não consegue retirar elementos da fila de entrada, pois está aguardando para liberar o resultado da computação. Isso poderia ser desencadeado para os outros estágios e a normalização estaria a cargo do estágio *rank*. Porém, aumentar o tamanho das filas não teve efeito positivo ou negativo no tempo de execução.

Pode ser observado na Figura 5.17 (em destaque), que filas menores também não impactaram significativamente no tempo de execução do Ferret. Houve variação somente com a classe H2, porém, com o desvio padrão alto não é possível inferir qual execução foi melhor. A seguir é mostrado um exemplo da linha de comando utilizada para executar o Fer-

ret. Os parâmetros são: *benchmark* Ferret executado com 12 *threads*, *precision* 128, filas com valor 5 e o parâmetro *ranking* 50. Para essa execução, o modo *trace* está desabilitado.

```
$ parsec_stream -p ferret -t 12 -i h2 -precision 128 \
                  -seg 5 -ext 5 -idx 5 -rank 5 \
                  -ranking 50 \
                  -notrace
```

O motivo desse comportamento foi investigado com as métricas de latência, observado nas Figuras 5.18, 5.19 e 5.20. As figuras em destaque mostram a redução da latência no cenário parametrizado. No entanto, é observado que o comportamento do *benchmark* no cenário parametrizado é próximo do cenário padrão. Os resultados estão em escalas logarítmicas, com exceção da latência entre estágio *rank* e *out* (*Rank->Out*).

```
$ parsec_stream -p ferret -t 12 -i h2 -precision 128 \
                  -seg 5 -ext 5 -idx 5 -rank 5 \
                  -ranking 50 \
                  -trace 1
```

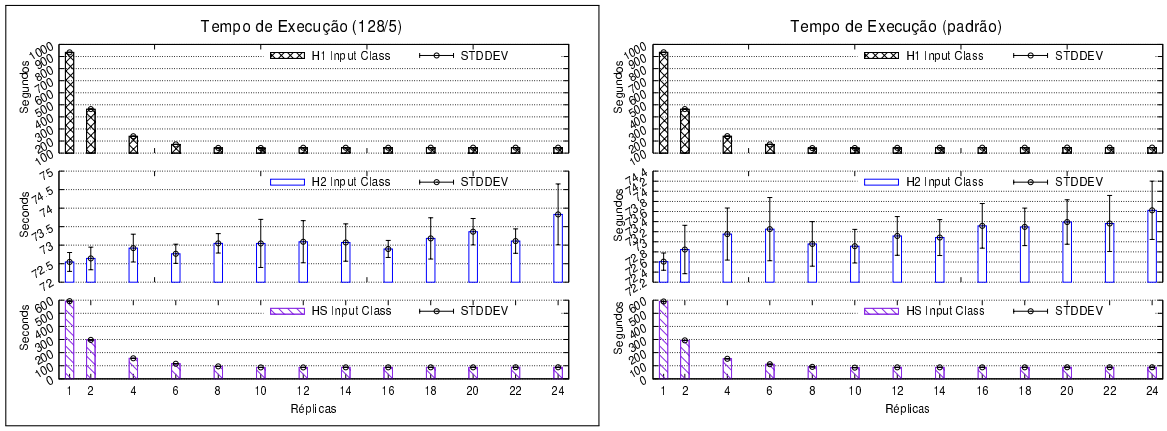

Figura 5.17 – Média do tempo de execução do Ferret. A figura em destaque representa o cenário de execução parametrizado.

As questões discutidas na Seção 5.1 sobre o desempenho não se alteraram com o cenário parametrizado. Com os resultados das execuções parametrizadas foi possível analisar o Ferret por outro cenário, que, mesmo parametrizado, apresentou comportamentos semelhantes ao cenário padrão. Por exemplo: a parametrização refletiu na latência menor entre os estágios, mas tal resultado não melhorou o tempo de execução do *benchmark*. Desse modo, além das limitações de segmentação e o processamento do estágio *rank*, as requisições de E/S no primeiro estágio também são um agravante ao *benchmark*.

Com a classe H2, os problemas também ficam evidentes no cenário parametrizado. A baixa latência observada na Figura 5.19 é uma falsa impressão de desempenho.

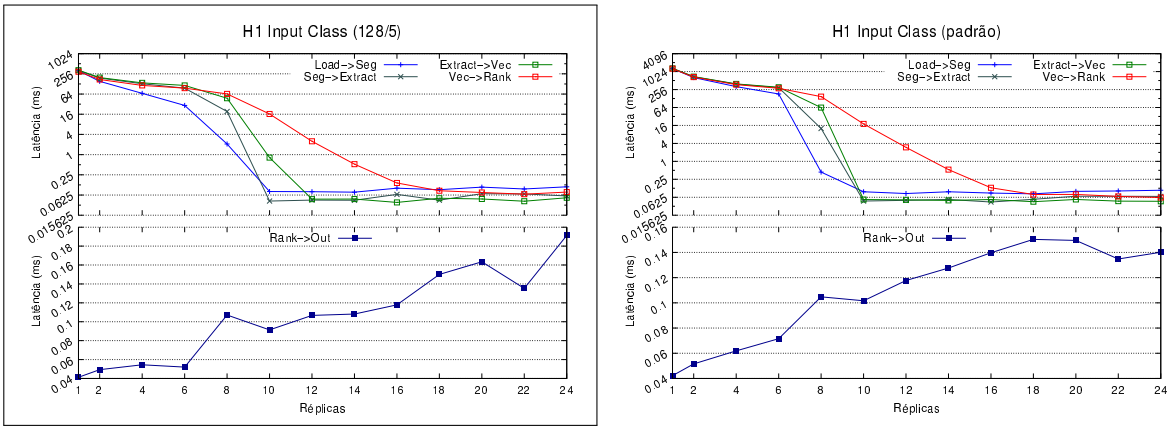

Figura 5.18 – Latência do Ferret com a sub-classe H1.

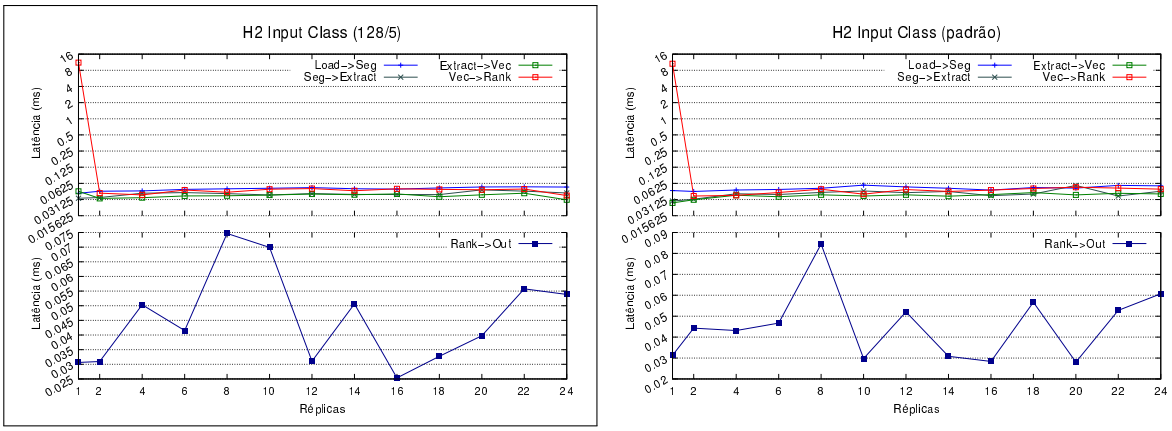

Figura 5.19 – Latência do Ferret com a sub-classe H2.

