UNIVERSIDADE FEDERAL DE SANTA CATARINA
CENTRO TECNOLÓGICO
CURSO DE GRADUAÇÃO EM CIÊNCIAS DA COMPUTAÇÃO

João Gabriel Trombeta

Avaliação do desempenho do particionamento de estado em Replicação Máquina de Estados Paralela

Florianópolis
2021

João Gabriel Trombeta

Avaliação do desempenho do particionamento de estado em Replicação Máquina de Estados Paralela

Trabalho de Conclusão de Curso submetida ao Curso de Graduação em Ciências da Computação da Universidade Federal de Santa Catarina para a obtenção do título de bacharel em Ciências da Computação.
Orientador: Prof. Odorico Machado Mendizabal, Dr.
Coorientador: Prof. Alvaro Junio Pereira Franco , Dr.

Florianópolis
2021

Ficha de identificação da obra elaborada pelo autor,
através do Programa de Geração Automática da Biblioteca Universitária da UFSC.

Trombeta, João Gabriel
Avaliação do desempenho do particionamento de estado em Replicação Máquina de Estados Paralela / João Gabriel Trombeta ; orientador, Odorico Machado Mendizabal, coorientador, Alvaro Junio Pereira Franco, 2021.
83 p.

Trabalho de Conclusão de Curso (graduação) - Universidade Federal de Santa Catarina, Centro Tecnológico, Graduação em Ciências da Computação, Florianópolis, 2021.

Inclui referências.

1. Ciências da Computação. 2. Replicação. 3. Sistemas distribuídos. 4. Tolerância a falhas. 5. Particionamento balanceado de grafos. I. Mendizabal, Odorico Machado. II. Franco, Alvaro Junio Pereira. III. Universidade Federal de Santa Catarina. Graduação em Ciências da Computação. IV. Título.

João Gabriel Trombeta

Avaliação do desempenho do particionamento de estado em Replicação Máquina de Estados Paralela

O presente trabalho em nível de bacharelado foi avaliado e aprovado por banca examinadora composta pelos seguintes membros:

Prof. Márcio Bastos Castro, Dr.

Prof. Pedro Belin Castellucci, Dr.

Certificamos que esta é a **versão original e final** do trabalho de conclusão que foi julgado adequado para obtenção do título de bacharel em Ciências da Computação.

Coordenação do Curso de Graduação

Prof. Odorico Machado Mendizabal, Dr.
Orientador

Prof. Alvaro Junio Pereira Franco , Dr.
Coorientador

Florianópolis, 2021.

AGRADECIMENTOS

Eu quero agradecer primeiramente à minha família. Meus pais Isamara e Oscar. Minha irmã Jéssica. Meus avós, Aldino e Elena, e Ireni e Chico. Meus tios e tias Alexandre, Jussara, e Angela. Aos meus primos, João (que me convenceu a cursar computação), Davi, Vitor, Lucas, Maria e Vinícius. Meu madrinho e madrinha, Ulisses e Ivonete. Eu cresci cercado de mais amor do que eu até hoje consigo entender. Não saber o futuro é uma das coisas que mais me causa ansiedade, mas ainda assim é extremamente reconfortante saber que independente do que aconteça, vocês estarão lá.

Pensei primeiro em fazer um parágrafo para os pais, e depois outro para o resto da família. Ao invés disso, decidi dar dois parágrafos inteiros para a família inteira. Cada um de vocês me ajudou e me ajuda a ser quem sou hoje, e por isso sou mutíssimo grato.

Também quero agradecer aos meus orientadores, Odorico e Alvaro. Foi/está sendo muito bom trabalhar com vocês, agradeço imensamente pela atensiosidade, disponibilidade e contribuições. E obrigado por se interessarem nesse projeto tanto quanto eu me interesso (mesmo com a loucura que é final de TCC hahaha). Aproveito para fazer uma menção ao professor Leandro, um dos melhores professores que tive a felicidade de ter na graduação. E também agraço à UFSC como um todo. Toda a infraestrutura e servidores. Cada vendedor na feirinha de quarta e cada tia do RU. Esses vários anos foram os que mais aprendi, tanto sobre computação quanto sobre mim mesmo, e tudo girou em volta desses 1 milhão de metros quadrados.

Depois, agradeço a meus amigos. No começo pensei em agradecer de maneira genérica. "Amigos". Mas eu realmente gostaria de tomar o tempo de mencionar cada pessoa, cada uma que foi importante para mim, de qualquer forma que tenha sido, em pelo menos uns 12 (doze) momentos. Eles não estão em nenhuma ordem específica. Agradeço à Gabriela, Luiz, Ana, José, Stephanie, Lara, Marcelo, Julia, Luiz, Camila, Ana, Cristóvan, Bianca, Graziela. Julia e Giovanna. Morgana, Jean, Victor, Thiago, Carine, Thaís. Allan. Gabriel, Lucca, Emanuel, Felipe, Rafael, Victoria, Priscilla. Luis, Otto, Evandro, Helena, Lucas, Thales, Mikael, Paloma, Teo, Arthur, Paola, Inã. Cauê, Arthur, Francisco, Gabriel, Lucas, Tarcísio, Lucas, Samuel, Gustavo, Diogo, Caique, Matheus, André, João, João, João, João, Caio, Marco, Pedro, Giovanni, Pedro, Bernardo, Nicole, Gabriel, Bernardo. Julia e Pedro. Lahis, Cesar, Gleydson, Wanda, Pedro, Camilla, Juliana. Alexandre, Camila e Vagner. Listar toda essa quantidade pode parecer demais, mas é bom que pareça demais. Quando eu era pequeno achava que não teria realmente muitos amigos, por ser a criança estranha. Obrigado por me fazerem perceber que eu estava errado, que eu tenho muitos amigos, por ser um quase adulto estranho.

Finalmente, termino com uma menção honrosa aos meus psicólogos, Vander-

lúcia e Pablo. Que todas as pessoas bebam água, tenham 8 horas de sono, e façam terapia, se possível.

“Como eles dizem na Rússia:

"я звезда - навсегда. спасибо большое".

”

(Zamolodchikova, 2018)

RESUMO

Replicação Máquina de Estados é uma técnica amplamente utilizada para prover tolerância a falhas e consistência forte em sistemas distribuídos. Nessa abordagem todas as requisições são executadas sequencialmente, na mesma ordem total, por todas as réplicas. Buscando melhorar a vazão do sistema, versões aprimoradas foram propostas, onde requisições independentes podem ser executadas em paralelo. Existe o desafio, porém, de como balancear a carga de trabalho entre *threads* trabalhadoras, ao mesmo tempo em que é necessário reduzir o número de sincronizações entre *threads*. Algoritmos de particionamento balanceado de grafos podem ser utilizados para atingir tais objetivos em sistemas paralelos e distribuídos. Esse trabalho apresenta um modelo de execução de Replicação Máquina de Estados Paralela que utiliza o particionamento balanceado de grafos, buscando balancear requisições e reduzir sincronizações entre *threads* em uma réplica arbitrária. Além disso, é apresentado um estudo que explora como a escolha do algoritmo de particionamento pode impactar o desempenho do sistema. Os resultados obtidos sugerem que ganhos de desempenho são possíveis, sendo altamente dependentes da característica da carga de trabalho, da frequência de reparticionamento, e algoritmo escolhido. **Palavras-chave**: Replicação. Sistemas distribuídos. Tolerância a falhas. Alta vazão. Desempenho. Particionamento balanceado de grafos.

ABSTRACT

State Machine Replication is a widely used technique to provide fault-tolerance and strong consistency in distributed systems. In this approach, all requests are executed sequentially, in the same total order, by all replicas. Aiming to improve the system's throughput, improved versions were proposed, where independent requests can be executed in parallel. However, there is the challenge of balancing the workload among worker threads, while it is also necessary to reduce the number of synchronizations among threads. Balanced graph partitioning algorithms can be utilized to achieve such objectives in parallel and distributed systems. This work presents a Parallel State Machine Replication execution model that uses balanced graph partitioning to balance workload and reduce synchronizations among threads in an arbitrary replica. Beyond that, a study on how the choice of partitioning algorithm can impact system's performance is presented. The results suggest that gains in performance are possible although highly dependent on the workload, repartition frequency, and chosen algorithm. **Keywords**: Replication. Distributed systems. Fault tolerance. High thorughput. Performance. Balanced graph partitioning.

LISTA DE FIGURAS

Figura 1 – Exemplo de particionamento de grafos multinível. 22
Figura 2 – Exemplo do escalonamento das requisições entre as filas de *threads*. 25
Figura 3 – Execução do sistema da Figura 2. 26
Figura 4 – Exemplo de grafo de trabalho. 27
Figura 5 – *Makespan* observado com YCSB-A para diferentes intervalos de reparticionamento, com grafo de 1.000 de vértices. 32
Figura 6 – *Makespan* observado com YCSB-A para diferentes intervalos de reparticionamento, com grafo de 1.000.000 de vértices. 33
Figura 7 – Vazão observada com YCSB-A com 8 partições e reparticionamento a cada 5,000,000 de requisições. Tamanho do grafo de 1.000 vértices. 34
Figura 8 – Vazão observada com YCSB-A com 8 partições e reparticionamento a cada 5,000,000 de requisições. Tamanho do grafo de 1.000.000 vértices. 35
Figura 9 – Vazão observada com YCSB-A para diferentes intervalos de reparticionamento. Tamanho do grafo de 1.000 de vértices. 36
Figura 10 – Vazão observada com YCSB-A para diferentes intervalos de reparticionamento. Tamanho do grafo de 1.000.00 de vértices. 37
Figura 11 – *Makespan* observado com YCSB-D para diferentes intervalos de reparticionamento. Grafo com 1.000 de vértices. 38
Figura 12 – *Makespan* observado com YCSB-D para diferentes intervalos de reparticionamento. Grafo com 1.000.000 de vértices. 39
Figura 13 – *Makespan* observado com YCSB-E com diferentes intervalos de reparticionamento. Tamanho do grafo de 1.000 de vértices. 40
Figura 14 – *Makespan* observado com YCSB-E com diferentes intervalos de reparticionamento. Tamanho do grafo de 1.000.000 de vértices. . . . 41
Figura 15 – Vazão observada por YCSB-E com 8 partições com repartições a cada 500,000 requisições. Tamanho do grafo de 1.000 de vértices. . 42
Figura 16 – Vazão observada por YCSB-E com 4 partições. Tamanho do grafo de 1,000,000 de vértices. 43
Figura 17 – Vazão observada com YCSB-E para diferentes intervalos de reparticionamento. Tamanho do grafo de 1.000 de vértices. 44
Figura 18 – Vazão observada com YCSB-E para diferentes intervalos de reparticionamento. Tamanho do grafo de 1.000.000 de vértices. 45
Figura 19 – Gráfico representando a vazão (eixo x) pela latência (eixo y), observados para a carga YCSB-A. 47
Figura 20 – Gráfico com métricas do sistema durante execução, observados para a carga YCSB-A com 8 partições e Δ_p = 50.000. 48

Figura 21 – Vazão observada com YCSB-A para diferentes intervalos de reparticionamento. 48
Figura 22 – Latência observada com YCSB-A para diferentes intervalos de reparticionamento. 49
Figura 23 – Gráfico de função de distribuição acumulada para latência de YCSB-A com 8 partições e Δ_p = 50.000. 49
Figura 24 – Gráfico representando a vazão (eixo x) pela latência (eixo y), observados para a carga YCSB-D. 50
Figura 25 – Gráfico com métricas do sistema durante execução, observados para a carga YCSB-D com 4 partições e Δ_p = 100.000. 51
Figura 26 – Vazão observada com YCSB-D para diferentes intervalos de reparticionamento. 52
Figura 27 – Latência observada com YCSB-D para diferentes intervalos de reparticionamento. 52
Figura 28 – Gráfico representando a vazão (eixo x) pela latência (eixo y), observados para a carga YCSB-E. 53
Figura 29 – Gráfico com métricas do sistema durante execução, observados para a carga YCSB-E com 8 partições e Δ_p = 50.000. 54
Figura 30 – Vazão observada com YCSB-E para diferentes intervalos de reparticionamento. 55
Figura 31 – Latência observada com YCSB-E para diferentes intervalos de reparticionamento. 55
Figura 32 – Gráfico de função de distribuição acumulada para latência de YCSB-E com 8 partições e Δ_p = 200.000. 56

LISTA DE ABREVIATURAS E SIGLAS

FE	FENNEL
Ka	KaHIP
ME	METIS
Re	ReFENNEL
RME	Replicação Máquina de Estados
RMEP	Replicação Máquina de Estados Paralela
YCSB	*Yahoo! Cloud Serving Benchmark*

LISTA DE SÍMBOLOS

G	Um grafo
V	Conjunto de vértices
E	Conjunto de arestas
v	Uma variável do sistema arbitrária/vértice no grafo
e	Uma aresta
P	Conjunto de partições
p	Uma partição
S	Estado da aplicação
r	Uma requisição
$V(r)$	Conjunto de variáveis acessadas por r
$P(v)$	Partição do grafo em que se encontra o vértice v
q	A fila de uma *thread*
T	Conjunto de *threads*
t	Uma *thread*
$c(v_i)$	Peso do vértice v
$w(e_{ij})$	Peso da aresta entre os vértices i e j
Δ_p	Intervalo de re-particionamento
γ	Variável configurável do algoritmo FENNEL
α	Variável configurável do algoritmo FENNEL

SUMÁRIO

1 INTRODUÇÃO 14
1.1 OBJETIVOS 15
1.1.1 Objetivo Geral 15
1.1.2 Objetivos Específicos 15
2 TRABALHOS RELACIONADOS 17
2.1 TÉCNICAS PARA REPLICAÇÃO MÁQUINA DE ESTADOS PARALELA 17
2.2 PARTICIONAMENTO EM SISTEMAS TRANSACIONAIS E RME . . 18
2.3 AVALIAÇÃO DE ALGORITMOS DE PARTICIONAMENTO BALANCEADO DE GRAFOS 19
3 PARTICIONAMENTO BALANCEADO DE GRAFOS 21
3.1 PARTICIONAMENTO MULTINÍVEL DE GRAFOS 21
3.2 PARTICIONAMENTO GULOSO 22
4 REPLICAÇÃO MÁQUINA DE ESTADOS PARALELA 24
4.1 ELABORAÇÃO DO MODELO 24
5 AVALIAÇÃO EXPERIMENTAL 28
5.1 CARGA DE TRABALHO 28
5.2 PROTÓTIPO DE RMEP 28
5.3 AVALIAÇÃO 30
5.3.1 Protótipo local 31
5.3.2 Protótipo distribuído 44
6 CONCLUSÃO 58
6.1 TRABALHOS FUTUROS 58
REFERÊNCIAS 60
APÊNDICE A – CÓDIGO DESENVOLVIDO 66
A.1 SIMULADOR 66
A.2 PROTÓTIPOS 67
APÊNDICE B – ARTIGOS 69
B.1 XI COMPUTER ON THE BEACH 69
B.2 XX ESCOLA REGIONAL DE ALTO DESEMPENHO DA REGIÃO SUL 77

1 INTRODUÇÃO

Replicação Máquina de Estados (RME) (SCHNEIDER, 1990; LAMPORT, 1978) é uma técnica bem estabelecida que provê disponibilidade para serviços distribuídos, enquanto assegura consistência forte, *i.e.* linearizabilidade (HERLIHY; WING, 1990), entre as réplicas. Nesse modelo, todos as requisições de clientes são ordenadas e entregues a cada réplica no sistema. Réplicas iniciam no mesmo estado e deterministicamente executam uma mesma sequência uniforme e ordenada de requisições, assegurando, portanto, consistência.

Em RME existe a limitação de que requisições são executadas sequencialmente, subutilizando recursos computacionais que podem estar disponíveis, e impondo um gargalo no processamento das requisições. Um corpo crescente de estudos em Replicação Máquina de Estados Paralela (RMEP) (KOTLA; DAHLIN, 2004; KAPRITSOS *et al.*, 2012; MARANDI *et al.*, 2014; ALCHIERI *et al.*, 2017; MENDIZABAL *et al.*, 2017; ALCHIERI *et al.*, 2018; LI *et al.*, 2018) buscam extrair paralelismo de RME, permitindo que requisições independentes sejam executadas em paralelo sem comprometer a consistência. Duas requisições são consideradas independentes se atualizam posições de memória disjuntas do estado da aplicação, ou caso sejam compostas exclusivamente por operações de leitura.

Apesar do potencial paralelismo observado, o desempenho da execução de qualquer sequência de requisições é altamente dependente de como as requisições são distribuídas entre as *threads* trabalhadoras, uma vez que um mau escalonamento resulta em sincronizações caras (*e.g.* barreiras) (ALCHIERI *et al.*, 2017). Idealmente, o processo de escalonamento deve balancear a carga de trabalho igualmente entre as *threads*, enquanto resulta numa quantidade mínima de sincronizações.

O problema da otimização do escalonamento assemelha-se ao problema do particionamento balanceado de grafos (GAREY *et al.*, 1974). No particionamento balanceado de grafos, o objetivo é dividir um grafo em subconjuntos disjuntos, e a soma dos pesos dos vértices em cada partição deve ser balanceada. Quando vértices são distribuídos em diferentes subconjuntos, podem existir arestas cujos vértices pertencerão a subconjuntos diferentes, chamadas arestas de corte. Em adição a balancear os pesos entre os subconjuntos, o objetivo é também minimizar a soma dos pesos das arestas de corte. Aplicando o balanceamento particionado de grafos no contexto do problema de escalonamento, vértices são variáveis da aplicação, valorados de acordo com o número de acessos, enquanto arestas e seus pesos representam requisições envolvendo múltiplas variáveis. Essas requisições multi-variáveis implicam em sincronização caso sejam representadas por arestas de corte.