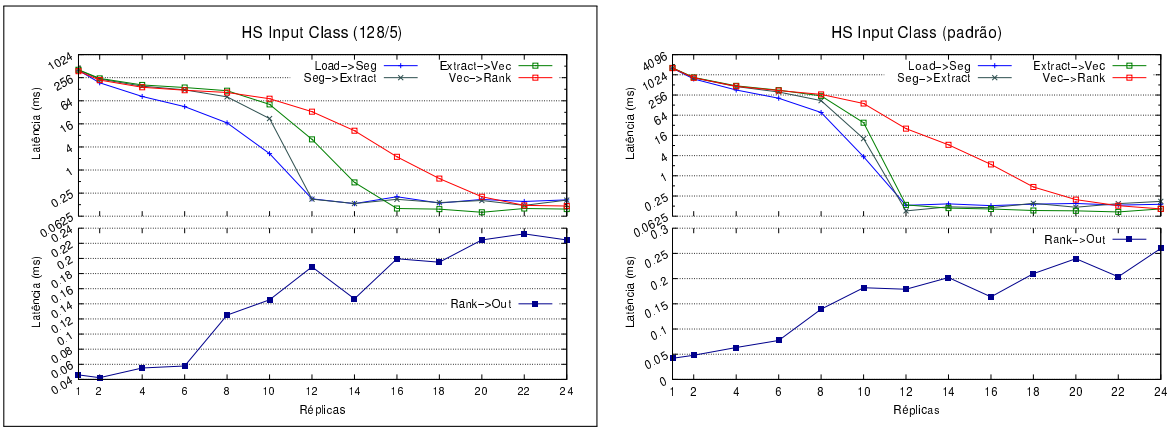

Figura 5.20 – Latência do Ferret com a sub-classe HS.

Aliado ao processo de segmentação, o tempo de E/S no primeiro estágio limita o ganho de desempenho.

O cenário parametrizado com o Ferret não apresentou diferenças significativas no comportamento ao ser comparado com o cenário padrão. As tentativas de parametrização do Ferret para explorar outros comportamentos ficaram limitadas. O maior impacto foi na métrica de latência, que foi influenciada pelo tamanho das filas. O algoritmo de segmentação gerou impacto no tempo de execução e na qualidade da busca de similaridade.

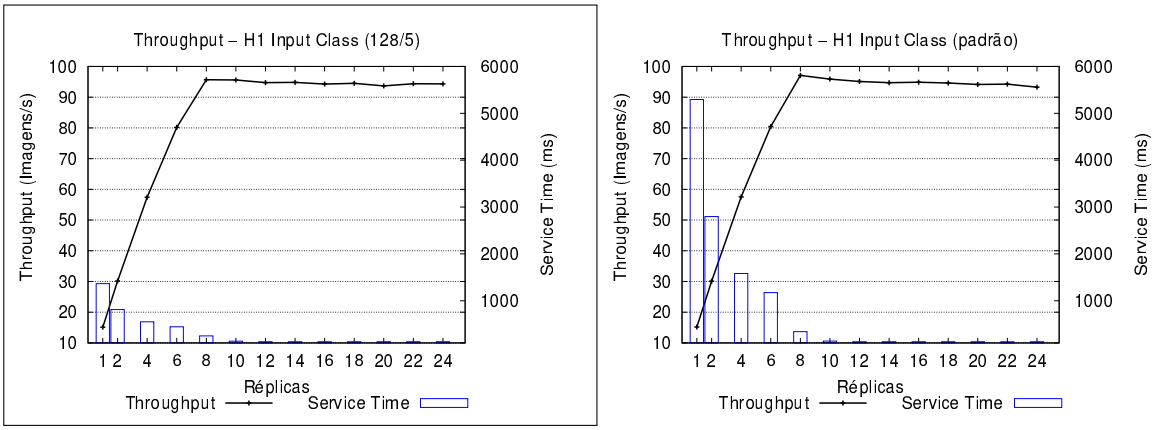

Figura 5.21 – Caracterização do desempenho com a sub-classe H1 no Ferret.

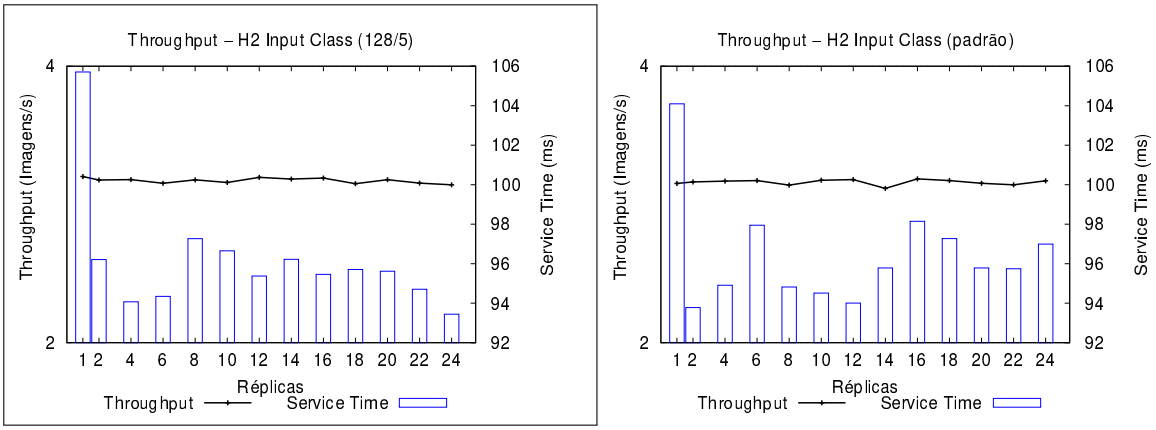

Figura 5.22 – Caracterização do desempenho com a sub-classe H2 no Ferret.