O particionamento balanceado de grafos é um problema NP-Completo (GAREY *et al.*, 1974), o que significa que não possui solução em tempo polinomial, a menos

que *P* = *NP*. Por representar um problema comum em muitas áreas, existem diversos estudos e algoritmos propostos que implementam diferentes heurísticas para procurar soluções, *e.g.* (KARYPIS; KUMAR, V., 1998a, 1998b; SANDERS; SCHULZ, 2013; TSOURAKAKIS *et al.*, 2014; NISHIMURA; UGANDER, 2013). Abordagens no campo de processamento distribuído e paralelo adotam o uso do particionamento de grafos escolhendo um algoritmo arbitrário para a fase de particionamento e avaliando seu impacto na vazão do sistema. Apesar disso, existe uma falta de conhecimento sobre como diferentes implementações e heurísticas podem afetar o grau de paralelismo provido pelo particionamento resultante, e as implicações no desempenho que esses algoritmos podem ter na prática.

Esse trabalho foca em aplicações usando o particionamento de grafos para balancear a execução entre *threads* trabalhadoras em RMEP. Réplicas consideram o padrão de acesso de requisições passadas para reparticionar seu estado e prever um particionamento melhor para requisições futuras. O objetivo é explorar como diferentes algoritmos podem impactar o paralelismo da execução e desempenho geral do sistema. Quanto aos algoritmos de particionamento balanceado de grafos, foram escolhidos o METIS, (KARYPIS; KUMAR, V., 1998a), KaHIP (SANDERS; SCHULZ, 2013), FENNEL (TSOURAKAKIS *et al.*, 2014) e ReFENNEL (NISHIMURA; UGANDER, 2013). Para avaliar os algoritmos, foram desenvolvidos protótipos de RMEP, analisados sob diferentes cargas de trabalho. As cargas de trabalho geradas reproduzem perfis de acesso definidos pelo consolidado *benchamrk Yahoo! Cloud Serving Benchmark* (YCSB) (COOPER *et al.*, 2010), buscando submeter o sistema a padrões de requisições realistas.

1.1 OBJETIVOS

1.1.1 Objetivo Geral

Realizar comparações entre diferentes algoritmos de particionamento balanceado de grafos, analisando o desempenho de cada um quando utilizado para balancear requisições e explorar paralelismo em réplicas em um sistema de RMEP.

1.1.2 Objetivos Específicos

- Definir um modelo de execução de RMEP que utilize do particionamento de estados para obter paralelismo.
- Realizar uma investigação sobre os algoritmos de particionamento balanceado de grafos existentes e elencar quais serão utilizados no estudo.
- Comparar o uso de algoritmos de particionamento balanceado de grafos com estratégias que utilizam particionamento estático.

- Medir o impacto no tempo de execução e na vazão do sistema causado pelas estratégias de reparticionamento através de protótipos de aplicação local e distribuído. Com o protótipo local pretende-se observar a qualidade dos particionamentos sem a influência dos protocolos de comunicação e consenso distribuído. Já o protótipo distribuído permite uma observação de um sistema replicado típico.

2 TRABALHOS RELACIONADOS

Esse capítulo apresenta trabalhos relacionados divididos em três grupos. Primeiro, diferentes estratégias de implementação de RMEP são discutidas. Depois, são apresentadas pesquisas com enfoque em particionamento em sistemas transacionais e RME. Finalmente, são discutidas pesquisas que avaliam e comparam algoritmos de particionamento balanceado de grafos.

2.1 TÉCNICAS PARA REPLICAÇÃO MÁQUINA DE ESTADOS PARALELA

Diferentes estratégias podem ser adotadas para lidar com dependência de requisições em RMEP. Em (KOTLA; DAHLIN, 2004), um grafo direcionado acíclico mantém registro das dependências das requisições, onde vértices representam requisições, e arestas direcionadas a dependência envolvendo as requisições. *Threads* trabalhadoras recebem requisições independentes (*i.e.*, vértices sem arestas de entrada) de um escalonador para serem executadas concorrentemente.

Em (MENDIZABAL *et al.*, 2017), requisições são agrupadas em lotes, e mapas de *bits* são usados para manter registro da dependência entre lotes. Lotes de requisições são designados a *threads* trabalhadoras, em oposição a requisições individuais. Filas individuais, uma por *thread*, são usadas pelo escalonador para despachar lotes independentes. Lotes dependentes são enfileirados em mais de uma fila de *thread* e as *threads* sincronizam a execução desses lotes.

Em (MARANDI *et al.*, 2014; LI *et al.*, 2018), os autores propõem uma variação de RMEP onde a execução e entrega das requisições ocorrem em paralelo. Tradicionalmente em RME, as réplicas participam em rodadas de consenso, uma rodada por vez, para ordenar as requisições. Ao invés de usar uma única sequência de rodadas de consenso, essa abordagem usa múltiplas sequências de consenso. Cada sequência de consenso entrega requisições para a partição a qual a informação acessada está endereçada, dada por uma função de mapeamento. O mapeamento pode ser estático (MARANDI *et al.*, 2014) ou dinâmico (LI *et al.*, 2018). Requisições envolvendo mais de uma partição implicam em caras sincronizações entre *threads* trabalhadoras de suas respectivas partições.

Em (ALCHIERI *et al.*, 2017, 2018), os autores propõem o uso de classes de conflito. Diferentes tipos de relações de dependência (*e.g.* conflito-para-todos, conflito-para-alguns, conflito-para-nenhum) são definidas antecipadamente, assim como a indicação de quais *threads* devem sincronizar na presença de tais classes. Então, requisições são etiquetadas com a classe de conflito apropriada e o escalonador despacha requisições para as *threads*, observando as regras associadas a cada classe de conflito.

Apesar das particularidades de cada implementação de RMEP, no Capítulo 4

é apresentado um modelo de execução de RMEP genérico capaz de descrever requisições concorrentes e sincronizantes. Portanto, os resultados apresentados nesse trabalho podem ser generalizados de forma a serem aplicáveis à maioria das implementações de RMEP.

2.2 PARTICIONAMENTO EM SISTEMAS TRANSACIONAIS E RME

Sistemas distribuídos e paralelos comumente adotam estratégias de particionamento para a fragmentação do estado da aplicação, visando melhorar escalabilidade e desempenho. Nessa seção foram selecionados trabalhos relacionados que usam o particionamento balanceado de grafos para dividir o estado da aplicação no contexto de sistemas transacionais e RME.

Ao aplicar particionamento horizontal em tabelas, isso é, dividir registros de uma única tabela de um banco de dados entre partições dispostas em múltiplos servidores, existe a necessidade de distribuir tuplas de maneira que a carga de trabalho seja balanceada entre as partições. Além disso, é preciso que as comunicações requeridas para a sincronização de servidores seja mantida ao mínimo. Schism (CURINO *et al.*, 2010) cria um grafo de trabalho, onde dados são representados como vértices e arestas valoradas representam acesso conjunto. O grafo de trabalho é particionado e o resultado é um candidato à particionamento. O candidato é comparado com outros esquemas de particionamento que visam reduzir a transferência de dados. Eles usam a biblioteca METIS para realizar o particionamento. As abordagens apresentadas em (KUMAR, K. A. *et al.*, 2013) e (QUAMAR *et al.*, 2013) representam dados de uma maneira similar, mas usando um hipergrafo. O hipergrafo é particionado usando hMETIS como um passo de pré processamento. No segundo passo, o particionamento resultante é submetido a mais heurísticas que levam em consideração replicação e custo de transferência de dados. No contexto de sistemas distribuídos, a necessidade de minimizar o custo de comunicação é mais proeminente, uma vez que o custo de sincronização entre redes de computadores é tipicamente alto.

O particionamento de grafos em RME e RMEP tem recebido crescente interesse por pesquisadores. Dynastar (LE *et al.*, 2019) fragmenta o estado do serviço em múltiplas partições, e cada partição é replicada como serviços de RME independentes. Ele conta com um servidor oráculo que mantém registro do padrão da carga de trabalho na forma de um grafo. O grafo é usado para reparticionar o estado da aplicação utilizando a biblioteca METIS. Em (LI *et al.*, 2018), um único servidor executa múltiplas instâncias independentes do protocolo de consenso. O particionamento de grafos é usado para particionar o estado da aplicação e atribuir dados para um dos protocolos de consenso, respeitando o mapeamento da partição. Sua abordagem usa o KaHIP como algoritmo de particionamento.

Outras pesquisas também adotam o particionamento de estado em RME e

RMEP, por exemplo (MARANDI *et al.*, 2014) que implementa múltiplos protocolos de consenso, similar a (LI *et al.*, 2018), e (COELHO; PEDONE, 2018) que implementa RME escalável para sistemas replicados geograficamente distribuídos. Porém, eles contam com mapeamento de partição estático, sendo suscetíveis à degradação de desempenho em cargas de trabalho desbalanceadas.

O particionamento de estados nas abordagens supracitadas é descrito em termos de arquitetura de *software* e potencial benefícios de desempenho, mas eles carecem uma discussão detalhada sobre os prós e contras dos algoritmos de particionamento, e o impacto da escolha do algoritmo no desempenho, causado pelo reparticionamento em tempo de execução. Neste trabalho são avaliadas as diferenças de desempenho causadas pelo algoritmo escolhido, exercitando a execução de diferentes algoritmos sob cargas de trabalhos realísticas de *benchmarks* consolidados (*i.e.* YCSB).

2.3 AVALIAÇÃO DE ALGORITMOS DE PARTICIONAMENTO BALANCEADO DE GRAFOS

O particionamento balanceado de grafos tem um papel importante em muitas áreas de estudos, e várias heurísticas e algoritmos de aproximação tentam resolvê-lo. Uma vez que existe um grande número de algoritmos propostos, alguns pesquisadores compararam a qualidade da solução oferecida por diferentes algoritmos.

Adone *et al.*(ADONI *et al.*, 2019) descrevem heurísticas comuns e comparam a qualidade do particionamento usando como métricas o desbalanceamento entre partições, tempo de execução, arestas de corte entre as partições e complexidade. Em (ABBAS *et al.*, 2018), algoritmos de fluxo contínuo (*streaming*) são comparados e classificados de acordo com aspectos como complexidade espacial e temporal, estratégias e restrições. Um estudo similar é apresentado em (PACACI; ÖZSU, 2019) usando METIS como referência. Análises efetuadas por ambos (ABBAS *et al.*, 2018) e (PACACI; ÖZSU, 2019) usam buscas em banco de dados de grafos, como *PageRank*, para comparar o desempenho do particionamento. Sakouhi *et al.* (SAKOUHI *et al.*, 2018) classificam algoritmos considerando estabilidade, escalabilidade e convergência, mas não são dados detalhes sobre a configuração do experimento. Finalmente, Guo *et al.* (GUO *et al.*, 2017) comparam algoritmos de *streaming* em buscas considerando grafos com centenas de milhões de vértices.

Apesar de investigarem a qualidade do particionamento e desempenho em buscas em bancos de dados de grafos, nenhuma das pesquisas mencionadas considera o particionamento de grafos sendo utilizado para otimizar buscas futuras com base em padrões passados. Neste trabalho, os algoritmos de particionamento balanceado são usados para balancear carga de trabalho e reduzir sincronização decorrente de requisições que ainda serão recebidas, baseado no padrão de cargas de trabalho pas-

sadas. O objetivo é avaliar algoritmos e entender o quão bem eles conseguem explorar paralelismo em réplicas de RMEP, e como afetam o tempo de execução e vazão.

3 PARTICIONAMENTO BALANCEADO DE GRAFOS

Considere um grafo $G = (V, E)$, onde V é o conjunto de vértice e E é o conjunto de arestas. Cada vértice v_i e aresta e_{ij} (conectando os vértices v_i e v_j) tem um peso inteiro positivo. Uma partição de G é uma coleção de conjuntos disjuntos de vértices $P = \{p_1, \ldots, p_k\}$, onde $V = \bigcup_{p \in P} p$. O peso de um conjunto p_t é definido como a soma dos pesos de todos os vértices em p_t. Note que a partição P induz um conjunto de arestas que atravessam diferentes conjuntos da coleção, o conjunto de arestas de corte. O peso do conjunto de arestas de corte é a soma dos pesos de todas as suas arestas. No problema do particionamento balanceado de grafos, o objetivo é particionar G em k partições $P = \{p_1, \ldots, p_k\}$, de forma que o peso de cada conjunto p_t em P seja próximo à soma dos pesos de todos os vértices do grafo dividida por k, acrescido de uma tolerância; e o peso do conjunto de arestas de corte seja mínimo. A tolerância é usada para garantir que cada vértice pertença a algum conjunto em P. Por simplicidade, cada conjunto disjunto de vértices p_t é também chamado de partição. O contexto sempre será suficiente para demarcar esses diferentes objetos.

O particionamento balanceado de grafos é conhecidamente um problema *NP*-Completo, mesmo quando o número de conjuntos de partições é fixo, maior ou igual a 2, e os pesos de todos os vértices e arestas são iguais a 1 (GAREY *et al.*, 1974).Por ser um problema *NP*-Completo, existem muitos algoritmos que implementam heurísticas para obter resultados em tempo polinomial. Esse trabalho foca em duas estratégias de particionamento comumente usadas, o *particionamento multinível* (HENDRICKSON; LELAND, 1995) e *particionamento guloso*. Os algoritmos multinível foram escolhidos porque alcançam excelentes resultados na prática. O particionamento guloso tem sido utilizado em cenários de fluxo contínuo (*streaming*) com resultados razoáveis.

3.1 PARTICIONAMENTO MULTINÍVEL DE GRAFOS

A abordagem multinível consiste em contrair o grafo em grafos menores. O particionamento é feito no grafo menor, e então ele é descontraído para sua forma original. A ideia é que uma boa solução no grafo menor ainda pode ser uma boa solução para o particionamento do grafo original.

Existem três fases distintas em um algoritmo multinível. A primeira é chamada de fase de *engrossamento*, é quando o grafo original é contraído em grafos menores. Múltiplos vértices são agrupados em um só, o peso do novo vértice é a soma dos pesos dos vértices que o compõem. O mesmo acontece para arestas.

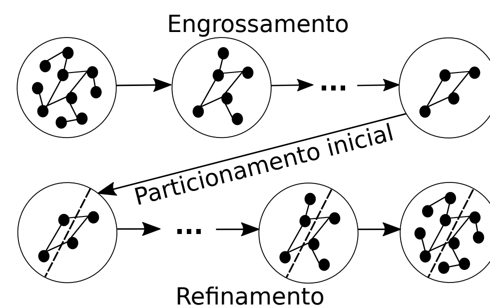
Na fase de *particionamento inicial* o grafo pequeno é particionado. Normalmente utilizam-se algoritmos que seriam muito caros para serem executados no grafo original, portanto a necessidade de contraí-lo.

Finalmente, na fase de *refinamento*, o grafo é descontraído, um passo de cada

vez. Depois de cada passo são aplicadas heurísticas, na tentativa de melhorar a qualidade do particionamento. O resultado dessa fase é o particionamento do grafo original.

A Fig. 1 mostra um exemplo de um particionamento, onde o grafo é submetido à mudanças pelo engrossamento, então o particionamento inicial é aplicado, e finalmente o refinamento, depois do qual é obtido o particionamento do grafo de entrada.

Figura 1 – Exemplo de particionamento de grafos multinível.



Fonte: Elaborado pelo autor (2019).

Da abordagem multinível, dois algoritmos foram avaliados, METIS (KARYPIS; KUMAR, V., 1998a) e KaHIP (SANDERS; SCHULZ, 2013). Eles diferem pelos algoritmos e heurísticas utilizados nas fases de engrossamento e refinamento.

O algoritmo METIS usa um algoritmo de emparelhamento de arestas pesadas ordenadas na fase de engrossamento, ele prioriza agrupar vizinhos com arestas pesadas, conectando-os. No particionamento inicial, ele usa o método de bisseções recursivas, onde o grafo é biparticionado recursivamente até que se atinja o número desejado de partições. Durante a fase de refinamento, o particionamento inicial é refinado usando algoritmos gulosos.

O *KaHIP* é utilizado em sua variante *KaFFPaFast*, que tem como objetivo um particionamento rápido. Ele utiliza dois algoritmos de emparelhamento na fase de engrossamento, um buscando velocidade e outro qualidade. Bisseções recursivas são também utilizadas na fase de particionamento. Durante a fase de refinamento, utiliza uma variante do algoritmo de FM (FIDUCCIA; MATTHEYSES, 1982) para melhorar a qualidade da solução.

3.2 PARTICIONAMENTO GULOSO

O *Particionamento Guloso* é uma maneira de particionar um grafo rapidamente. Os primeiros algoritmos gulosos trabalhavam em cenários *offline*, *i.e.*, o grafo é conhecido ao início do processo de particionamento. Atualmente algoritmos consideram um cenário *online*, onde o grafo é desconhecido, e seus elementos (vértices, arestas e seus pesos) são descobertos em tempo real, como em um fluxo continuo de dados. Alguns algoritmos gulosos para cenários *offline* aparecem em (JAIN *et al.*, 1998;

BATTITI; BERTOSSI, 1999), e para cenários *online* em (TSOURAKAKIS *et al.*, 2014; NISHIMURA; UGANDER, 2013). Esse trabalho explora a variante *online*, chamados de algoritmos de fluxo contínuo (*streaming*).