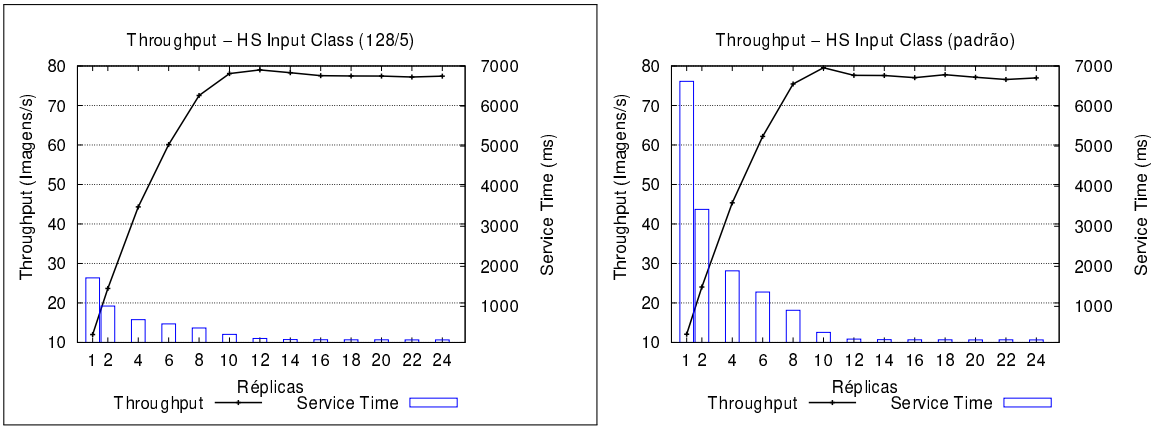

Figura 5.23 – Caracterização do desempenho com a sub-classe HS no Ferret.

5.3 Visão Geral dos Resultados

Nas seções anteriores foram mostrados os resultados com os *benchmarks* Dedup e Ferret. Na Seção 5.1, o cenário padrão de execução foi apresentado, o qual utiliza os valores padrões do ambiente do PARSEC para esses *benchmarks*. Os resultados do cenário de execução parametrizado é apresentado em seguida, na Seção 5.2. Nesse ce-

nário, os parâmetros implementados para os *benchmarks* foram explorados e os resultados mostrados.

O PARSEC é uma suíte composta por *benchmarks* que representam aplicações emergentes e de vários domínios [Bie11]. Esse é um dos atributos que consolidou a suíte na computação, pois com seus *benchmarks*, os esforços em pesquisas para demanda das aplicações podem ser direcionados [Bie11]. Sendo assim, para essa dissertação foram selecionados dois *benchmarks* com a finalidade de torná-los parametrizáveis, levando em conta algumas das características encontradas nas aplicações paralelas do processamento de *stream*.

Ao tornar os *benchmarks* parametrizáveis, foi possível analisar diversas questões relacionadas ao seus comportamentos de execução e também da suíte PARSEC. Algumas questões ligadas ao PARSEC foram apresentadas no Capítulo 4, quando o projeto de implementação desse trabalho foi discutido. Além disso, com os resultados das execuções, novas limitações foram encontradas nos *benchmarks*.

Para compreender tais limitações é importante considerar outro atributo da suíte PARSEC descrito por Bienia [Bie11], que afirma que a suíte foi criada para estudar os aspectos das arquiteturas paralelas de memória compartilhada com processadores *multi-cores*. Por esse motivo, diversas questões ligadas ao desempenho são negligenciadas nos *benchmarks*, pois o objetivo está em avaliar as características da arquitetura. Em consequência desse atributo da suíte, exemplos de trabalhos que investigam os comportamentos das aplicações em infraestruturas de nuvens computacionais utilizam os *benchmarks* do PARSEC [GVM+18]. Por outro lado, alguns trabalhos desconsideram esse atributo e investigam o desempenho, bem como a melhoria dos *benchmarks* para esse aspecto [NATC09, CCM+15, GDTF17].

A presente dissertação também investigou o desempenho do Dedup e do Ferret. Além disso, investigou uma maneira de gerar diferentes comportamentos nos *benchmarks* por meio de um conjunto de parâmetros. Foram criados, ainda, novos conjuntos de entradas (denominadas classes e sub-classes) para esses *benchmarks*, com o propósito de aproximar as execuções de um cenário real. No caso do Dedup, atualmente uma aplicação do domínio de compressão de dados processa diversos tipos de arquivos e tamanhos, diferentes dos existentes nas entradas oferecida pelo PARSEC (*Native* com 672 MB, por exemplo). Além disso, com as arquiteturas de processamento disponíveis atualmente, as entradas do PARSEC podem ser insignificantes.

Com o novo cenário de *benchmarks* parametrizáveis, várias questões sobre o desempenho foram analisadas detalhadamente nas Seções 5.1 e 5.2. Com essas discussões, as características das aplicações de *stream* puderam ser exploradas e analisadas utilizando estes *benchmarks*. Em seguida, uma visão geral sobre tais descobertas será apresentada.

Nas aplicações de *stream* que utilizam o *pipeline*, um fator determinante para o desempenho é o balanceamento de carga. Em uma aplicação desbalanceada ou com

estágios desbalanceados, o desempenho da aplicação é limitado ao estágio mais lento da aplicação [GTA06]. Além disso, com a aplicação desbalanceada existe sobrecarga gerada pela serialização e comunicação dos elementos [HSS+14]. No Dedup é observado que o desbalanceamento entre os estágios exige um alto custo de comunicação (Figura 5.9 e Figura 5.10), mas com a parametrização, esse problema foi amenizado, onde taxas maiores de *throughput* foram alcançadas, bem como a diminuição da latência em alguns casos.

No Ferret, esse problema com o balanceamento de carga também acontece. Inicialmente, o estágio *rank* realiza a maior parte da computação do Ferret, que acaba limitando também o desempenho do *benchmark*. Ainda que os estágios *segmentation* e *indexing* realizem uma quantidade significativa de trabalho, ele sempre será menor que o do estágio *rank*. Os dados utilizados no processamento também influenciam esse problema de desbalanceamento. Especificamente sobre os resultados dessa dissertação, temos que quando imagens com resoluções maiores são utilizadas, elas sobrecarregam o estágio *segmentation*, que faz a segmentação das imagens (Figura 5.6 e Figura 5.19). Diversos trabalhos que investigam o Ferret do PARSEC não utilizam entradas diferentes para ele e nem aumentam a quantidade de dados para o seu processamento. Portanto, o cenário de execução com o Ferret que foi implementado, mostrou que mesmo com um conjunto de entrada diferente do original, o desbalanceamento de carga persistiu.

A granularidade de trabalho é outro fator que impacta no desempenho de aplicações de *stream* que utilizam *pipeline*. Naturalmente, essas aplicações têm um comportamento de intensa comunicação, pois passar os elementos de um estágio a outro requer um ou mais canais de comunicação (filas) na aplicação. Em consequência disso, quando a granularidade de trabalho é pequena, um dos benefícios do *pipeline* é perdido porque a comunicação é mais frequente e torna-se um agravante para o desempenho da aplicação [GTA06]. Esse comportamento fica evidente com os resultados de caracterização do Dedup e do Ferret.

No Dedup, quando o tamanho do *chunk* (elemento) é maior no cenário parametrizado, o impacto se mostra positivo no tempo de execução e no *service time*. Além disso, a latência demonstrou ser menor, principalmente quando a comunicação acontecia entre as réplicas que executavam em processadores diferentes (considerando a arquitetura com multiprocessadores utilizada para testes), como mostram os resultados com mais de seis réplicas. Nessa situação, mesmo que a latência não tenha sido melhor para o Dedup em todos os casos, a aplicação não é dependente dela [ZBBL17].

No cenário com o Ferret, que nesse caso representa as aplicações de *stream* que utilizam o *pipeline*, a comunicação mostra um efeito negativo no desempenho quando a granularidade é pequena. Tal comportamento é evidente quando a semelhança existente nos gráficos de todos os cenários do Ferret é analisada. Graficamente, esses comportamentos são representados pelos formatos das linhas e das caixas. A granularidade (elemento) no Ferret é definida nos estágios *segmentation* e *extraction*. O primeiro nível de granularidade

é utilizado somente entre o segundo e o terceiro estágio. No segundo estágio, em razão da biblioteca *CASS-image*, o processo de extração reduz significativamente o elemento porque são criadas versões mais compactadas de cada região da imagem segmentada.

No entanto, mesmo o Ferret não melhorando o tempo de execução como o Dedup, a diminuição da latência foi o principal impacto que a parametrização gerou nele. Essa aplicação é dependente de latência [ZBBL17] e, em versões com processamento distribuído, isso poderia ser um fator determinante para o tempo de execução. Nessa dissertação, o processamento distribuído não foi analisado com o Dedup e o Ferret.

Pode ser amplo o cenário de parametrização de características do paralelismo de *stream*. O panorama com o Dedup e o Ferret apresentados nas seções anteriores, retrata algumas das principais características encontradas em aplicações reais. Outras características desse cenário que não foram incluídas no presente trabalho poderiam ser analisadas. Por exemplo: múltiplos *streams* de entrada e versões com processamento distribuído destes *benchmarks*. Além disso, outras aplicações também poderiam fazer parte da análise que foi realizada com o Dedup e o Ferret. No entanto, com essas e outras questões, a parametrização dependerá das características das aplicações.

Com os resultados discutidos até o momento, qualquer usuário tem um ponto de partida que indica quais principais características do paralelismo de *stream* podem influenciar o comportamento da aplicação. Os resultados aqui apresentados se aproximam do usuário preocupado com o comportamento da aplicação ou do seu desempenho. Se ele for um programador, as possibilidades de parametrização nos *benchmarks* o ajudariam a encontrar resultados próximos de um cenário real das aplicações. Assim, diversos comportamentos podem ser antecipados com a execução dos *benchmarks*. Além disso, os resultados das execuções podem ser utilizados para dimensionar corretamente uma arquitetura de processamento para estas aplicações, tanto na contratação de recursos em nuvem quanto na aquisição de equipamentos. Por outro lado, se o usuário não tem muito conhecimento sobre as aplicações, um *benchmark* parametrizável seria uma ferramenta de apoio para adquirir conhecimentos, uma vez que as características do paralelismo de *stream* são os parâmetros do *benchmark*.

6. CONCLUSÃO

Essa dissertação apresentou um cenário de avaliação com *benchmarks* parametrizáveis do domínio de processamento paralelo de *stream*. No Capítulo 1, introduziram-se as perspectivas do trabalho e sua motivação para compreender as características do domínio de *stream*. Cada vez mais, essas aplicações estão no cotidiano das pessoas, aplicadas em diversos cenários. Desse modo, no Capítulo 2 foram contextualizadas as aplicações de *stream*, apresentando suas principais diferenças e características. Aspectos relacionados ao seu desempenho também foram abordados, como a paralelização e as métricas de desempenho. No Capítulo 3, uma visão geral da literatura foi apresentada, onde se discutiu que nem todos os *benchmarks* abordam as características do processamento paralelo de *stream*, bem como a necessidade de se aprofundar o conhecimento sobre essa área. Sendo assim, no Capítulo 4, foi apresentado o projeto para alcançar os requisitos de parametrização com os *benchmarks* da suíte PARSEC. A maior parte dos parâmetros implementados representam as características do processamento paralelo de *stream*. Por meio disso, foi possível explorar diferentes comportamentos nos *benchmarks*, que foram apresentados e discutidos no Capítulo 5.

A presente pesquisa investigou as características das aplicações de *stream* e do seu processamento paralelo. Algumas das características foram transformadas em parâmetros que podem ser ajustados nos *benchmarks* Dedup e Ferret da suíte PARSEC. Além disto, novos conjuntos de entrada foram criados para executar com esses *benchmarks*, diferentes dos originais da suíte PARSEC. Com o novo cenário, foram avaliados os comportamentos baseados nas características existentes nas aplicações paralelas do domínio *stream*. A partir dos resultados alcançados, as conclusões foram organizadas da seguinte forma: na Seção 6.1, se discute sobre a característica janela deslizante; na Seção 6.2, é abordado sobre o elemento de *stream*; e sobre os canais de comunicação na Seção 6.3.

6.1 Janela Deslizante (*Sliding Window*)

Aplicações de *stream* implementadas com o *pipeline*, exigem intensivamente a comunicação entre *threads*, que ocorre por meio de filas, as quais são canais de comunicação usados para encaminhar os elementos do *stream* através dos estágios. A janela deslizante faz parte dessas comunicações, pois esta característica determina a quantidade de comunicação, manipulando quando necessário os elementos das filas. A manipulação das filas é a ação de retirar ou colocar elementos, realizado por cada estágio ou réplica. Para cada estágio não paralelo é necessário uma *thread*. Para cada estágio paralelo, podem existir réplicas, onde cada réplica é uma *thread*. Portanto, para manipular as filas entre os estágios, é

necessário um controle que serializa tal manipulação e afeta principalmente o desempenho da aplicação. Isso para os casos onde existe uma única fila entre os estágios.

Para as aplicações paralelas de *stream* que são executadas em um cenário com multiprocessadores e vários *cores*, podem existir dois agravantes gerados por essa serialização. O primeiro acontece devido a latência da comunicação entre os estágios que executam em processadores distintos, que pode aumentar o tempo de manipulação das filas se a latência for alta. O segundo ocorre quando existe concorrência de várias réplicas ou estágios em execução acessando uma única fila. Essa concorrência é o fator que causa a contenção de *locks* e pode ser potencializada se a latência de comunicação for alta. Portanto, todas as referidas questões estão relacionadas com a janela deslizante, pois esta característica determina a quantidade de comunicação na aplicação.

A janela deslizante foi uma característica analisada por meio da parametrização. No Dedup foi analisado o impacto gerado principalmente para o desempenho. Janelas deslizantes maiores diminuíram a necessidade de manipular as filas, fator que gerou um impacto positivo ao Dedup. Além disso, quando a comunicação acontecia entre os processadores, a latência foi menor em algumas situações. Isso foi discutido detalhadamente na Seção 5.2.1 e na Seção 5.3.