A abordagem de *particionamento de grafos em fluxo contínuo(streaming)* nasceu na área de processamento de buscas em grafos. Quando um grafo muito grande é lido do dispositivo de armazenamento e vai ser processado de maneira distribuída, ele precisa ser gradualmente carregado por uma máquina coordenadora, que atribui vértices e arestas a múltiplos trabalhadores. Já que os vértices e arestas são carregados e colocados em alguma máquina distribuída, a coordenadora pode aplicar heurísticas enquanto realiza a distribuição, tentando minimizar o peso do conjunto de arestas de corte enquanto faz isso (desde que as heurísticas não afetem o desempenho). Para alguns algoritmos, como os descritos nesse trabalho, uma função de custo determina o custo de designar vértices e arestas para cada máquina, baseado nos que já lhe foram designados. Os vértices e arestas são enviados à máquina que atinge um melhor resultado local da função de custo.

Da abordagem *streaming*, dois algoritmos gulosos foram avaliados, FENNEL (TSOURAKAKIS *et al.*, 2014) e ReFENNEL (NISHIMURA; UGANDER, 2013).

No FENNEL, vértices são carregados um por vez, e é calculado o resultado de uma função de custo desse vértice para cada partição. A função de custo depende do peso do vértice carregado, o peso atual da partição sendo avaliada, e a soma dos pesos das arestas entre o vértice carregado e seus vizinhos que já estão nesta partição. O algoritmo atribuiu o vértice para a partição que minimiza a função de custo.

O ReFENNEL é um algoritmo que estende a utilização do FENNEL. Inicialmente, no FENNEL, todas as partições estão vazias, o que faz com que os primeiros vértices tenham pouca informação sobre as partições ao calcular a função de custo. Durante a execução do ReFENNEL, o cálculo do primeiro particionamento é feito através da execução tradicional do FENNEL. Para os próximos reparticionamentos do mesmo grafo, o ReFENNEL usa a partição atual como ponto inicial, para que a atribuição dos primeiros vértices seja uma decisão mais informada, melhorando a qualidade da solução.

4 REPLICAÇÃO MÁQUINA DE ESTADOS PARALELA

Em Replicação Máquina de Estados tradicional (SCHNEIDER, 1990; LAMPORT, 1978), um conjunto ilimitado de clientes envia requisições determinísticas para um conjunto limitado de réplicas. As requisições são ordenadas via, por exemplo, um algoritmo de consenso ou difusão atômica, que define uma ordem uniforme de entrega para todas as réplicas. Uma vez que as réplicas iniciam no mesmo estado e executam as mesmas requisições determinísticas, na mesma ordem sequencial, a consistência entre réplicas é garantida.

Replicação Máquina de Estados Paralela (*e.g.* (KOTLA; DAHLIN, 2004; KAPRITSOS *et al.*, 2012; MARANDI *et al.*, 2014; ALCHIERI *et al.*, 2017; MENDIZABAL *et al.*, 2017; ALCHIERI *et al.*, 2018; LI *et al.*, 2018)) vem da observação de que apenas requisições dependentes devem ser executadas sequencialmente para garantir consistência, enquanto requisições independentes podem ser executadas em paralelo. Duas requisições são consideradas independentes se acessam posições de memória diferentes ou ambas são operações de leitura. Caso contrário, elas são requisições dependentes, e devem ser ordenadas. Este trabalho segue um modelo de execução similar ao proposto em (MENDIZABAL *et al.*, 2017; MARANDI *et al.*, 2014; LI *et al.*, 2016, 2018).

4.1 ELABORAÇÃO DO MODELO

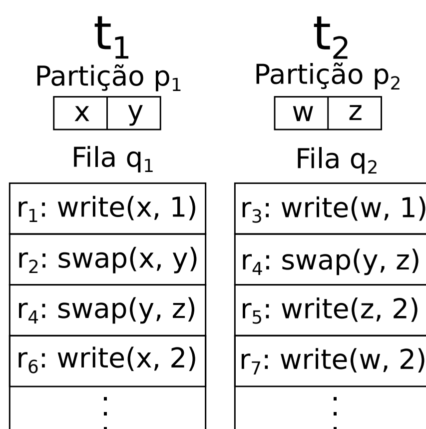
No modelo proposto, o estado da aplicação S é particionado em k conjuntos disjuntos, aqui chamados de partições. O conjunto de partições é denotado por $P = \{p_1, p_2, ..., p_k\}$, de forma que $\bigcup_{i=1}^{k} p_i = S$. O sistema executa k *threads* trabalhadoras, onde a *thread* t_i é responsável pelas requisições envolvendo a partição p_i. Uma requisição arbitrária r executa sobre um conjunto de variáveis da aplicação. Chamamos de $V(r)$ o conjunto de variáveis acessadas por r, e $P(v)$ a partição para qual uma variável v em $V(r)$ foi designada. Toda *thread* t_i tem sua própria fila q_i, com requisições que acessam sua partição designada p_i. Quando uma requisição r é entregue, ela é inserida nas filas de todas as *threads* cujas variáveis são acessadas por r. Formalmente, r é colocada na fila q_i, para todo i, tal que $p_i \in \bigcup_{v \in V(r)} P(v)$.

Como exemplo, o modelo de execução proposto implementa um sistema chave-valor. Um armazenamento chave-valor é visto como pares $\langle chave, valor \rangle$, onde o acesso atômico é majoritariamente implementado na granularidade de chaves individuais, o que permite concorrência e potencial execução paralela. Algumas operações realizam acessos multi-chaves (AMER-YAHIA *et al.*, 2008; DAS *et al.*, 2010), por exemplo escanear um intervalo de chaves ou trocar o valor de duas chaves. Nesse caso, a atomicidade é atingida através de mecanismos de sincronização que garantem exclusão mútua no acesso às chaves. Logo, esse sistema é trivialmente abrangido pelo

modelo proposto, uma vez que requisições podem acessar uma única ou múltiplas chaves, que são as variáveis da aplicação.

Considerando o sistema com o conjunto de chaves $S = \{x, y, w, z\}$ e um conjunto de *threads* $T = \{t_1, t_2\}$, a Figura 2 representa uma possibilidade de particionamento, com as *threads* t_1 e t_2 sendo responsáveis por requisições nas partições $p_1 = \{x, y\}$ e $p_2 = \{w, z\}$, respectivamente. Adicionalmente, a figura mostra as filas das *threads* após receber as requisições r_1 até r_7. Nesse exemplo encontram-se dois tipos de requisições, *write*(k, v), que escreve v na chave k, e *swap*(k_i, k_j) que troca os valores das chaves k_i e k_j. Por se tratar de uma réplica em RMEP, requisições são recebidas do protocolo de consenso e um escalonador as despacha para as filas apropriadas respeitando a ordem de chegada.

Figura 2 – Exemplo do escalonamento das requisições entre as filas de *threads*.



Fonte: Elaborado pelo autor (2019).

Requisições são divididas em duas categorias, requisições de *variável única* e requisições *multi-variáveis*. Requisições de variável única acessam somente uma variável do estado da aplicação, em um passo atômico, enquanto requisições multi-variáveis acessam duas ou mais. Requisições multi-variáveis também podem ser dividas em dois tipos, requisições de *partição única* ou *multi-partições*.

Mais precisamente, requisições de *variável única* são requisições que acessam uma única variável, então, dada uma requisição r, $|V(r)| = 1$. Na Figura 2 as requisições r_1, r_3, r_5, r_6 e r_7 são de variável única e podem ser executadas sem a necessidade de sincronização entre *threads*.

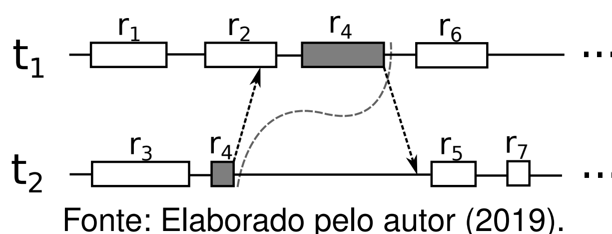
Requisições *multi-variáveis de partição única* acessam múltiplas chaves que pertencem à mesma partição e, portanto, são executadas pela mesma *thread* t. No exemplo, a requisição r_2 é multi-variável de partição única.

Requisições *multi-variáveis multi-partições* acessam duas ou mais variáveis, sendo pelo menos duas pertencentes a partições diferentes. A requisição r_4 é uma multi-variável multi-partições, ela acessa y que pertence à partição da *thread* t_1, e z,

que pertence à partição da *thread* t_2. Esse tipo de requisição requer sincronização entre as *threads* para garantir consistência.

A Figura 3 ilustra um traço de execução para as *threads* t_1 e t_2, exemplificadas pela Figura 2. Todas as *threads* executam em paralelo, r_1, r_3, r_5, r_6 e r_7 são requisições de variável única e podem ser executadas assim que estiverem na frente das suas filas. A requisição r_2 é multi-variável de partição única, ela envolve x e y, e ambas as variáveis são acessadas por t_1, então r_2 pode ser executada sem sincronização. A requisição r_4 é multi-variável multi-partições, portanto está presente em t_1 e t_2. Quando t_2 retira r_4 de sua fila, ela espera por t_1 em uma barreira, representada pela curva pontilhada. Quando t_1 retira r_4, todas as *threads* envolvidas estão prontas e r_4 é executada por uma única *thread* arbitrária, por exemplo, pela *thread* com o menor identificador (nesse exemplo, t_1). Depois disso, ambas as *threads* podem continuar suas execuções. O tempo entre t_2 atingir r_4 e sua execução, t_2 permanece completamente ociosa.

Figura 3 – Execução do sistema da Figura 2.



Fonte: Elaborado pelo autor (2019).

Existem dois objetivos ao particionar o estado da aplicação e distribuir essas partições entre *threads*. O primeiro é balancear a carga de trabalho, de forma que nenhuma *thread* fique subutilizada. Distribuir o mesmo número de variáveis para todas as *threads* não garante o balanceamento, uma vez que podem existir variáveis que são acessadas frequentemente, enquanto outras mal são acessadas. O segundo objetivo é reduzir a quantidade de requisições multi-variáveis multi-partições, uma vez que elas requerem sincronização e, portanto, fazem com que *threads* permaneçam ociosas. Para resolver esse problema de otimização, *particionamento balanceado de grafos* pode ser usado.

Para particionar o estado da aplicação é necessário manter registro do padrão de acesso às variáveis. Isso pode ser feito na forma de um grafo, para que seja usado como entrada para um algoritmo de particionamento balanceado de grafos. O *grafo de trabalho* é um grafo valorado G que registra o padrão de acessos. Uma variável i é representada por um vértice v_i cujo peso $c(v_i)$ indica quantas vezes a variável foi acessada por requisições multi-variáveis ou de variável única. Quando duas variáveis, i e j, são acessadas por uma requisição multi-variável, esse acesso é representado por uma aresta e_{ij} ligando os vértices v_i e v_j. O peso dessa aresta é $w(e_{ij})$ e representa o número de vezes em que i e j foram acessadas juntas por requisições multi-variáveis.

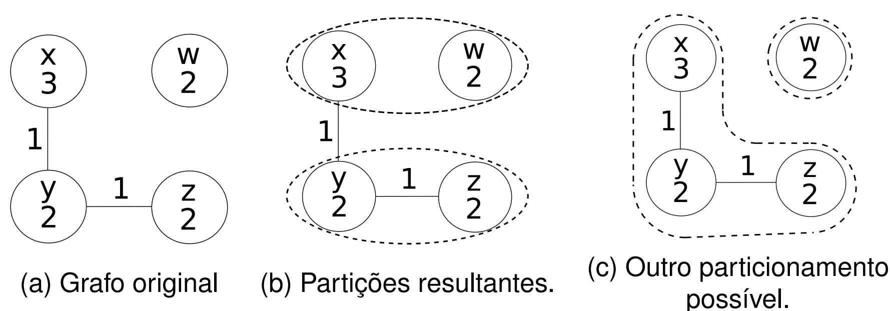
O grafo de trabalho é dinâmico e é atualizado durante a execução do serviço. Depois de certo evento que serve como gatilho, o grafo de trabalho é particionado usando um algoritmo de particionamento balanceado de grafos, e as partições resultantes são associadas às *threads*. A ideia por trás disso é que o conhecimento do padrão de acesso de requisições passadas pode ajudar a obter um esquema de particionamento melhor para requisições futuras.

Um bom esquema de particionamento deve balancear a carga de trabalho igualmente entre as *threads*, assim como manter a execução de requisições *multi-variáveis* dentro de uma única partição, sempre que possível. O particionamento balanceado de grafos é composto por dois subproblemas, o balanceamento das partições e a minimização da soma dos pesos das arestas cujos vértices estão em partições diferentes. Pela restrição de balanceamento, a soma dos pesos dos vértices em cada partição deve ser similar, o que significa que o número de requisições em cada partição deve ser similar, balanceando a carga de trabalho. Pela minimização da soma dos pesos das arestas cujos vértices estão em partições diferentes, o particionamento vai priorizar manter na mesma partição grupos de variáveis que são frequentemente acessados juntas por requisições multi-variáveis, evitando sincronizações.

A Figura 4(a) mostra um exemplo de um grafo de trabalho baseado na execução ilustrada pela Figura 2. As requisições r_2 e r_4 são responsáveis pela criação das arestas de peso 1 e_{xy} e e_{yz}, respectivamente. Depois do particionamento do grafo, duas partições novas, $p_1 = \{v_x, v_w\}$ e $p_2 = \{v_y, v_z\}$, são criadas, como ilustrado pela Figura 4(b). Nessa partição resultante, apenas uma aresta de peso 1 teve seus vértices separados, e_{xy}. A aresta e_{yz} é mantida na mesma partição.

O problema de particionamento permite múltiplas soluções, como a mostra a Figura 4(c), que mantém todas as requisições multi-variáveis em uma única partição. Esse particionamento evitaria qualquer sincronização, mas resultaria em um desbalanceamento da carga de trabalho.

Figura 4 – Exemplo de grafo de trabalho.



(a) Grafo original (b) Partições resultantes. (c) Outro particionamento possível.

Fonte: Elaborado pelo autor(2019).

5 AVALIAÇÃO EXPERIMENTAL

Esse capítulo apresenta a comparação de desempenho dos quatro algoritmos de particionamento balanceado de grafos apresentados no Capítulo 3, junto com um particionamento por *Round-Robin* para ser usado como referência.

5.1 CARGA DE TRABALHO

A carga de trabalho foi gerada utilizando o *Yahoo Cloud Serving Benchmark* (YCSB) (COOPER *et al.*, 2010). É um *benchmark* amplamente utilizado que fornece cargas de trabalho para sistemas chave-valor e sistemas em nuvem. Os padrões de *benchmark* escolhidos foram YCSB-A, YCSB-E e YCSB-D.

YCSB-A é composto por requisições de variável única, sendo 50% escritas e 50% leituras. Essa carga é utilizada para analisar o balanceamento de trabalho fornecido por cada algoritmo, uma vez que não existem requisições multi-variáveis e portanto não existem sincronizações a serem evitadas.

Em YCSB-E 95% são requisições de escaneamento de intervalos, *i.e.*, requisições multi-variáveis, enquanto 5% das requisições são inserções. A grande quantidade de requisições multi-variáveis desse padrão de acesso é adequada para analisar o quão bem cada algoritmo consegue evitar a execução de requisições multi-variáveis multi-partições, que implicam em sincronização. A carga de trabalho foi configurada para escanear até 8 chaves por requisição.

Por fim, em YCSB-D 5% das requisições são inserções e 95% são leituras, essas leituras são realizadas principalmente em chaves recém inseridas. Essa é uma carga desafiadora para o modelo de execução, uma vez que o padrão de acesso das requisições às chaves muda constantemente, e o modelo prevê novos particionamentos baseado no histórico de requisições.

5.2 PROTÓTIPO DE RMEP

Para comparar os algoritmos, foi implementado um serviço replicado em RMEP que executa um sistema chave-valor, onde o número de partições é o número de *threads*, e o algoritmo utilizado para o particionamento é configurável. Dois protótipos foram desenvolvidos, sendo um deles de execução local e outro de execução distribuída.

Ambos protótipos estão armazenados de maneira pública na plataforma Github, e podem ser acessados via link: https://github.com/gabrieltron/kvpaxos. O protótipo distribuído encontra-se na *branch master*, enquanto o protótipo local está na *branch simplified-replica*.

Nos protótipos, pares $\langle$*chave*, *valor*$\rangle$ podem ser inseridos, atualizados ou lidos. Valores associados às chaves são recuperados através de duas operações, *leitura* e

escaneamento de intervalo. A operação leitura recupera o valor associado a uma chave, enquanto o escaneamento de intervalo retorna uma lista de valores associados a um intervalo de chaves consecutivas. Operações de atualização comprimem os valores e os inserem à tabela, enquanto operações de leitura descomprimem antes de retornar o valor. Chaves são inteiros de 32 bits, enquanto valores são *strings* de 4 kbytes.

O armazenamento chave-valor possui uma implementação *multi-threaded* composta por um *escalonador*, um conjunto de *threads trabalhadoras* e uma *thread responsável por atualizações no grafo de trabalho*.

- O *escalonador* lê uma lista de requisições e as despacha para as *threads* apropriadas de acordo com o mapeamento das partições. Em *períodos de reparticionamento* específicos, dados a cada Δ_p requisições despachadas, o escalonador sincroniza com a *thread* do grafo de trabalho, executa o particionamento e atualiza o mapeamento das partições. Antes de continuar o despacho de requisições com um novo mapeamento, o escalonador envia uma requisição de sincronização para todas as *threads* trabalhadoras, para que todas as requisições de um particionamento antigo sejam executadas antes de iniciar as referentes ao novo.