Devido às limitações de implementação do Ferret, não foi possível analisar a janela deslizante como forma de parâmetro. No entanto, é observado um impacto negativo gerado ao Ferret quando um valor baixo é utilizado na janela deslizante. Naturalmente, essa aplicação tem um comportamento intensivo de comunicação. Em consequência disso, o valor baixo da janela deslizante exige que a manipulação das filas seja mais frequente. Percebe-se, assim, um dos problemas que gera desbalanceamento de carga no Ferret. Detalhes sobre essas questões estão na Seção 5.1.2, Seção 5.2.2 e Seção 5.3.

6.2 Elemento do *Stream*

Nas aplicações de *stream*, o elemento pode ser representado por qualquer dado computacional que deve ser processado. No Dedup, o elemento é o *chunk*, que contém partes dos dados de entrada que foram fragmentados. No Ferret, o elemento contém as características de cada imagem. Essas características mudam de acordo com o processamento que cada estágio realiza. Nesse caso, pode-se dizer que o elemento representa uma imagem. O tipo do elemento e/ou seu tamanho, influencia a aplicação tanto no desempenho, como seu comportamento ao processá-lo. O tamanho do elemento consegue determinar a quantidade de comunicação entre os estágios. Já o tipo de elemento, além da comunicação, também determina fatores relacionados à maneira de processar os elementos.

No Dedup, com elementos de *stream* maiores, houve mais processamento ao invés de comunicação. Em uma situação assim, os elementos maiores compensam o fator de comunicação, pois mais tempo é dedicado ao processamento do elemento. Desse modo, como a comunicação sempre envolve operações sequenciais, o tempo maior de processamento e a pouca comunicação mantêm os benefícios do *pipeline* para uma aplicação como o Dedup. Essa discussão também se aplica para o tipo do elemento no Dedup, pois os dados multimídias geram elementos maiores após a fragmentação. Além disso, a principal influência do tipo do elemento foi no processo de deduplicação e compressão. Elementos que envolvem dados do tipo mídia (fotos, vídeos e áudios), mostraram um resultado pouco eficiente quando comparados com dados do tipo texto.

Devido às restrições do Ferret discutidas no Capítulo 4, não foi possível avaliar o tipo do elemento, somente seu tamanho foi avaliado. Nos resultados do Ferret, é possível observar o impacto do tamanho do elemento sobre a comunicação entre estágios. Com elementos menores, a comunicação é mais frequente porque a computação dos elementos é mais rápida em alguns estágios. Por outro lado, com elementos maiores, a computação é mais lenta e a comunicação é mais rápida. Porém, no caso do Ferret, a rápida comunicação com estes tamanhos de elementos é um falso desempenho causado pelo desbalanceamento de carga na aplicação. Isso foi discutido detalhadamente no Capítulo 5, na Seção 5.1.2 e na Seção 5.2.2.

6.3 Canais de Comunicação

As filas são o principal canal de comunicação entre os estágios de uma aplicação de *stream* em sistemas *multicores*. Além disso, alguns métodos de escalonamento se dão por essas filas, como ocorre em sistemas de processamento em tempo real de *stream* [LOSM18]. Em consequência disso, o desempenho das aplicações pode ser influenciado em razão desses canais de comunicação.

Existem operações sequenciais nestes canais de comunicação que as aplicações paralelas de *stream* realizam. Esta serialização ocorre porque é necessário um controle para tais operações. Isso é o mesmo que a manipulação realizada pela janela deslizante, explicada anteriormente. Quando possível, a serialização pode ser amenizada com o uso de *buffers*, que são como “filas” exclusivas dentro dos estágios.

No Ferret, a referida característica impactou na latência quando diferentes tamanhos de filas foram utilizados. Com filas menores, a latência é menor. Isso acontece porque o processamento dos elementos é rápido em alguns estágios. No entanto, utilizar somente as filas e um valor pequeno da janela deslizante, impactou principalmente no desempenho da aplicação e gerou um desbalanceamento de carga. No Dedup, somente os *buffers* foram

avaliados. Quando tamanhos maiores são utilizados, a latência é negativamente afetada em alguns casos comparados aos resultados do cenário padrão.

Mesmo que o tempo de execução do Ferret não tenha sido impactado pelos tamanho das filas, trata-se de uma aplicação dependente de latência [ZBBL17] e para essa situação, filas menores tiveram um resultado positivo que garantiu uma latência menor. Por outro lado, Dedup é dependente de *throughput*, e para os casos onde a latência foi maior, as taxas de *throughput* foram mais elevadas.

6.4 Visão Geral

Todas as questões discutidas nas seções anteriores, foram possíveis de serem analisadas com a parametrização das características do paralelismo de *stream* desenvolvidas no Dedup e no Ferret. Nas versões originais da suíte PARSEC isso não é possível, pois entender o comportamento da arquitetura se mostra como um dos principais objetivos da equipe do PARSEC. Por outro lado, essa dissertação analisa o comportamento das aplicações quando determinados parâmetros são ajustados. Tais parâmetros também são algumas das características mais comuns das aplicações do domínio de *stream*. Assim, ao ajustar as características, diversas questões relacionadas ao desempenho foram analisadas, como o tempo de execução, o *service time*, a latência e o *throughput*. Além disso, foi demonstrado que diferentes comportamentos são gerados, como a utilização do armazenamento, processamento eficiente da aplicação. Isso significa que um cenário mais realístico pode ser explorado com a parametrização das aplicações.

Além dos resultados que envolvem o domínio de aplicações de *stream*, essa dissertação complementa o cenário da suíte PARSEC. Esse trabalho pode ser utilizado como um documento adicional para compreender detalhadamente algumas questões sobre o cenário da suíte PARSEC. Foi visto que muitas questões não são evidenciadas nos trabalhos do PARSEC, por isso foi necessário buscar uma compreensão teórica e técnica além dos documentos disponíveis para que os resultados fossem alcançados.

6.5 Lista de Artigos

- **Em Direção à Comparação do Desempenho das Aplicações Paralelas nas Ferramentas OpenStack e OpenNebula**. 15th Escola Regional de Alto Desempenho do Estado do Rio Grande do Sul (ERAD/RS) [MGVS15].

- **Private IaaS Clouds: A Comparative Analysis of OpenNebula, CloudStack and OpenStack.** 24th Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP) [VGM+16].
- **Medindo o Desempenho de Implantações de OpenStack, CloudStack e OpenNebula em Aplicações Científicas.** 16th Escola Regional de Alto Desempenho do Estado do Rio Grande do Sul (ERAD/RS) [VMGS16].
- **Desempenho de OpenStack e OpenNebula em Estações de Trabalho: Uma Avaliação com Microbenchmarks e NPB.** Revista Eletrônica Argentina-Brasil de Tecnologias da Informação e da Comunicação (REABTIC) [MGSF16].
- **Em Direção à um Benchmark de Workload Sintético para Paralelismo de Stream em Arquiteturas Multicore.** 16th Escola Regional de Alto Desempenho do Estado do Rio Grande do Sul (ERAD/RS) [MGF16].
- **Caracterização do Desempenho de Aplicações Pipeline em Instâncias KVM e LXC de uma Nuvem CloudStack.** 17th Escola Regional de Alto Desempenho do Estado do Rio Grande do Sul (ERAD/RS) [BMGS17].
- **Improving the Network Performance of a Container-Based Cloud Environment for Hadoop Systems.** International Conference on High Performance Computing & Simulation (HPCS) [RGMF17].
- **Performance of Data Mining, Media, and Financial Applications under Private Cloud Conditions.** 23rd IEEE Symposium on Computers and Communications (ISCC) [GVM+18].
- **Uma Suíte de Benchmarks Parametrizáveis para o Domínio de Processamento de Stream em Sistemas Multi-Core.** 18th Escola Regional de Alto Desempenho do Estado do Rio Grande do Sul (ERAD/RS) [MF18].
- **Should PARSEC Benchmarks be More Parametric? A Case Study with Dedup.** 27th Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP) [MVGF19].