- Toda *thread trabalhadora* é associada a uma partição de acordo com o mapeamento de partições. *Threads* trabalhadoras repetidamente verificam suas filas, e extraem e executam a primeira requisição nela. Requisições de partição única são executadas imediatamente, e requisições multi-partições são coordenadas com as demais *threads* envolvidas (como visto na Figura 3).

- A *thread do grafo de trabalho* é responsável por registrar o padrão de acesso das requisições. Essa *thread* recebe todas as requisições despachadas para as *threads* trabalhadoras, na mesma ordem em que chegaram, e atualiza o grafo de trabalho.

Para todas as cargas de trabalho utilizadas no protótipo foram feitas execuções individuais de cada algoritmo, considerando 2 quantidades de chaves iniciais distintas. Inicialmente, o número de chaves inicial no sistema é de 1.000 chaves, o que significa que o grafo de trabalho tem inicialmente 1.000 de vértices. Após, foi testado 1.000.000 de chaves iniciais no sistema, o que significa que o grafo de trabalho tem inicialmente 1.000.000 de vértices. Foram feitos testes com 2, 4 e 8 partições e, para cada caso, foram analisados os seguintes valores para Δ_p : 100.000, 300.000, 500.000 e 1.000.000. Para as cargas de trabalho YCSB-A e YCSB-D foram ainda executados cenários com Δ_p assumindo os valores 5.000.000 e 10.000.000. Nas cargas de trabalho com inserções de chaves, estas representam 5% das requisições, resultando em um grafo de aproximadamente 2.501.000 vértices para a carga YCSB-D e 251.000 vértices para a carga YCSB-E ao final da execução.

No começo da execução, chaves são mapeadas para partições seguindo um esquema *Round-Robin*, *i.e.*, as chaves são designadas à partições em ordem circular. Para comparar o impacto no desempenho causado por diferentes estratégias de particionamento de grafos, foram feitas execuções com cada um dos algoritmos a serem avaliados, FENNEL (FE), ReFENNEL (Re), KaHIP (Ka) e METIS (ME). KaHIP foi configurado para KaFFPaFast, que busca um particionamento rápido. Os algoritmos FENNEL e ReFENNEL possuem variáveis configuráveis chamadas γ e α, ambas foram definidas seguindo a sugestão dos autores no artigo original (TSOURAKAKIS *et al.*, 2014), com $\gamma = 1.5$ e $\alpha = \sqrt{k} \times W(E)/W(V)^{1.5}$, onde $W(E)$ é a soma dos pesos de todas as arestas, $W(V)$ é a soma dos pesos de todos os vértices, e k é o número de partições. Para todos os algoritmos foi permitido um desbalanceamento de até 20% entre partições.

É importante mencionar que o algoritmo FENNEL foi originalmente desenvolvido para trabalhar em cenários de processamento de fluxo contínuo (*streaming*), onde o grafo é dado gradualmente. O ReFENNEL, além da característica de *streaming*, supõe que os grafos de entrada consecutivos, cada um com particionamentos possivelmente distintos, são similares. O desempenho do FENNEL e ReFENNEL é, nesse trabalho, comparada em um cenário *offline*, sem restrições de memória. Os experimentos originais realizados pelos autores desses algoritmos assumem grafos cujo peso dos vértices e arestas são iguais a 1. Nesse trabalho, o peso de vértices e arestas são inteiros positivos e ambos algoritmos são executados em intervalos fixos. Além disso, o ReFENNEL parte de um particionamento já calculado previamente via FENNEL como passo inicial. Esse comportamento foi adaptado de duas maneiras distintas, denominadas Re1 e Re2. Em Re1, o primeiro particionamento utiliza FENNEL, e os reparticionamentos subsequentes executam ReFENNEL, tendo a partição atual como ponto inicial. Em Re2, todos os reparticionamentos executam inicialmente FENNEL sob o grafo, e a saída do FENNEL é então usada como ponto de partida para o ReFENNEL.

Para o caso da abordagem *Round-Robin*, como esse esquema de particionamento não considera o padrão de acesso aos dados, a *thread do grafo* não é executada e o reparticionamento é desabilitado. Consequentemente, não existem atualizações no grafo, interrupções no escalonamento, nem execução de reparticionamento a cada intervalo Δ_p.

5.3 AVALIAÇÃO

O particionamento resultante fornecido por diferentes algoritmos de particionamento podem afetar o grau de paralelismo entre *threads* trabalhadoras e, consequentemente, o desempenho na execução de requisições. Portanto, são analisados os algoritmos de particionamento pela execução do protótipo sob diferentes cenários. Todos os experimentos foram executados utilizando a plataforma Emulab (WHITE *et al.*,

2002), que disponibiliza infraestrutura sob demanda para a execução de experimentos. As execuções foram realizadas em máquinas denominadas d430, que são computadores com processador E5-2630v3 DE a 2.4 GHz, e com 64GB de memória RAM, configurados para utilizar o sistema operacional Ubuntu Linux 18.04. Para o protótipo distribuído, as máquinas são conectadas em LAN com velocidade de transmissão de dados de 10Gb/s. Os experimentos envolvem sistemas *multi-threads*. As *threads* são gerenciadas através da implementação presente na biblioteca padrão, que nesse cenário é uma abstração que utiliza *threads* POSIX.

5.3.1 Protótipo local

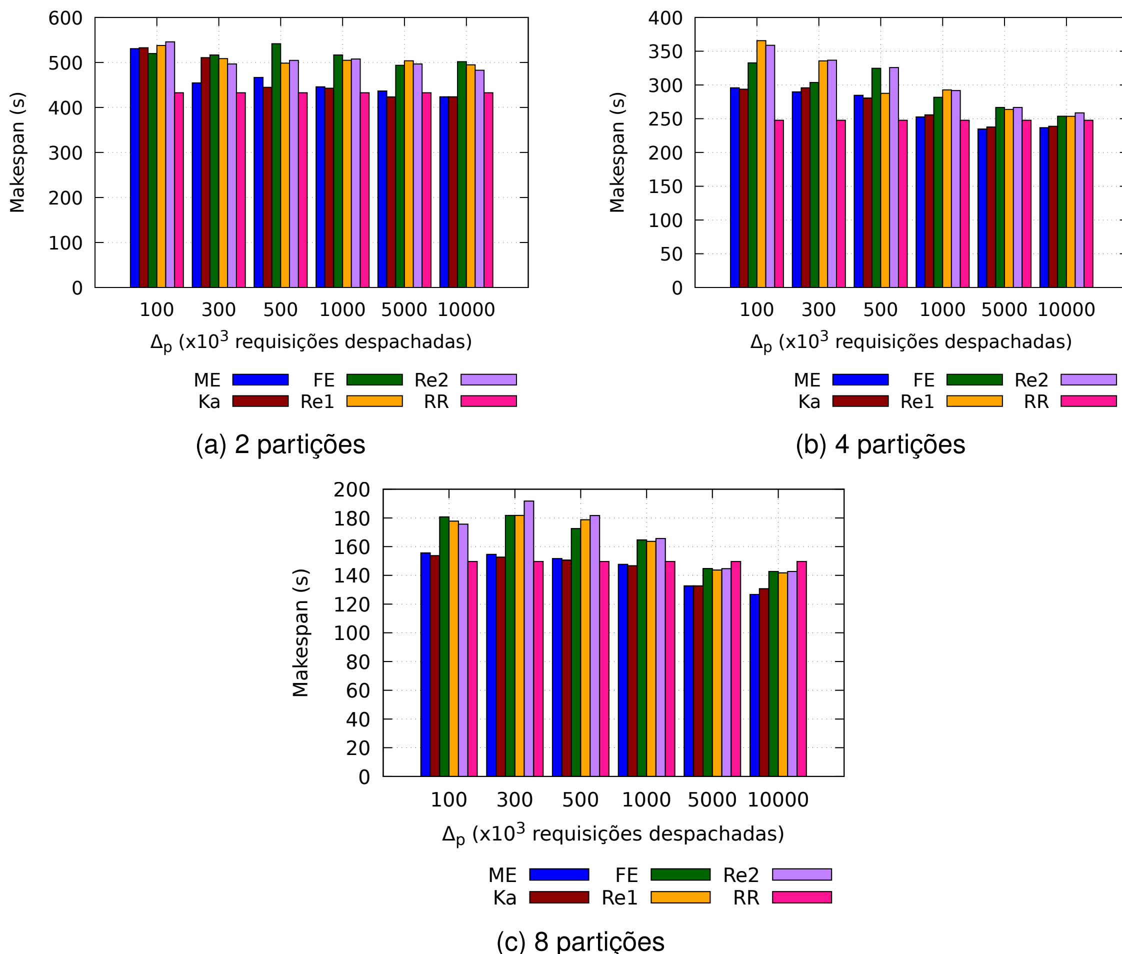
No protótipo de execução local, o algoritmo de consenso é abstraído e é considerado que o sistema já possui uma fila de requisições ordenadas e determinísticas. O objetivo é analisar a execução do particionamento em uma única réplica, e não em um sistema distribuído, que implicaria em outros custos como o protocolo de consenso e comunicação via rede. O objetivo principal dessa análise é examinar o grau de paralelismo oferecido por cada um dos algoritmos, e a sobrecarga causada pela execução do algoritmo. Para fazer isso, o *Makespan* (tempo total de execução) e vazão foram analisados sob diferentes cenários.

A Figura 5 mostra o *makespan* observado pela carga de trabalho YCSB-A. As Figuras 5(a), 5(b), e 5(c) mostram as configurações de 2, 4 e 8 *threads*/partições, respectivamente, variando o valor de Δ_p utilizado durante a execução. As mesmas requisições foram processadas em todos os testes, ao todo foram 50.000.000 requisições para cada caso de teste. É possível perceber que, no geral, houve melhora no desempenho com o aumento no número de partições. Enquanto mais de 6 minutos são necessários para processar todas as requisições com 2 partições, aproximadamente 2 minutos são necessários para processar as mesmas requisições com 8 partições. Como pode ser observado, em cenários com poucas partições e particionamento frequente, o *makespan* do KaHIP e METIS estão acima do *Round-Robin*, enquanto FENNEL e ReFENNEL tiveram os valores ainda mais altos. Uma vez que todas as requisições são de variável única, o algoritmo *Round-Robin* distribui chaves de uma maneira bastante balanceada, então o custo de reparticionamento nem sempre é compensado pela possível melhora de desempenho. Conforme Δ_p aumenta, o sistema passa menos tempo calculando uma nova partição, e com um Δ_p suficientemente grande e um alto número de *threads*, é possível que o reparticionamento comece a mostrar um *makespan* menor do que o do *Round-Robin*, como mostrado na Figura 5(c).

O *makespan* usando algoritmos de reparticionamento também aumenta significativamente com a adição de mais vértices, como pode ser visto na Figura 6. Com uma maior quantidade de vértices para particionar, é necessário mais tempo para o sistema

voltar a executar requisições depois que um reparticionamento é iniciado, aumentando o tempo necessário para executar todas as requisições. A Figura 6(c) mostra como o reparticionamento pode fortemente afetar o tempo de execução conforme o grafo cresce. Com repartições com frequência de 100.000 de requisições, o *Round-Robin* mostrou um *makespan* em torno de 5 vezes menor quando comparado ao *makespan* mais baixo de outros algoritmos. Quando a frequência de reparticionamento diminui, porém, o tempo de execução de todas as estratégias é comparável. Dentre os algoritmos de particionamento, uma tendência de *makespan* similar à vista na Figura 6 é percebida. Algoritmos de *streaming* mostraram um tempo de execução ligeiramente maior do que os algoritmos multiníveis, com o KaHIP apresentando o menor *makespan*.

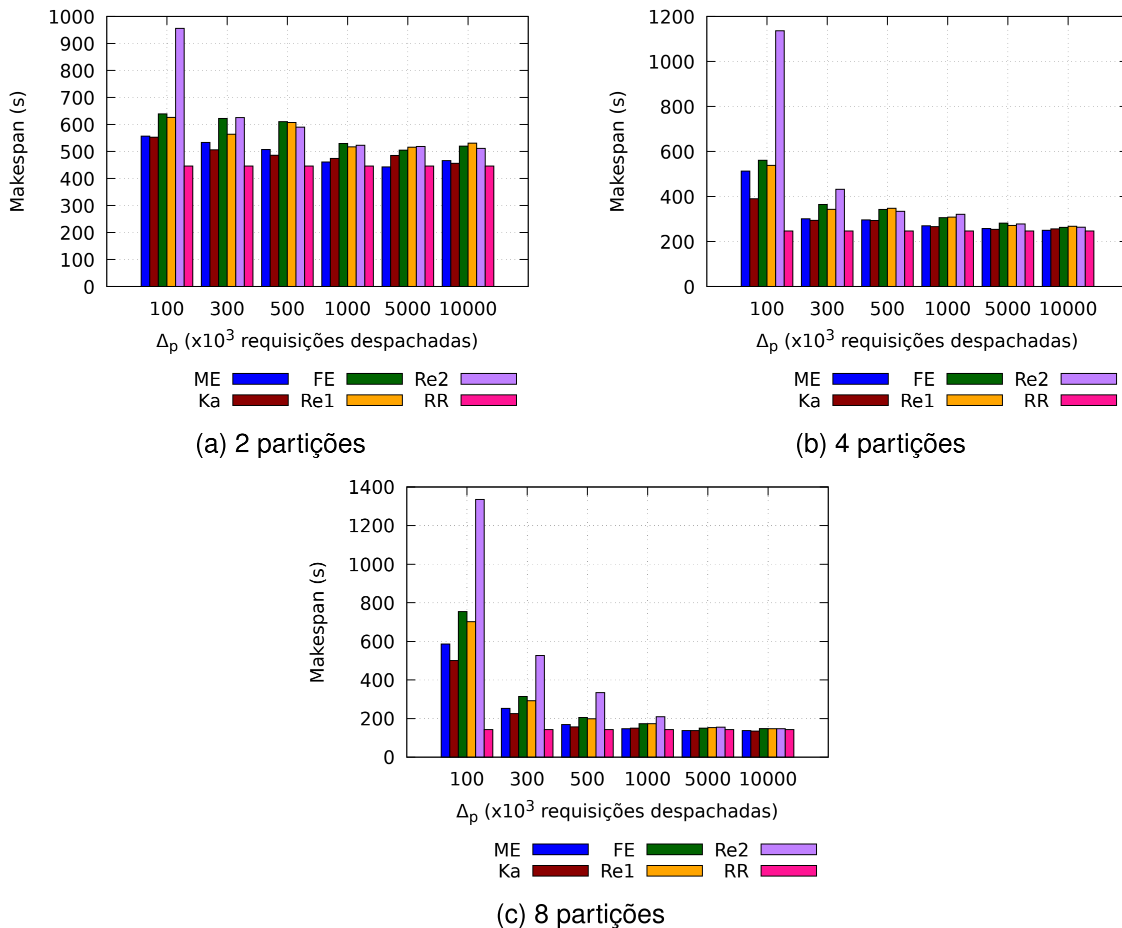
Figura 5 – *Makespan* observado com YCSB-A para diferentes intervalos de reparticionamento, com grafo de 1.000 de vértices.



Fonte: Elaborado pelo autor (2021).

Existe uma tensão entre os benefícios do reparticionamento e seu custo. Enquanto reparticionamentos periódicos podem melhorar o escalonamento baseado na variação da carga de trabalho, sua execução causa uma interrupção momentânea na execução de requisições. Para ilustrar esse efeito, a Figura 7 apresenta o gráfico da

Figura 6 – *Makespan* observado com YCSB-A para diferentes intervalos de reparticionamento, com grafo de 1.000.000 de vértices.



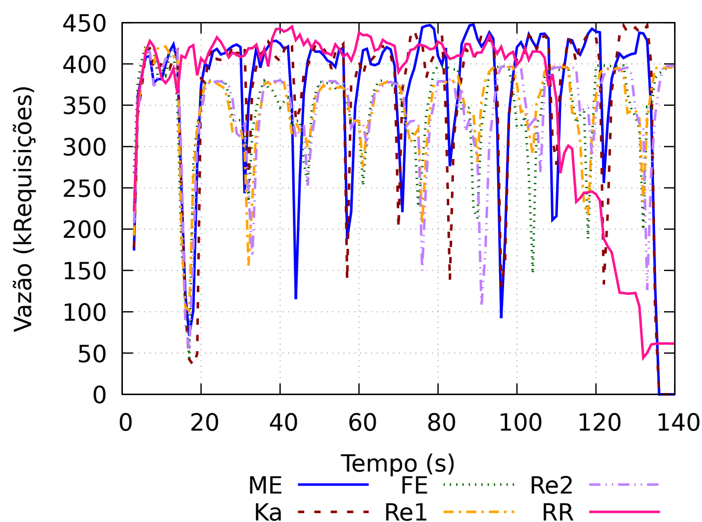
(a) 2 partições (b) 4 partições

(c) 8 partições

Fonte: Elaborado pelo autor (2021).

vazão pelo tempo para o cenário YCSB-A com 8 partições e Δ_p = 5.000.000. Como pode ser observado, em todos os algoritmos, existem vales na vazão. Esses valores baixos observados revelam os instantes em que os algoritmos de reparticionamento estão sendo executados, todos os quatro algoritmos tiveram vales de aproximadamente mesmo tamanho. Quando esses vales ocorrem com frequência, o sistema passa a maior parte do tempo reparticionando, mas quando esses vales são espaçados existe mais tempo para que novas partições executem suas requisições. Isso ajuda a justificar o ganho de desempenho com um alto Δ_p mencionado previamente, com um Δ_p alto repartições acontecem menos frequentemente e existe tempo para perceber os benefícios de novos particionamentos, mesmo que pequenos. Existia o pensamento de que talvez o FENNEL e ReFENNEL gastariam menos tempo reparticionando, por se tratarem de algoritmos gulosos, mas este efeito não foi observado nesse cenário. Já que o grafo é pequeno e cabe inteiramente em memória, tanto METIS quanto KaHIP tiveram um tempo de execução comparável aos algoritmos que utilizam estratégias gulosas.