6.6 Trabalhos Futuros

Essa dissertação foi o ponto de partida para compreender com detalhes as características do paralelismo de *stream*. Diversos aspectos observados nesse cenário foram discutidos nos capítulos anteriores. No entanto, os resultados possibilitaram novas questões que ainda podem ser analisadas. Para que seja viável a continuidade desse trabalho, tais questões serão descritas em seguida.

Muitas características do paralelismo de *stream* não foram abordadas na presente pesquisa. Algumas aplicações utilizam múltiplos canais de comunicação, bem como de entrada ou de saída nos estágios. Essa é uma característica que deve ser analisada, pois existem operações sequenciais nos canais de comunicações, e nesse trabalho somente um canal de saída e entrada foi avaliado.

Os canais de comunicação avaliados sempre tiveram tamanhos únicos em cada execução. Com diferentes tamanhos, possivelmente o balanceamento de carga entre os estágios seria controlado. Por exemplo: filas entre estágios com processamento rápido podem ser menores, pois, assim, um elemento não aguardaria muito tempo até ser processado. Esses mesmos estágios poderiam conter *buffers* maiores, que ajudariam a evitar o acesso constante às filas. Não foram avaliadas as mencionadas questões, mas com a parametrização desenvolvida para o Dedup e o Ferret, elas ainda podem ser exploradas.

Outras abordagens relacionadas ao elemento do *stream* podem ser avaliadas para compreender essa característica nas aplicações. Por exemplo, melhorias podem ser feitas nas técnicas de fragmentação e segmentação do Dedup e do Ferret. Além disso, outras classes de entrada ainda podem ser avaliadas utilizando as versões parametrizáveis que foram desenvolvidas.

Novas implementações do Dedup e do Ferret podem ser feitas, melhorando as versões utilizadas nesse trabalho. Por exemplo: no Ferret podem ser implementados *buffers* em cada estágio, e o Dedup utilizar outros algoritmos de compressão. Além disso, outros *benchmarks* do PARSEC podem ser analisados em um cenário de parametrização. Essa pesquisa avaliou detalhadamente somente o Dedup e o Ferret, dos 13 *benchmarks* ainda disponíveis na suíte (incluíndo as versões distribuídas). Aplicações de outros domínios também podem ser analisadas, como aplicações de rede, processamento de IoT, e outras.

Técnicas que adaptam o grau de paralelismo também podem ser avaliadas nas versões parametrizáveis que foram desenvolvidas. Diversos trabalhos que estudam os *benchmarks* do PARSEC investigam o impacto do desempenho quando diferentes números de réplicas são utilizados em cada estágio [GSV+18, VGS+18]. No entanto, todos eles utilizam as versões originais, sem as parametrizações.

REFERÊNCIAS BIBLIOGRÁFICAS

[ADKT17] Aldinucci, M.; Danelutto, M.; Kilpatrick, P.; Torquati, M. “FastFlow: High-Level and Efficient Streaming on Multi-Core”. In: *Programming Multi-core and Many-core Computing Systems*, Wiley, 2017, cap. 13, pp. 261–280.

[AGT14] Andrade, H. C. M.; Gedik, B.; Turaga, D. S. “Fundamentals of Stream Processing: Application Design, Systems, and Analytics”. Cambridge University Press, 2014, 529p.

[AHL+15] Arnold, J. A.; Huppler, K.; Lange, K.-D.; Henning, J. L.; Cao, P.; et al.. “How to Build a Benchmark”. In: Proceedings of the 6th ACM/SPEC International Conference on Performance Engineering, 2015, pp. 333–336.

[Bie11] Bienia, C. “Benchmarking Modern Multiprocessors”, Tese de Doutorado, Princeton University, 2011, 153p.

[BKSL08] Bienia, C.; Kumar, S.; Singh, J. P.; Li, K. “The PARSEC Benchmark Suite: Characterization and Architectural Implications”. In: Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques, 2008, pp. 72–81.

[BL97] Burns, R. C.; Long, D. D. E. “Efficient Distributed Backup with Delta Compression”. In: Proceedings of the 5th Workshop on I/O in Parallel and Distributed Systems, 1997, pp. 27–36.

[BLM+09] Blumenthal, A.; Luedde, M.; Manzke, T.; Mielenhausen, B.; Swanepoel, C. E. “Measuring Software System Performance Using Benchmarks”, US Patent 7,546,598, 2009, 16p.

[BM09] Bhattacharjee, A.; Martonosi, M. “Characterizing the TLB Behavior of Emerging Parallel Workloads on Chip Multiprocessors”. In: Proceedings of the 18th International Conference on Parallel Architectures and Compilation Techniques, 2009, pp. 29–40.

[BMGS17] Baum, W.; Maron, C. A. F.; Griebler, D.; Schepke, C. “Caracterização do Desempenho de Aplicações Pipeline em Instâncias KVM e LXC de uma Nuvem CloudStack”. In: Proceedings of the 17th Escola Regional de Alto Desempenho do Estado do Rio Grande do Sul, 2017, pp. 267–270.

[CBM+09] Che, S.; Boyer, M.; Meng, J.; Tarjan, D.; Sheaffer, J. W.; Lee, S. H.; Skadron, K. “Rodinia: A Benchmark Suite for Heterogeneous Computing”. In: Proceedings of the IEEE International Symposium on Workload Characterization, 2009, pp. 44–54.

[CCM+15] Chasapis, D.; Casas, M.; Moretó, M.; Vidal, R.; Ayguadé, E.; Labarta, J.; Valero, M. “PARSECSs: Evaluating the Impact of Task Parallelism in the PARSEC Benchmark Suite”, *ACM Transactions on Architecture and Code Optimization*, vol. 12–4, Dez 2015, pp. 1–41.

[Cha09] Chang, B. “A Running Time Improvement for Two Thresholds Two Divisors Algorithm”, Dissertação de Mestrado, San Jose State University - SJSU Scholar Woks, 2009, 38p.

[CM08] Contreras, G.; Martonosi, M. “Characterizing and Improving the Performance of Intel Threading Building Blocks”. In: Proceedings of the IEEE International Symposium on Workload Characterization, 2008, pp. 57–66.

[DLP03] Dongarra, J. J.; Luszczek, P.; Petitet, A. “The LINPACK Benchmark: Past, Present and Future”, *Concurrency and Computation: Practice and Experience*, vol. 15–9, Ago 2003, pp. 803–820.

[DSDMT+17] De Sensi, D.; De Matteis, T.; Torquati, M.; Mencagli, G.; Danelutto, M. “Bringing Parallel Patterns Out of the Corner: The P3 ARSEC Benchmark Suite”, *ACM Transactions on Architecture and Code Optimization*, vol. 14, Out 2017, pp. 1–33.

[ET05] Eshghi, K.; Tang, H. K. “A Framework for Analyzing and Improving Content-Based Chunking Algorithms”, Relatório Técnico, Hewlett-Packard Labs Technical Report, 2005, 10p.

[GDTF17] Griebler, D.; Danelutto, M.; Torquati, M.; Fernandes, L. G. “SPar: A DSL for High-Level and Productive Stream Parallelism”, *Parallel Processing Letters*, vol. 27–01, Mar 2017, pp. 20.

[GF17] Griebler, D.; Fernandes, L. G. “Towards Distributed Parallel Programming Support for the SPar DSL”. In: Proceedings of the International Conference on Parallel Computing, 2017, pp. 10.

[GFDF18] Griebler, D.; Filho, R. B. H.; Danelutto, M.; Fernandes, L. G. “High-Level and Productive Stream Parallelism for Dedup, Ferret, and Bzip2”, *International Journal of Parallel Programming*, Fev 2018, pp. 1–19.

[GQS10] Geisler, S.; Quix, C.; Schiffer, S. “A Data Stream-based Evaluation Framework for Traffic Information Systems”. In: Proceedings of the ACM SIGSPATIAL International Workshop on GeoStreaming, 2010, pp. 11–18.

[Gra92] Gray, J. “Benchmark Handbook: For Database and Transaction Processing Systems”. Morgan Kaufmann Publishers Inc., 1992, 334p.

[Gri16] Griebler, D. “Domain-Specific Language & Support Tools for High-Level Stream Parallelism”, Tese de Doutorado, Faculdade de Informática - PPGCC - PUCRS, 2016, 243p.

[GSV+18] Griebler, D.; Sensi, D. D.; Vogel, A.; Danelutto, M.; Fernandes, L. G. “Service Level Objectives via C++11 Attributes”. In: Proceedings of the Euro-Par 2018: Parallel Processing Workshops, 2018, pp. 745–756.

[GTA06] Gordon, M. I.; Thies, W.; Amarasinghe, S. “Exploiting Coarse-grained Task, Data, and Pipeline Parallelism in Stream Programs”. In: Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems, 2006, pp. 151–162.

[GVM+18] Griebler, D.; Vogel, A.; Maron, C. A. F.; Maliszewski, A. M.; Schepke, C.; Fernandes, L. G. “Performance of Data Mining, Media, and Financial Applications under Private Cloud Conditions”. In: Proceedings of the 23th IEEE Symposium on Computers and Communications, 2018, pp. 450–456.

[HOC+10] Hong, S.; Oguntebi, T.; Casper, J.; Bronson, N.; Kozyrakis, C.; Olukotun, K. “Eigenbench: A Simple Exploration Tool for Orthogonal TM Characteristics”. In: Proceedings of the IEEE International Symposium on Workload Characterization, 2010, pp. 1–11.

[HSS+14] Hirzel, M.; Soulé, R.; Schneider, S.; Gedik, B.; Grimm, R. “A Catalog of Stream Processing Optimizations”, *ACM Computing Surveys*, vol. 46–4, Mar 2014, pp. 1–46.

[HVT98] Hunt, J. J.; Vo, K.-P.; Tichy, W. F. “Delta Algorithms: An Empirical Analysis”, *ACM Transactions on Software Engineering and Methodology*, vol. 7–2, Abr 1998, pp. 192–214.

[IPE14] Iosup, A.; Prodan, R.; Epema, D. “IaaS Cloud Benchmarking: Approaches, Challenges, and Experience”. Springer, 2014, cap. 1, pp. 83–104.

[Jai90] Jain, R. “The Art of Computer Systems Performance Analysis: Techniques for experimental Design, Measurement, Simulation, and Modeling”. John Wiley & Sons, 1990, 685p.

[LCL04] Lv, Q.; Charikar, M.; Li, K. “Image Similarity Search with Compact Data Structures”. In: Proceedings of the 13th ACM International Conference on Information and Knowledge Management, 2004, pp. 208–217.

[LJW+06] Lv, Q.; Josephson, W.; Wang, Z.; Charikar, M.; Li, K. “Ferret: A Toolkit for Content-based Similarity Search of Feature-rich Data”. In: Proceedings of

the 1st ACM SIGOPS/EuroSys European Conference on Computer Systems, 2006, pp. 317–330.

[LJW+07] Lv, Q.; Josephson, W.; Wang, Z.; Charikar, M.; Li, K. “Multi-probe LSH: Efficient Indexing for High-dimensional Similarity Search”. In: Proceedings of the 33rd International Conference on Very Large Data Bases, 2007, pp. 950–961.

[LOSM18] Langer, T.; Osinski, L.; Schmid, M.; Mottok, J. “Work-in-Progress: Real-Time Scheduling of Parallel Applications with Gang Scheduling”. In: Proceedings of the 31th International Conference on Architecture of Computing Systems, 2018, pp. 1–8.

[Lv06] Lv, Q. “Similarity Search for Large-scale Image Datasets”, Tese de Doutorado, Princeton University, 2006, 125p.

[LWXH14] Lu, R.; Wu, G.; Xie, B.; Hu, J. “Stream Bench: Towards Benchmarking Modern Distributed Stream Computing Frameworks”. In: Proceedings of the 7th IEEE/ACM International Conference on Utility and Cloud Computing, 2014, pp. 69–78.

[MCM01] Muthitacharoen, A.; Chen, B.; Mazières, D. “A Low-bandwidth Network File System”. In: Proceedings of the 18th ACM Symposium on Operating Systems Principles, 2001, pp. 174–187.

[MF18] Maron, C. A. F.; Fernandes, L. G. “Uma Suíte de Benchmarks Parametrizáveis para o Domínio de Processamento de Stream em Sistemas Multi-Core”. In: Proceedings of the 18th Escola Regional de Alto Desempenho, 2018, pp. 2.

[MGF16] Maron, C. A. F.; Griebler, D.; Fernandes, L. G. “Em Direção à um Benchmark de Workload Sintético para Paralelismo de Stream em Arquiteturas Multicore”. In: Proceedings of the 16th Escola Regional de Alto Desempenho, 2016, pp. 171–172.

[MGSF16] Maron, C. A. F.; Griebler, D.; Schepke, C.; Fernandes, L. G. “Desempenho de OpenStack e OpenNebula em Estações de Trabalho: Uma Avaliação com Microbenchmarks e NPB”, *Revista Eletrônica Argentina-Brasil de Tecnologias da Informação e da Comunicação*, vol. 1–6, Dez 2016, pp. 15.