Figura 7 – Vazão observada com YCSB-A com 8 partições e reparticionamento a cada 5,000,000 de requisições. Tamanho do grafo de 1.000 vértices.

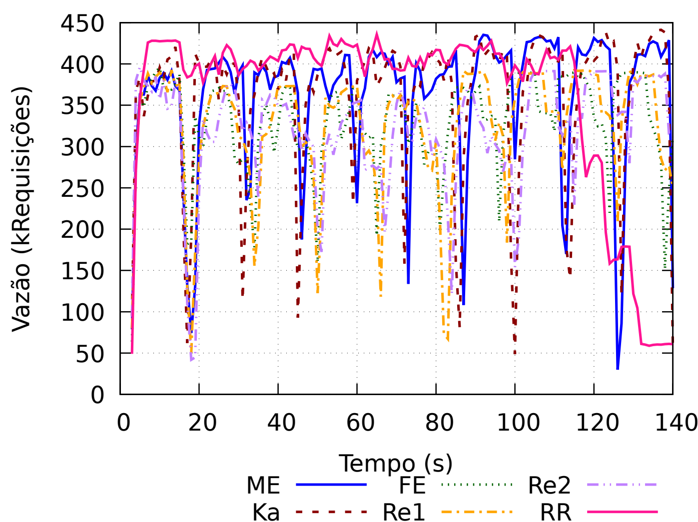


Fonte: Elaborado pelo autor (2021).

A Figura 8 mostra o cenário equivalente ao da Figura 7, mas com 1.000.000 de vértices. Apesar de existir uma pequena diferença na vazão do *Round-Robin*, vales formados pelo tempo gasto reparticionamento são, em geral, menores. Apensar do KaHIP criar vales menores comparados aos outros algoritmos, ele também tem o maior ganho de vazão após reparticionamentos, ultrapassando levemente o *Round-Robin* após 80 segundos de execução. Isso mostra que apesar de tomar mais tempo para particionar, o particionamento resultante apresenta um melhor paralelismo, explicando o *makespan* menor visto na Figura 6. No lado oposto está o FENNEL. Apesar de perder pouco tempo durante o reparticionamento, o FENNEL quase não ganha vazão depois dele, resultando em *makespans* maiores.

A Figura 9 mostra um diagrama de caixas com a vazão observada com a carga de trabalho YCSB-A. Uma caixa é desenhada em volta da região entre o primeiro e terceiro quartil, e a linha horizontal representa o valor da mediana. Linhas verticais estendem-se dos limites da caixa até os pontos mais distantes em y. Com um pequeno número de partições, como mostrado na Figura 9(a), a caixa *Round-Robin* mal é visível. O baixo desvio padrão é esperado, já que o *Round-Robin* não causa interrupções momentâneas no escalonamento durante a execução. Com o aumento do número de partições, porém, o desbalanceamento causado pelo particionamento *Round-Robin* torna-se perceptível. Observe que o desvio padrão aumenta para o *Round-Robin* com 4 e 8 partições. Os outros algoritmos demonstram flutuações visíveis na vazão independente do número de partições em uso. Isso é uma consequência da sincronização entre *threads* causada pelo reparticionamento periódico. Perceba que essa sobrecarga é compensada pelo melhor balanceamento de requisições em partições. Esse é o caso

Figura 8 – Vazão observada com YCSB-A com 8 partições e reparticionamento a cada 5,000,000 de requisições. Tamanho do grafo de 1.000.000 vértices.



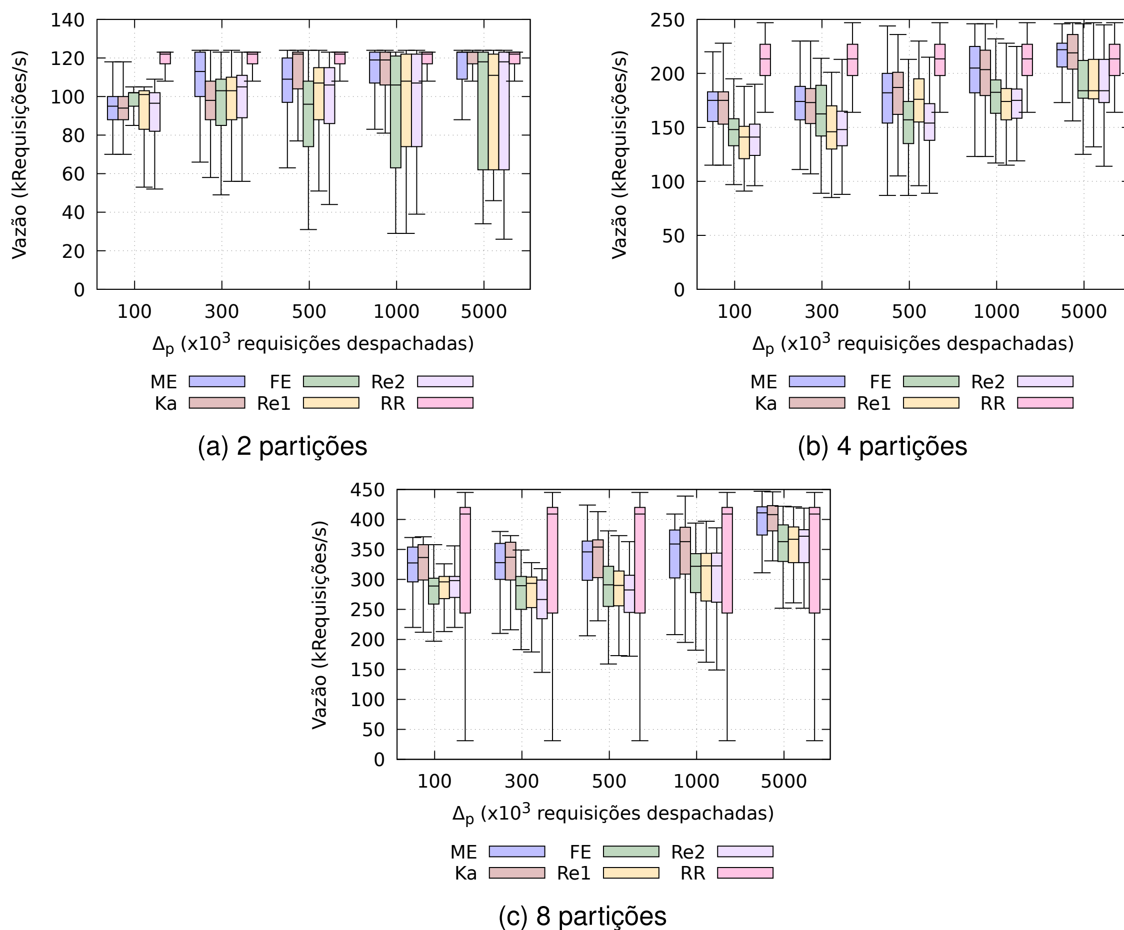
Fonte: Elaborado pelo autor (2021).

para METIS e KaHIP, que apresentaram valores de mediana levemente superiores do que *Round-Robin* para Δ_p = 5.000.000, e 4 ou 8 partições.

A Figura 10 mostra um gráfico de caixas similar à Figura 9, mas com um grafo de 1,000,000 vértices. O padrão de vazão permanece similar, com estratégias com reparticionamento mostrando uma alta flutuação na vazão, causada pelo tempo de reparticionamento e as sincronizações que ele causa. Conforme o aumento do número de partições e o decréscimo de Δ_p, todos os algoritmos de particionamento têm uma média de vazão menor do que com *Round-Robin*, causada pelo alto tempo de reparticionamento causado pelo aumento do número de vértices.

De maneira análoga ao experimento anterior, a Figura 11 mostra o *makespan* observado para diferentes intervalos de reparticionamento e número de partições, mas para a carga de trabalho YCSB-D, com 1.000 chaves iniciais. A carga de trabalho YCSB-D muda o intervalo de chaves mais prováveis a serem acessada de acordo com o tempo, então ela apresenta uma carga de trabalho desafiadora para os algoritmos de reparticionamento. Como antes, a carga de trabalho é composta por 50.000.000 requisições. Como 5% das requisições são inserções, até o final da execução o gráfico é composto por 2.501.000 vértices. Uma vez que as estratégias de reparticionamento adotadas na pesquisa consideram apenas requisições passadas para tomar decisões sobre o novo particionamento, elas não apresentam melhoras no *makespan* comparadas ao particionamento estático. Em adição a isso, o particionamento estático é executado uma única vez, no começo, enquanto outras estratégias reexecutam a cada intervalo Δ_p. Como pode ser observado, o *Round-Robin* supera as estratégias de reparticionamento em todos os casos de testes. Exceto no cenário com Δ_p = 100.000, o que

Figura 9 – Vazão observada com YCSB-A para diferentes intervalos de reparticionamento. Tamanho do grafo de 1.000 de vértices.



(a) 2 partições

(b) 4 partições

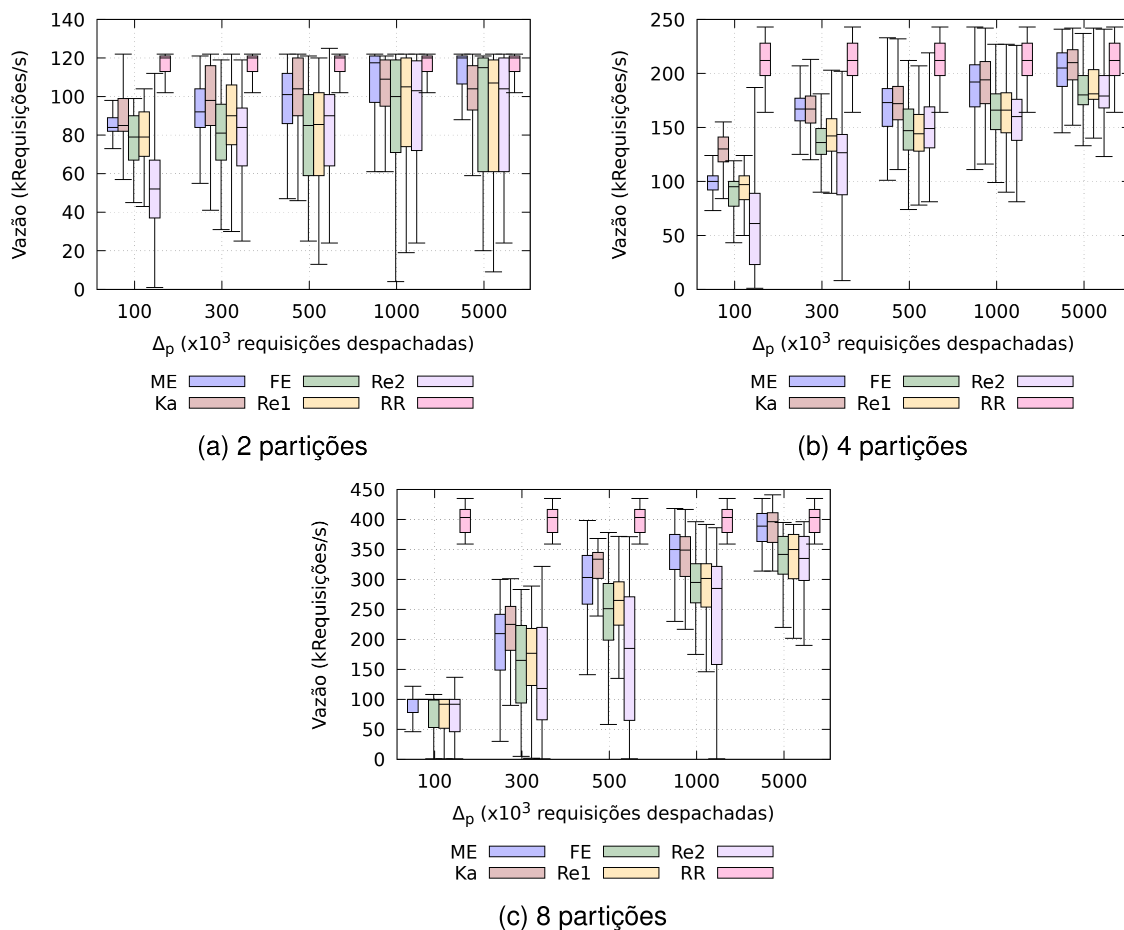
(c) 8 partições

Fonte: Elaborado pelo autor (2021).

representa reparticionamento muito frequente, METIS e ReFENNEL (Re1) apresentam constantemente um *makespan* menor do que KaHIP e FENNEL. Para o ReFENNEL (Re2), o tempo de execução de dois algoritmos foi muito caro nesse cenário, apresentando um *makespan* mais elevado. O *makespan* de METIS e KaHIP foram um pouco menores do que os do FENNEL e ReFENNEL, mas todos os algoritmos de reparticionamento mostraram consistentemente valores de *makespan* maiores do que Round-Robin. Enquanto em YCSB-A um grande número de partições e um Δ_p alto reduz o impacto do particionamento, e faz com que o desempenho ultrapasse o do *Round-Robin*, em YCSB-D essa estratégia faz com que os algoritmos convirjam para o *Round-Robin*, mas sem superá-lo. Isso significa que o ganho de desempenho ao aumentar Δ_p acontece apenas porque os métodos de reparticionamento estão se aproximando a não reparticionar, então o *Round-Robin* apresenta-se como a estratégia mais apropriada.

Figura 12 mostra a execução do mesmo cenário, mas com 1.000.000 de chaves iniciais. Uma vez que YCSB-D tem requisições de inserção, até o fim da execução o

Figura 10 – Vazão observada com YCSB-A para diferentes intervalos de reparticionamento. Tamanho do grafo de 1.000.00 de vértices.



(a) 2 partições

(b) 4 partições

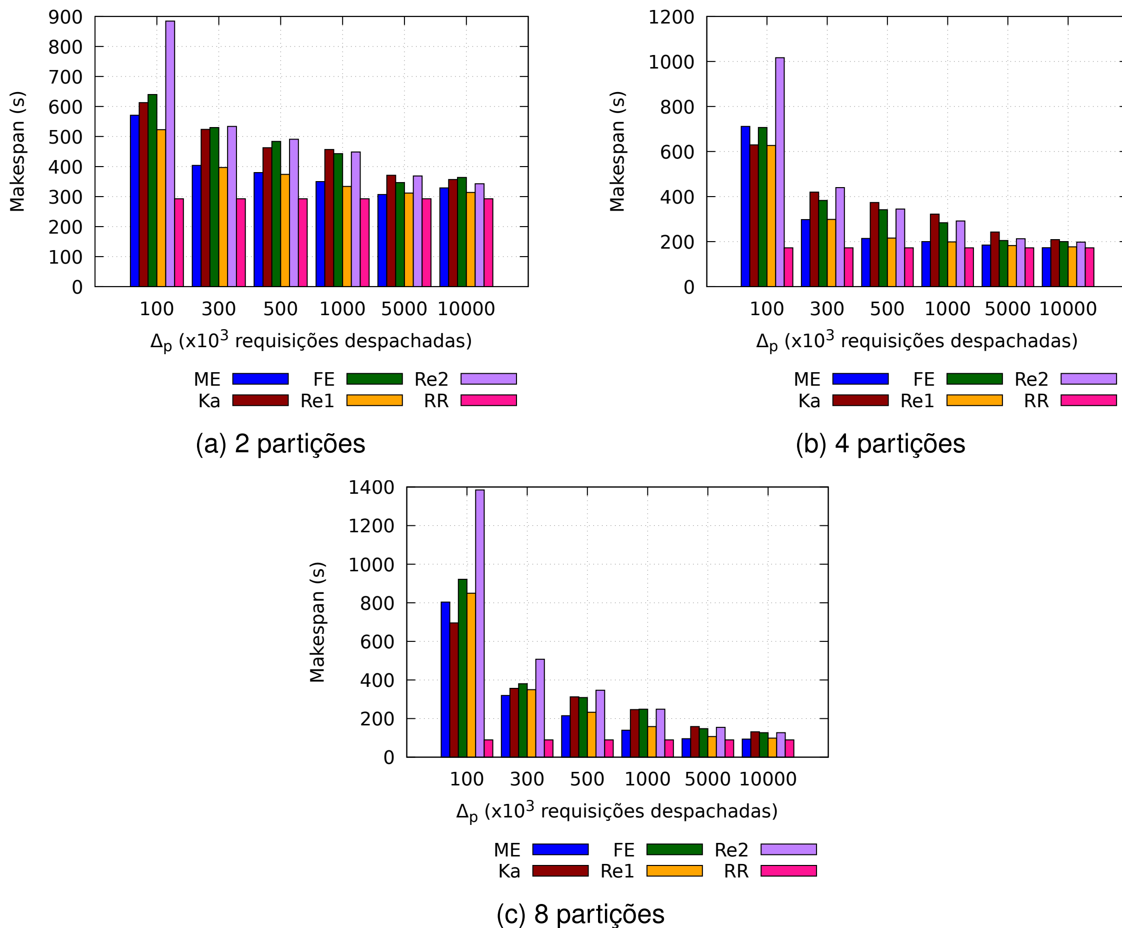
(c) 8 partições

Fonte: Elaborado pelo autor (2021).

tamanho do grafo será de 3.500.000 vértices. *Round-Robin* se mantêm com o menor *makespan* quando o sistema possui 1.000.000 de chaves iniciais. Já que o grafo com mais vértices demora mais para reparticionar. todos os algoritmos de reparticionamento mostraram um *makespan* expressivamente maior. Como visto na Figura 12(c), com um maior número de partições (8) e Δ_p pequeno (100,000), estratégias usando algoritmos de particionamento tomam pelo menos 600 segundos para executar todas as requisições, chegando a até 1.200 segundos. No mesmo cenário, o *Round-Robin* tem um *makespan* de aproximadamente 100 segundos. Assim como em 11, estratégias de particionamento começam a convergir no *Round-Robin* com um alto Δ_p. Novamente, isso acontece porque o sistema está chegando mais perto de não reparticionar, assim com o *Round-Robin*.