[MGVS15] Maron, C. A. F.; Griebler, D.; Vogel, A.; Schepke, C. “Em Direção à Comparação do Desempenho das Aplicações Paralelas nas Ferramentas OpenStack e OpenNebula”. In: Proceedings of the 15th Escola Regional de Alto Desempenho do Estado do Rio Grande do Sul, 2015, pp. 205–208.

[MS17] Maleszewski, J.; Sosnowski, J. “Managing and Enhancing Performance Benchmarks”. In: *Advances in Dependability Engineering of Complex Systems*, Springer, 2017.

[MVGF19] Maron, C. A. F.; Vogel, A.; Griebler, D.; Fernandes, L. G. “Should PARSEC Benchmarks be More Parametric? A Case Study with Dedup”. In: Proceedings of the 27th Euromicro International Conference on Parallel, Distributed and Network-Based Processing, 2019, in press.

[NATC09] Navarro, A.; Asenjo, R.; Tabik, S.; Cascaval, C. “Analytical Modeling of Pipeline Parallelism”. In: Proceedings of the 18th International Conference on Parallel Architectures and Compilation Techniques, 2009, pp. 281–290.

[NBSV13] Nambiar, R.; Bhardwaj, R.; Sethi, A.; Vargheese, R. “A Look at Challenges and Opportunities of Big Data Analytics in Healthcare”. In: Proceedings of the IEEE International Conference on Big Data, 2013, pp. 17–22.

[NN04] Nock, R.; Nielsen, F. “Statistical Region Merging”, *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 26–11, Nov 2004, pp. 1452–1458.

[PGB11] Pusukuri, K. K.; Gupta, R.; Bhuyan, L. N. “Thread Reinforcer: Dynamically Determining Number of Threads via OS Level Monitoring”. In: Proceedings of the IEEE International Symposium on Workload Characterization, 2011, pp. 116–125.

[Rab81] Rabin, M. O. “Fingerprinting by Random Polynomials”, Relatório Técnico, Department of Computer Science - Harvard University, 1981, 14p.

[RDGF16] Redmon, J.; Divvala, S.; Girshick, R.; Farhadi, A. “You Only Look Once: Unified, Real-Time Object Detection”. In: Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition, 2016, pp. 779–788.

[RGMF17] Rista, C.; Griebler, D.; Maron, C. A. F.; Fernandes, L. G. “Improving the Network Performance of a Container-Based Cloud Environment for Hadoop Systems”. In: Proceedings of the International Conference on High Performance Computing & Simulation, 2017, pp. 619–626.

[RKO+11] Raman, A.; Kim, H.; Oh, T.; Lee, J. W.; August, D. I. “Parallelism Orchestration Using DoPE: The Degree of Parallelism Executive”. In: Proceedings of the 32th ACM SIGPLAN Conference on Programming Language Design and Implementation, 2011, pp. 26–37.

[SEH03] Sim, S. E.; Easterbrook, S.; Holt, R. C. “Using Benchmarking to Advance Research: A Challenge to Software Engineering”. In: Proceedings of the 25th International Conference on Software Engineering., 2003, pp. 74–83.

[SHWH12] Shilane, P.; Huang, M.; Wallace, G.; Hsu, W. “WAN-optimized Replication of Backup Datasets Using Stream-informed Delta Compression”, *ACM Transactions on Storage*, vol. 8–4, Nov 2012, pp. 1–26.

[SM02] Suel, T.; Memon, N. “Algorithms for Delta Compression and Remote File Synchronization”. Academic Press, 2002, cap. 13, pp. 1–24.

[Tar14] Tarvo, A. “Automatic Performance Modeling of Multithreaded Programs”. In: Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering, 2014, pp. 721–723.

[Thi10] Thies, William; Amarasinghe, S. “An Empirical Characterization of Stream Programs and Its Implications for Language and Compiler Design”. In: Proceedings of the 19th International Conference on Parallel Architectures and Compilation Techniques, 2010, pp. 365–376.

[Tic85] Tichy, W. F. “RCS — A System for Version Control”, *Software: Practice and Experience*, vol. 15–7, Jul 1985, pp. 637–654.

[TKA02] Thies, W.; Karczmarek, M.; Amarasinghe, S. P. “StreamIt: A Language for Streaming Applications”. In: Proceedings of the 11th International Conference on Compiler Construction, 2002, pp. 179–196.

[TR18] Tarvo, A.; Reiss, S. P. “Automatic Performance Prediction of Multithreaded Programs: A Simulation Approach”, *Automated Software Engineering*, vol. 25–1, Mar 2018, pp. 101–155.

[VGM+16] Vogel, A.; Griebler, D.; Maron, C. A. F.; Schepke, C.; Fernandes, L. G. “Private IaaS Clouds: A Comparative Analysis of OpenNebula, CloudStack and OpenStack”. In: Proceedings of the 24th Euromicro International Conference on Parallel, Distributed and Network-Based Processing, 2016, pp. 672–679.

[VGS+18] Vogel, A.; Griebler, D.; Sensi, D. D.; Danelutto, M.; Fernandes, L. G. “Autonomic and Latency-Aware Degree of Parallelism Management in SPar”. In: Proceedings of the Euro-Par 2018: Parallel Processing Workshops, 2018, pp. 28–39.

[VMGS16] Vogel, A.; Maron, C. A. F.; Griebler, D.; Schepke, C. “Medindo o Desempenho de Implantações de OpenStack, CloudStack e OpenNebula em Aplicações Científicas”. In: Proceedings of the 16th Escola Regional de Alto Desempenho do Estado do Rio Grande do Sul, 2016, pp. 279–282.

[WDQ+12] Wallace, G.; Douglis, F.; Qian, H.; Shilane, P.; Smaldone, S.; Chamness, M.; Hsu, W. “Characteristics of Backup Workloads in Production Systems”. In: Proceedings of the 10th USENIX Conference on File and Storage Technologies, 2012, pp. 4–4.

[ZBBL17] Zhan, X.; Bao, Y.; Bienia, C.; Li, K. “PARSEC3.0: A Multicore Benchmark Suite with Network Stacks and SPLASH-2X”, *ACM SIGARCH Computer Architecture News*, vol. 44–5, Fev 2017, pp. 1–16.

[ZL78] Ziv, J.; Lempel, A. “Compression of Individual Sequences via Variable-Rate Coding”, *IEEE Transactions on Information Theory*, vol. 24–5, Set 1978, pp. 530–536.

Pontifícia Universidade Católica do Rio Grande do Sul
Pró-Reitoria de Graduação
Av. Ipiranga, 6681 - Prédio 1 - 3º. andar
Porto Alegre - RS - Brasil
Fone: (51) 3320-3500 - Fax: (51) 3339-1564
E-mail: prograd@pucrs.br
Site: www.pucrs.br