A Figura 13 mostra o *makespan* observado com a carga de trabalho YCSB-E, com 1.000 de chave iniciais. Diferentemente das cargas de trabalho prévias, onde foram executadas apenas requisições de variável única, em YCSB-E 95% das requisições são escaneamento de intervalos (i.e., requisições multi-variáveis) e 5% inserções.

Figura 11 – *Makespan* observado com YCSB-D para diferentes intervalos de reparticionamento. Grafo com 1.000 de vértices.

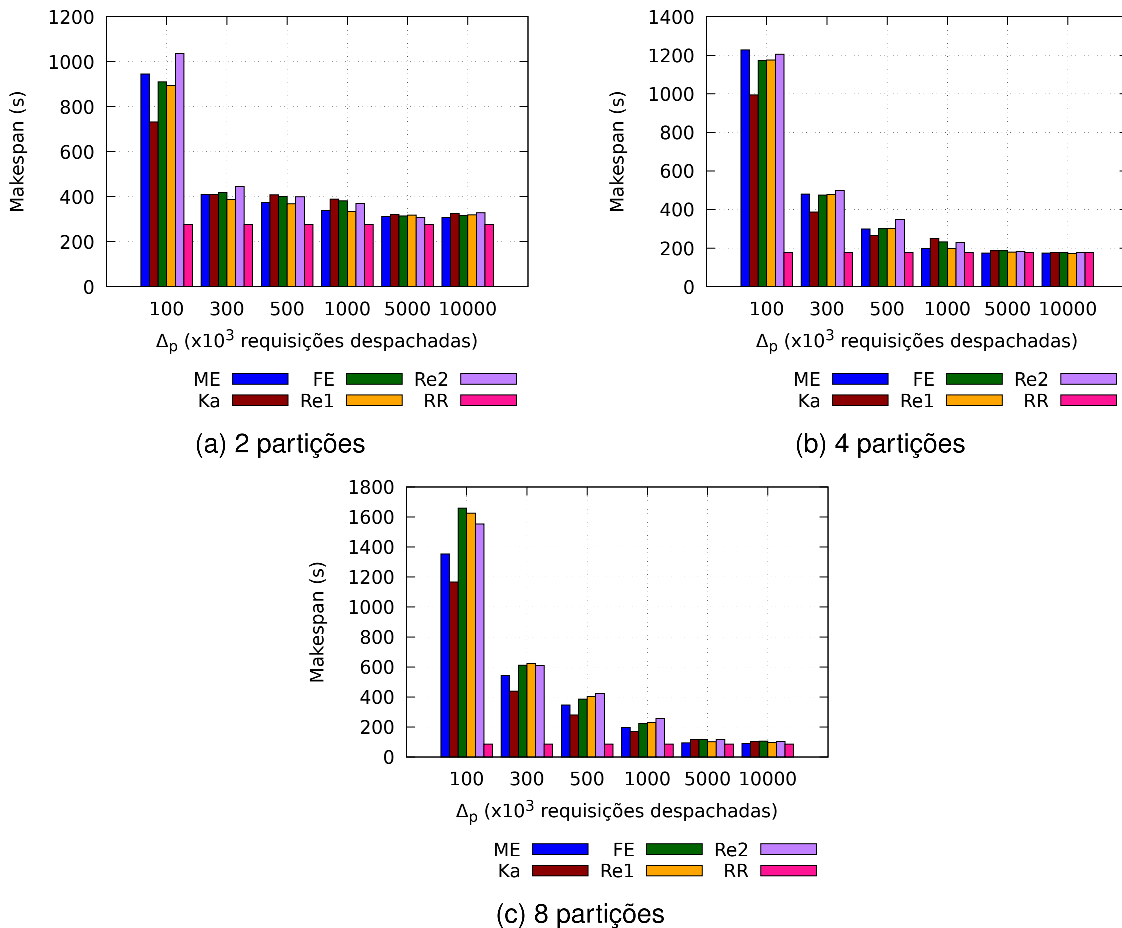


(a) 2 partições (b) 4 partições (c) 8 partições

Fonte: Elaborado pelo autor (2021).

Para essa carga de trabalho, um total de 5.000.000 de requisições foram executadas. Nesses experimentos os escaneamento de intervalos foram configurados para escanear de 2 a 8 chaves consecutivas. Aqui o *Round-Robin* tem uma alta desvantagem, uma vez que a maioria das requisições requerem sincronização de pelo menos 2 partições. Algoritmos de reparticionamento mostram uma grande melhora no *makespan*, uma vez que movem chaves frequentemente acessadas juntas para a mesma partição, reduzindo as sincronizações e, consequentemente, a ociosidade das *threads*. FENNEL mostrou um dos menores valores para *makespan*, seguido pelo METIS e KaHIP, nessa ordem. Pode ser visto que o ReFENNEL (Re1) mostrou algum ganho comparado ao *Round-Robin*, mas ainda permanece muito próximo a ele e longe de ter o mesmo desempenho mostrado pelos outros algoritmos. O ReFENNEL (Re2), porém, tem resultados muito similares ao FENNEL. Uma vez que as requisições são mais caras computacionalmente do que aquelas vindas do YCSB-A e D, um baixo Δ_p nesse cenário aumenta o desempenho. Com Δ_p = 10.000.000, a execução leva mais do que duas vezes o tempo tomado pela execução com Δ_p = 100.000, em cenários com 4

Figura 12 – *Makespan* observado com YCSB-D para diferentes intervalos de reparticionamento. Grafo com 1.000.000 de vértices.



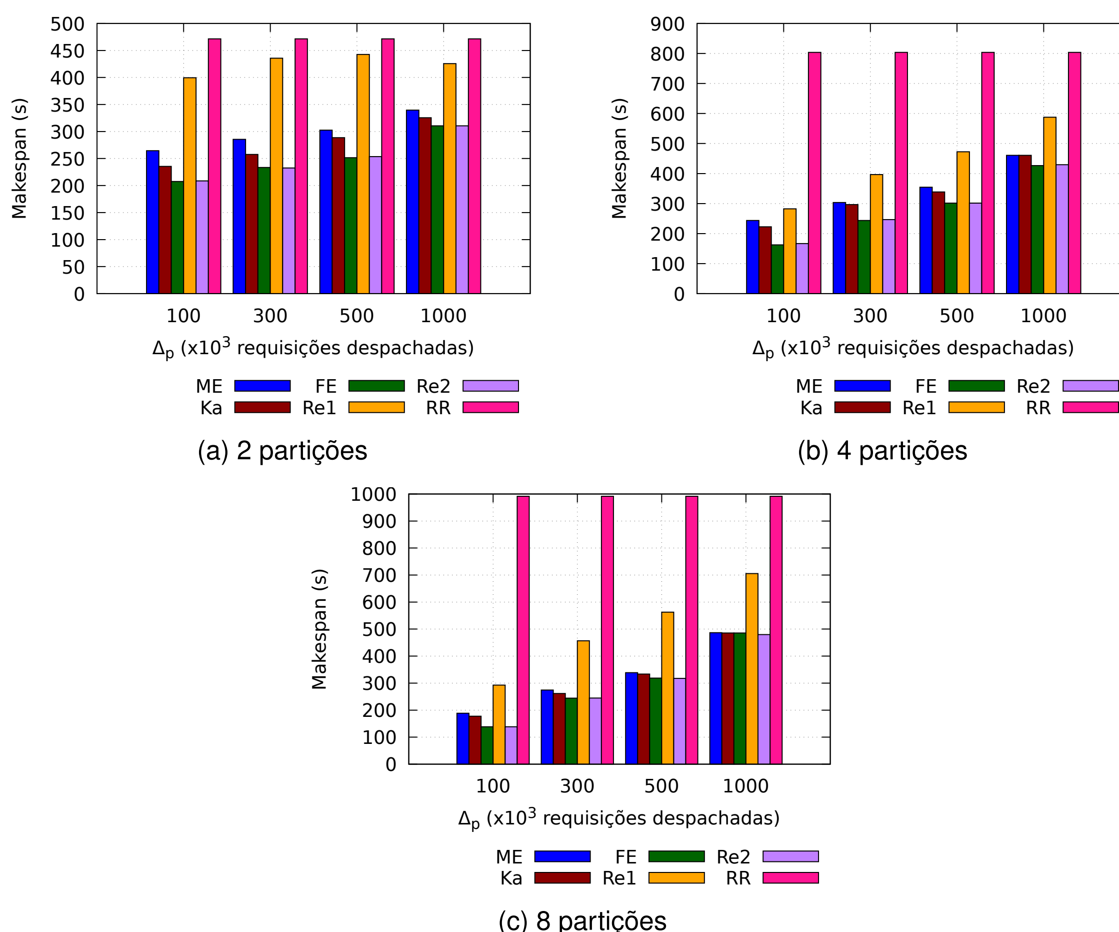
(a) 2 partições (b) 4 partições (c) 8 partições

Fonte: Elaborado pelo autor (2021).

e 8 partições. Outra observação que pode ser feita é que, enquanto o *Round-Robin* mostrou um aumento no *makespan* com a adição de mais partições, os algoritmos de particionamento tiveram seus valores de *makespan* reduzidos. O tempo de execução do FENNEL com Δ_p = 100.000 é por volta de 200 segundos com 2 partições, enquanto com 8 partições esse valor é reduzido para pouco acima de 100.

O aumento no número de vértices não impacta negativamente o sistema. A Figura 14 mostra o mesmo cenário, mas com 1,000,000 de chaves, isso é, um grafo com 1.000.000 de vértices. Como 5% das requisições são inserções, o grafo terá 3.500.000 vértices ao final da execução. Mesmo com um pequeno número de partições, onde *Round-Robin* tem o seu melhor desempenho, ele ainda mostra um *makespan* quase duas vezes maior do que as outras técnicas, como pode ser visto na Figura 14(a). Mais uma vez, os algoritmos de reparticionamento são altamente beneficiados por um grande número de partições e pequeno Δ_p. Quando requisições multi-vaiáveis são predominantes, reparticionar frequentemente para evitar sincronizações permanece benéfico, mesmo com um grafo maior. Como pode ser visto em todos os cenários,

Figura 13 – *Makespan* observado com YCSB-E com diferentes intervalos de reparticionamento. Tamanho do grafo de 1.000 de vértices.

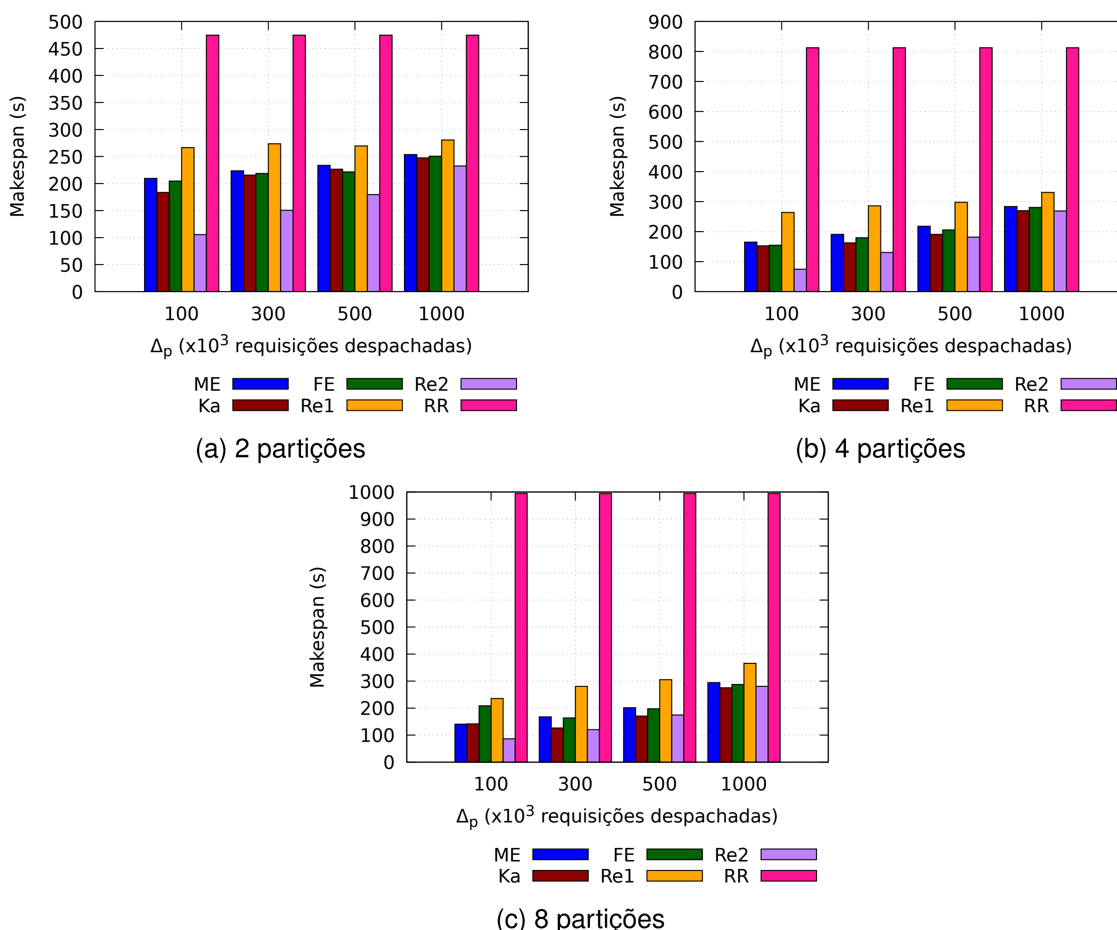


Fonte: Elaborado pelo autor (2021).

ReFENNEL (Re1) permanece o algoritmos com o maior *makespan*, seguido do METIS. KaHIP e FENNEL mostraram resultados comparáveis entre os cenários, com KaHIP resultando em um *makespan* levemente menor. ReFENNEL (Re2) funcionou muito bem nesse cenário, visto que o FENNEL já teve bons valores de *makespan*. Então, nesse caso, o tempo de executar dois algoritmos sequencialmente não prejudica a performance do sistema, mas a melhora.

A Figura 15 apresenta a vazão pelo tempo para a carga YCSB-E, no cenário com 8 partições e Δ_p = 500.000. Antes do primeiro reparticionamento todas as técnicas seguem o mapeamento de partições inicial, dados por *Round-Robin*. Como observado, essa estratégia possui um desempenho ruim, com uma vazão de aproximadamente 5.800 requisições por segundo. Depois da primeira repartição, pode ser observada uma melhora considerável no desempenho de todas as técnicas dinâmicas. Isso é esperado, uma vez que escaneamentos de intervalos acessam chaves consecutivas, o que representa o pior caso para o *Round-Robin*, onde chaves sucessivas são mapeadas para partições vizinhas. Interessantemente, a vazão logo após o repar-

Figura 14 – *Makespan* observado com YCSB-E com diferentes intervalos de reparticionamento. Tamanho do grafo de 1.000.000 de vértices.



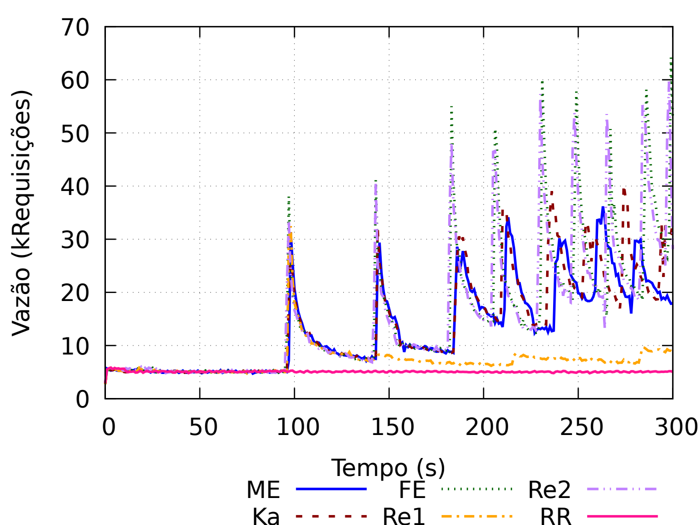
Fonte: Elaborado pelo autor (2021).

ticionamento atinge um pico e então diminui para uma vazão quase reta, formando sucessivos formatos de bacia. Esse aumento abrupto na vazão é consequência de um melhor reparticionamento, e também se beneficia do fato que as filas das *threads* estão vazias logo após o reparticionamento. Note que quando as *threads* se tornam mais ocupadas, a probabilidade de uma requisição na fila requerer sincronização com as outras *threads* aumenta. Esse efeito causa ócio para as *threads* que estão esperando para executar requisições de sincronização. Uma observação similar é discutida em (ALCHIERI *et al.*, 2017), onde os autores perceberam que diminuir a velocidade de escalonamento de requisições quando a ocupação das filas está alta pode ser uma boa escolha.

O formato de bacia ainda pode ser visto na Figura 16, que mostra o mesmo cenário mas com o grafo inicial composto por 1.000.000 de vértices (3.500.000, ao final da execução). A Figura 16(b) mostra que ambos METIS e FENNEL atingiram um pico local em sua vazão logo após o reparticionamento, e então essa vazão é reduzida até o próximo reparticionamento. O ReFENNEL (Re1) também mostrou picos em sua

vazão, apesar de permanecer por volta de 20.000 de requisições por segundo. O KaHIP mostra um particionamento inicial de alta qualidade, atingindo quase 50.000 de requisições por segundo logo após o primeiro reparticionamento em ambos cenários mostrados na Figura 16. Por outro lado, a Figura 16(a) mostra que o KaHIP também produz os menores vales. Isso faz sentido com os resultados na Figura 8, que também mostra o KaHIP produzindo partições de alta qualidade, mas com mais impacto na vazão. Para a carga de trabalho, o comportamento do ReFENNEL (Re2) foi superior dentre todos os algoritmos de particionamento. Com Re2 e $\Delta_p = 100,000$, todas as 5.000.000 de requisições foram processadas em menos de 80 segundos, enquanto após 140 segundos outros algoritmos ainda não terminaram. A Figura 16(a) também mostra que METIS, FENNEL e até ReFENNEL (Re1) tem um aumento estável na vazão durante a execução. Enquanto KaHIP mostra um alto pico de 60.000 requisições por segundo em 40 segundos, ele não mantém essa alta vazão. METIS e FENNEL, por outro lado, mantêm uma vazão um pouco menor, de aproximadamente 45.000 requisições por segundos, depois de 100 segundos de execução.

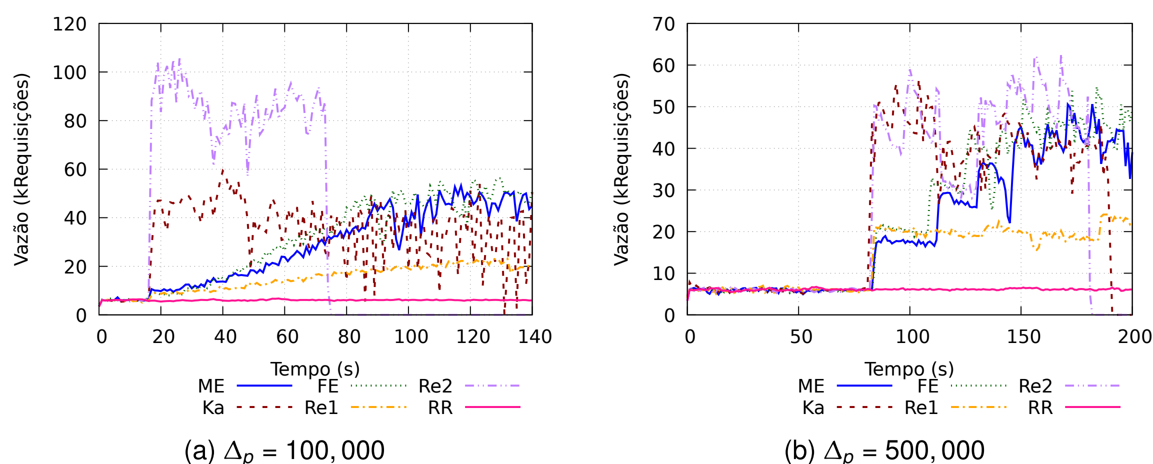
Figura 15 – Vazão observada por YCSB-E com 8 partições com repartições a cada 500,000 requisições. Tamanho do grafo de 1.000 de vértices.



Fonte: Elaborado pelo autor (2021).

Finalmente, a Figura 17 mostra o diagrama de caixas da vazão observada com a carga de trabalho YCSB-E. Em todos os cenários, especialmente naqueles com menor Δ_p, a vazão do *Round-Robin* permanece, no máximo, próximo aos menores valores das estratégias de reparticionamento. Todos os outros algoritmos aumentaram a vazão, apesar de apresentarem um alto desvio padrão. A Figura 18 mostra o mesmo gráfico para 1,000,000 de chaves iniciais. Nesse caso, o resultado para ReFENNEL (Re2) com 2 partições é impressionante. Sua menor vazão apresentada é maior do que a vazão máxima (KaHIP) mostrada por outros algoritmos. Apesar dos padrões dos valores da

Figura 16 – Vazão observada por YCSB-E com 4 partições. Tamanho do grafo de 1,000,000 de vértices.



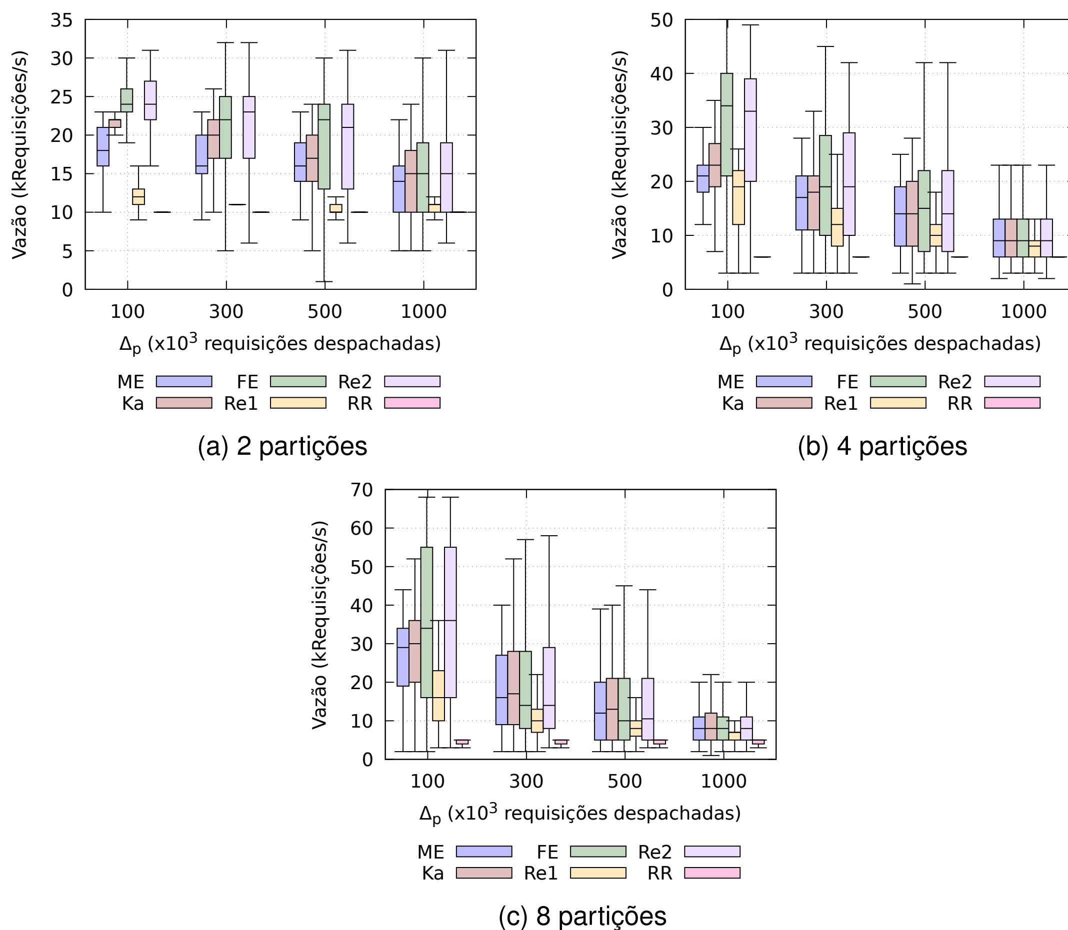
(a) $\Delta_p = 100,000$ (b) $\Delta_p = 500,000$

Fonte: Elaborado pelo autor.

vazão permanecerem similares aos vistos no gráfico anterior, o desvio padrão de todos os algoritmos é muito maior. Outra observação é que em alguns cenários, como como 4 partições e $\Delta_p = 100,000$, a mediana da vazão dos algoritmos de particionamento são maiores do que no mesmo cenário mas com 1.000 de vértices. Com 1.000 de vértices, a mediana da vazão do METIS é menor do que 25.000 requisições por segundo, enquanto para 1.000.000 de vértices essa vazão cresce para mais de 30.000 de requisições por segundo. Uma possível causa para essa diferença é que com mais vértices torna-se mais fácil acomodar vértices em partições apropriadas. Em adição à isso, o custo de sincronização entre *threads* é tal que o custo do particionamento é quase negligenciável nesse cenário, mesmo com um grafo grande.

Os resultados demonstraram que ambos METIS e KaHIP apresentam desempenhos comparáveis, sem um superior claro para grafos pequenos. Conforme o grafo cresce, o KaHIP toma mais tempo para executar, mas resulta em partições mais capazes de reduzir sincronização entre *threads*. Quanto aos algoritmos de *streaming*, ReFENNEL (Re1) atingiu resultados levemente melhores do que o FENNEL ao balancear requisições de variável única, enquanto o FENNEL obteve um desempenho amplamente maior ao balancear requisições multi-variáveis. O tempo extra gasto pelo ReFENNEL (Re2) é menos evidente em grafos maiores e ao balancear requisições multi-variáveis. Não vale a pena o utilizar para balancear requisições de variável única uma vez que o tempo extra de particionamento torna-se prejudicial. Em uma comparação geral, uma vez que todos os grafos cabiam em memória, os algoritmos não gulosos (METIS e KaHIP) tiveram desempenhos tão bons quanto os demais, atingindo melhores resultados em alguns cenários. Além disso, ficou claro pelos experimentos que uma boa estratégia de reparticionamento é fortemente dependente da carga de trabalho e tamanho do grafo. Intervalos esparsos entre reparticionamentos são preferí-

Figura 17 – Vazão observada com YCSB-E para diferentes intervalos de reparticionamento. Tamanho do grafo de 1.000 de vértices.



(a) 2 partições

(b) 4 partições

(c) 8 partições

Fonte: Elaborado pelo autor (2021).

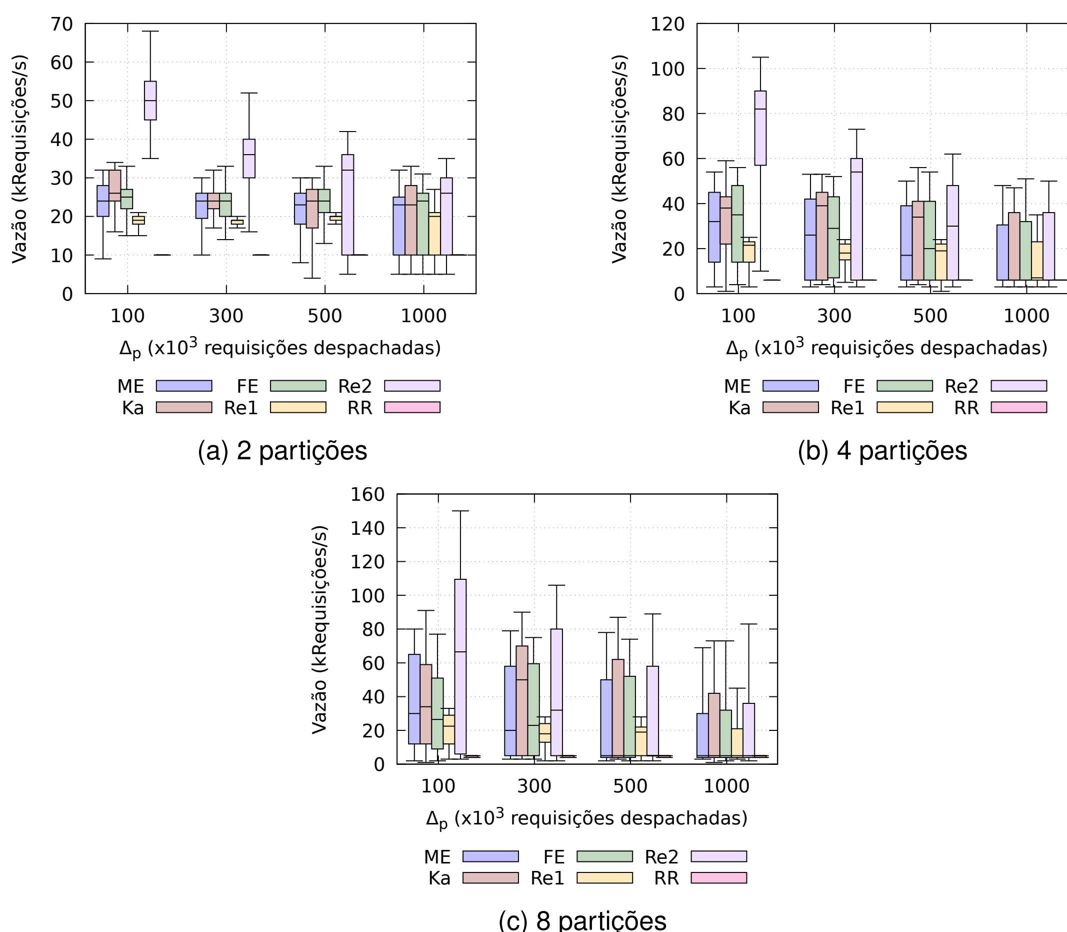
veis para acessos de variável única de alta vazão. Inversamente, reparticionamentos frequentes podem aumentar de forma significativa o paralelismo em cargas de trabalho com muitas requisições multi-variáveis. A observação sugere a adoção de mecanismos adaptativos, que possam trocar Δ_p de acordo com as flutuações da carga de trabalho e tamanho do grafo.

5.3.2 Protótipo distribuído

No protótipo distribuído, três máquinas são utilizadas como servidores, e uma faz o papel de cliente. O objetivo é analisar a execução do particionamento em um sistema próximo de um ambiente real, considerando custos causados pela transmissão de pacotes na rede e ordenamento das requisições.

Os três servidores comunicam-se entre si e ordenam as requisições utilizando o protocolo de consenso Paxos. A biblioteca LibPaxos (SCIASCIA, 2013) é utilizada como implementação do algoritmo Paxos. Nela, os servidores utilizam conexões TCP para a troca de mensagens do protocolo de consenso. O envio de requisições do

Figura 18 – Vazão observada com YCSB-E para diferentes intervalos de reparticionamento. Tamanho do grafo de 1.000.000 de vértices.



(a) 2 partições

(b) 4 partições

(c) 8 partições

Fonte: Elaborado pelo autor (2021).

cliente para as réplicas se dá através de uma conexão TCP estabelecida no começo da execução, enquanto os servidores respondem as requisições através de conexões UDP. Todo envio e recebimento de mensagens acontece através de *websockets*.

Para a avaliação, as métricas observadas foram latência e vazão. A vazão é medida no lado servidor, e representa a quantidade de requisições executadas em um dado período de tempo. A latência, por sua vez, é medida no cliente, e representa o tempo entre o envio de uma requisição e o recebimento de sua resposta.

No protótipo distribuído, os cenários executados iniciam com 1.000.0000 de chaves presentes no sistema. Isso implica em um grafo de 1.000.000 de vértices para a carga A, e 3.500.000 de vértices ao fim da execução das cargas D e E. Todos os cenários foram executados com uma carga total de 500.000 requisições.

Foram feitos testes com números de *threads* variáveis na máquina cliente para identificar o ponto de saturação do sistema. O intuito foi encontrar a quantidade de requisições máxima que o sistema é capaz de processar sem perder desempenho. Isso é feito para que os testes posteriores executem em cerca de 70% da capacidade

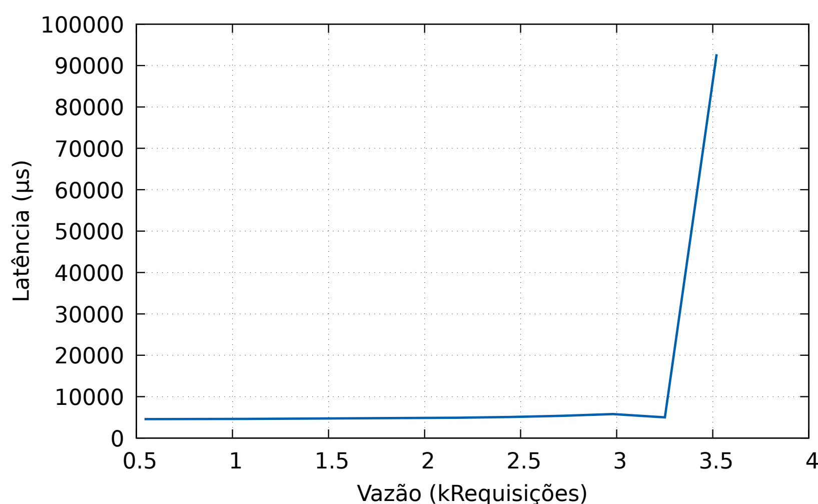
máxima do ambiente, simulando assim um serviço executando continuamente, que não está sempre no seu limite. Para medir o ponto de saturação, a execução segue o particionamento *Round-Robin*.

A Figura 19 mostra o gráfico de vazão por latência do sistema para a carga de trabalho YCSB-A. É possível perceber que no cenário com 4 partições, Figura 19(a), a latência do sistema aumenta consideravelmente quando a vazão é por volta de 3.250 requisições por segundo. Nesse ponto não existe mais aumento de vazão, indicando que o sistema está saturado. A Figura 19(b) mostra que o ponto de saturação para o mesmo cenário com 8 partições é próximo de 3.500 requisições por segundo. Portanto, para os testes com 4 partições, o sistema foi submetido a uma carga que resulta em uma vazão de aproximadamente 2.275 requisições por segundo executando *Round-Robin*. Testes com 8 partições foram executados com uma carga de aproximadamente 2.450 requisições por segundo executando *Round-Robin*.

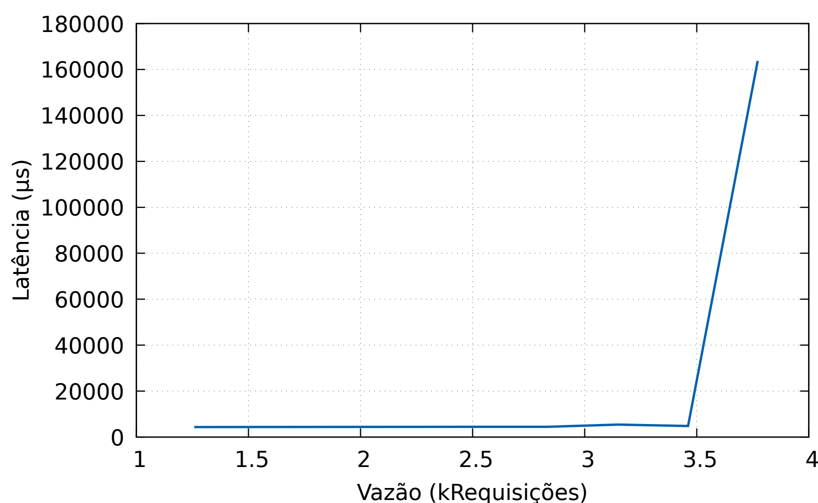
A Figura 20, que apresenta vazão e latência para Δ_p = 50.000 e 8 partições, mostra como os algoritmos de particionamento tiveram pouco impacto no desempenho do sistema. A Figura 20(a) mostra que durante a execução de todos os algoritmos a vazão cai para 0 por poucos segundos, causando um enfileiramento de requisições enquanto a réplica reparticiona. Após o fim do reparticionamento, o sistema possui uma longa fila de requisições a serem executadas. Como existem muitas requisições aguardando execução, ao término do reparticionamento a réplica apresenta altos picos na vazão, e retorna para a média de 2.500 requisições por segundo uma vez que o excedente é processado. O período em que a réplica suspende a execução para particionar causa grandes picos na latência, mostrados na Figura 20(b). O tempo de resposta de uma requisição chega em até 6 segundos, quando requisições são enviadas próximo ao instante de reparticionamento. Além disso, é possível observar como os algoritmos que passam mais tempo executando, isso é, as duas variações do ReFENNEL, também apresentam os maiores picos na latência, já que a réplica passa mais tempo sem processar requisições.

A Figura 21 mostra gráficos de caixa para a vazão observada para diferentes Δ_p. É possível perceber que, de maneira geral, o reparticionamento pouco afetou a vazão do sistema. Na Figura 21(a) é visto como a vazão permanece por volta de 2.450 requisições por segundo em todos os cenários. . Já no cenário com 8 partições, Figura 21(b), a vazão permanece sempre pouco acima de 2.500 requisições por segundo. O algoritmo ReFENNEL (Re2), porém mostrou uma variação muito maior na vazão apresentada para Δ_p = 50.000 . Isso é explicado uma vez que Re2 efetivamente executa dois algoritmos de particionamento sequencialmente, aumentando o tempo de execução. A variação decorre do comportamento visto anteriormente na Figura 20, onde as requisições são enfileiradas, zerando a vazão, e depois o sistema possui uma grande quantidade de requisições para serem executadas, causando um aumento

Figura 19 – Gráfico representando a vazão (eixo x) pela latência (eixo y), observados para a carga YCSB-A.



(a) 4 partições



(b) 8 partições

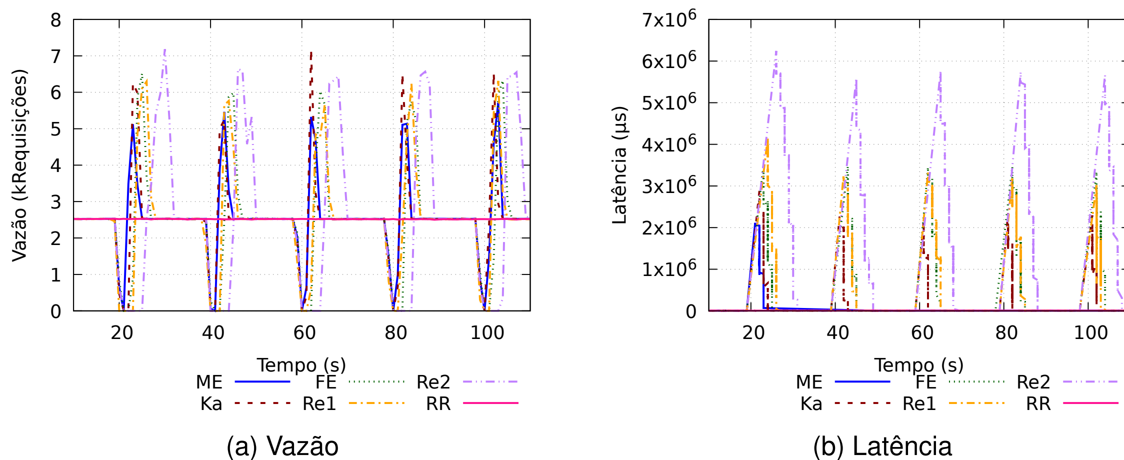
Fonte: Elaborado pelo autor (2021).

abrupto na vazão.

A Figura 22 foi redimensionada de forma a omitir alguns dados para melhor visualizar a variação da latência entre as técnicas. Isso é feito já que latências que chegam a até 6 segundos, como as apresentadas na Figura 20(b), prejudicam a visualização das latências menores. Para Δ_p = 50.000 os algoritmos de *streaming* apresentaram uma variação extremamente grande na latência, enquanto os algoritmos multinível apresentaram uma latência menos variável. Com o aumento do Δ_p a latência tende a estabilizar próximo aos números atingidos pelo Round-Robin, o que pode ser causado por estar mais próximo de não realizar particionamento.

A função de distribuição acumulada da latência é mostrada na Figura 23. Nela é possível comparar de forma mais clara os algoritmos de reparticionamento. O algoritmo

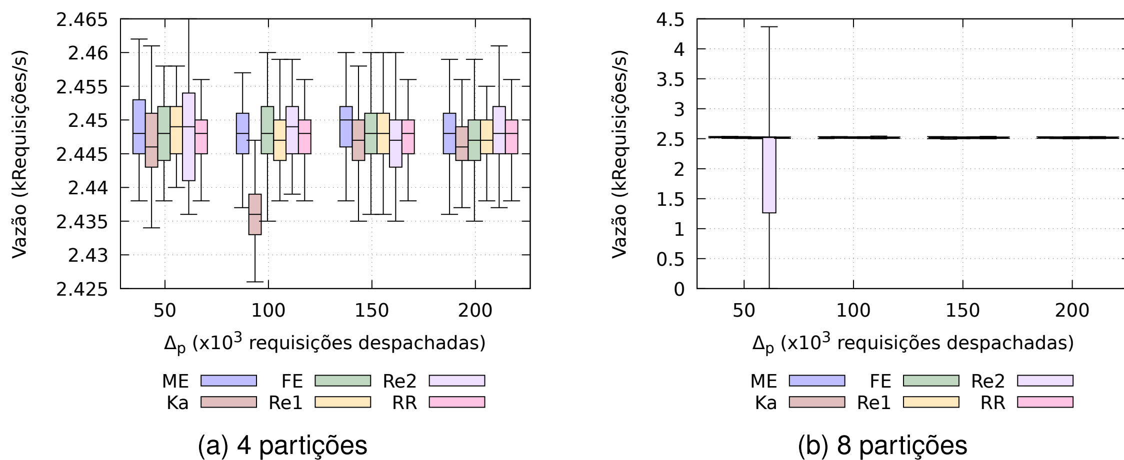
Figura 20 – Gráfico com métricas do sistema durante execução, observados para a carga YCSB-A com 8 partições e Δ_p = 50.000.



(a) Vazão (b) Latência

Fonte: Elaborado pelo autor (2021).

Figura 21 – Vazão observada com YCSB-A para diferentes intervalos de reparticionamento.



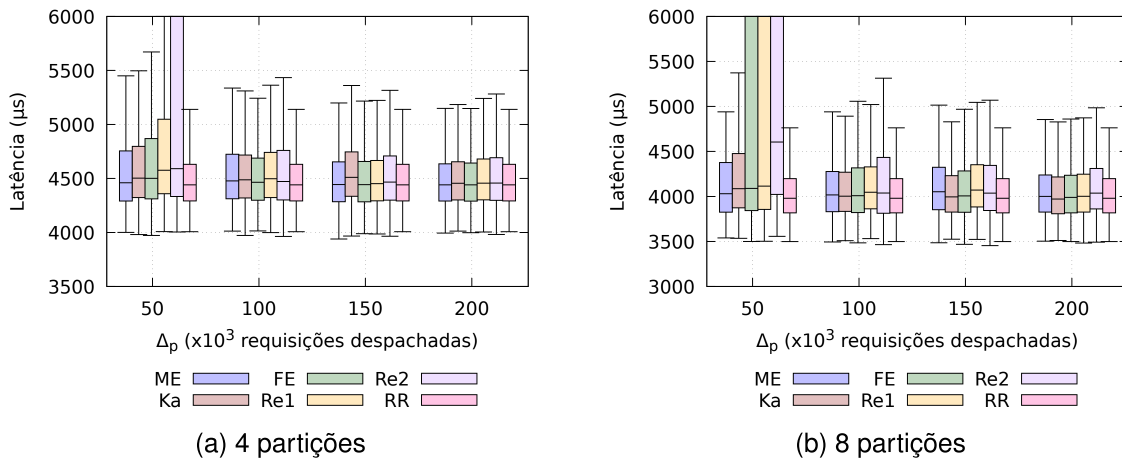
(a) 4 partições (b) 8 partições

Fonte: Elaborado pelo autor (2021).

ReFENNEL (Re2), como esperado devido ao maior tempo de particionamento, apresenta a latência mais alta. Apenas cerca de 60% de suas requisições apresentaram latências menores do que 1 segundo. Seguido, ReFENEL (Re1) e FENNEL apresentaram desempenhos comparáveis, com 80% de suas requisições tendo latência de até 1 segundo. Ambos METIS e KaHIP apresentaram desempenhos similares, com cerca de 90% das latências tendo valores menores do que 1 segundo. A latência elevada se deu principalmente pelo elevado tempo de particionamento, como *Round-Robin* não executa nenhum algoritmo de reparticionamento, todas as requisições são completas em poucos milésimos de segundo.

A execução do protótipo local já sugeriu que a carga YCSB-A apresenta pouco benefício ao utilizar técnicas de reparticionamento, criando vales na vazão durante

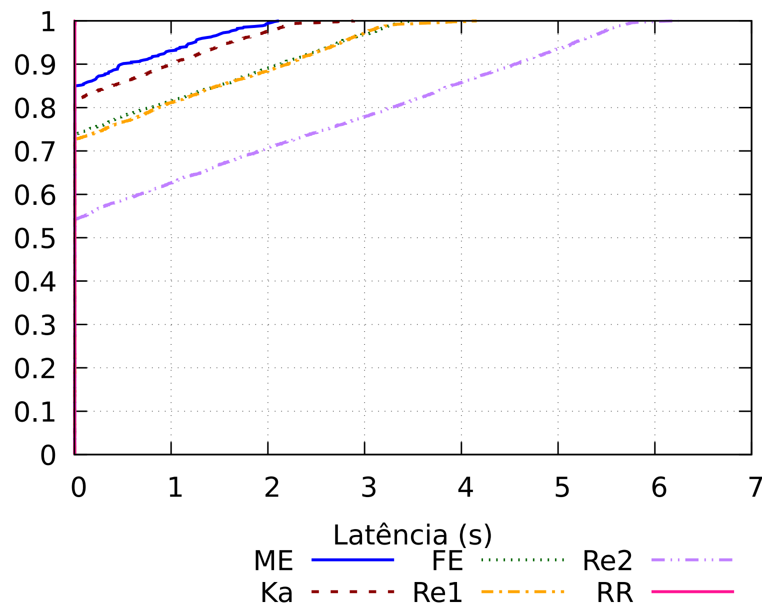
Figura 22 – Latência observada com YCSB-A para diferentes intervalos de reparticionamento.



(a) 4 partições (b) 8 partições

Fonte: Elaborado pelo autor (2021).

Figura 23 – Gráfico de função de distribuição acumulada para latência de YCSB-A com 8 partições e Δ_p = 50.000.



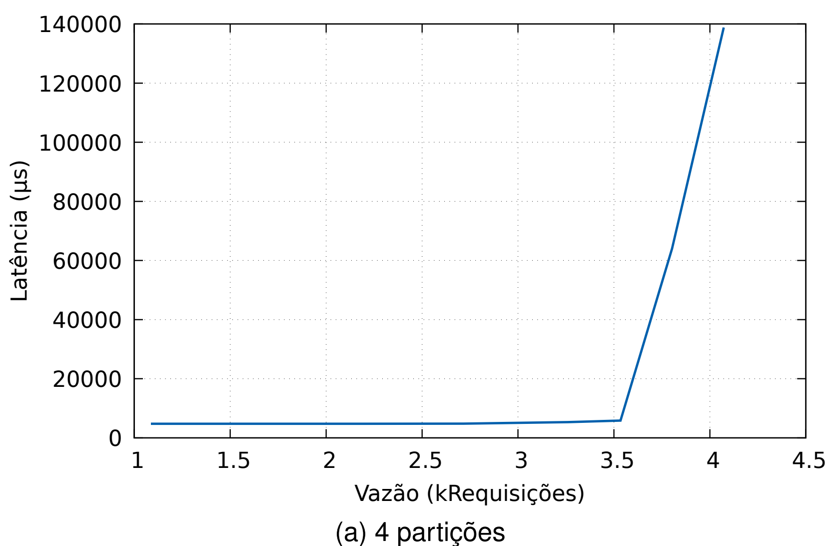
Fonte: Elaborado pelo autor (2021).

a execução. No protótipo distribuído, pelo fato do sistema não estar saturado desde o início, as réplicas conseguem criar picos de vazão após os vales, já que possuem capacidade de processamento para as requisições extras que foram enfileiradas. Depois que as requisições extras são executadas, os valores retornam para próximo ao *Round-Robin*. Esse comportamento faz com que as réplicas consigam compensar os vales, o que por sua vez faz com que a vazão da execução seja, em sua maioria, comparável ao *Round-Robin*. Por outro lado, a existência de vários vales e picos faz

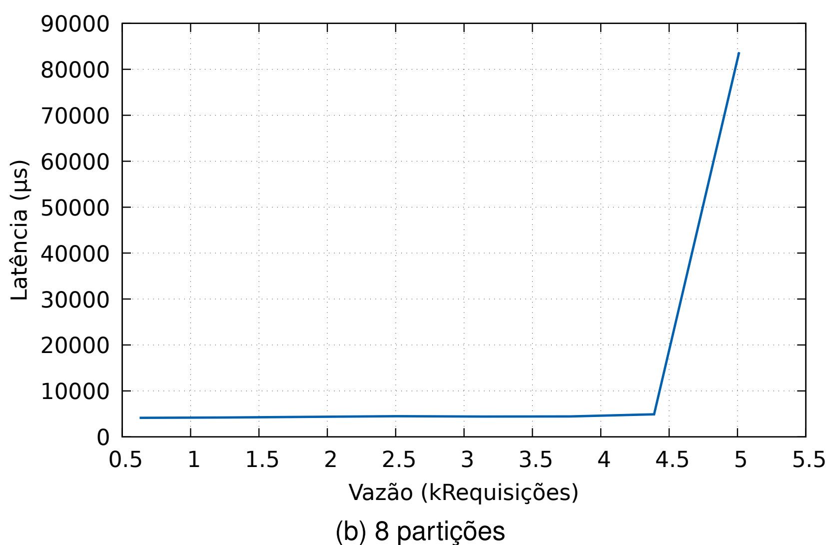
com que a latência sofra uma variação enorme caso a frequência de reparticionamento seja muito alta. Para todas as técnicas de reparticionamento, a variação da latência é reduzida expressivamente com a redução de Δ_p.

O gráfico de vazão pela latência para a carga YCSB-D pode ser visto na Figura 24. Novamente, a execução para a identificação do ponto de saturação é feita utilizando *Round-Robin*. Para o cenário com 4 partições, o ponto de saturação foi encontrado próximo a vazão de 3.500 requisições por segundo, portanto testes foram feitos para uma carga de aproximadamente 2.700 requisições por segundo. Para o cenário com 8 partições, o sistema fica saturado ao processar pouco menos de 4.500 requisições por segundo. Os testes foram então calibrados para executar em um cenário com vazão de 3.100 requisições por segundo.

Figura 24 – Gráfico representando a vazão (eixo x) pela latência (eixo y), observados para a carga YCSB-D.



(a) 4 partições

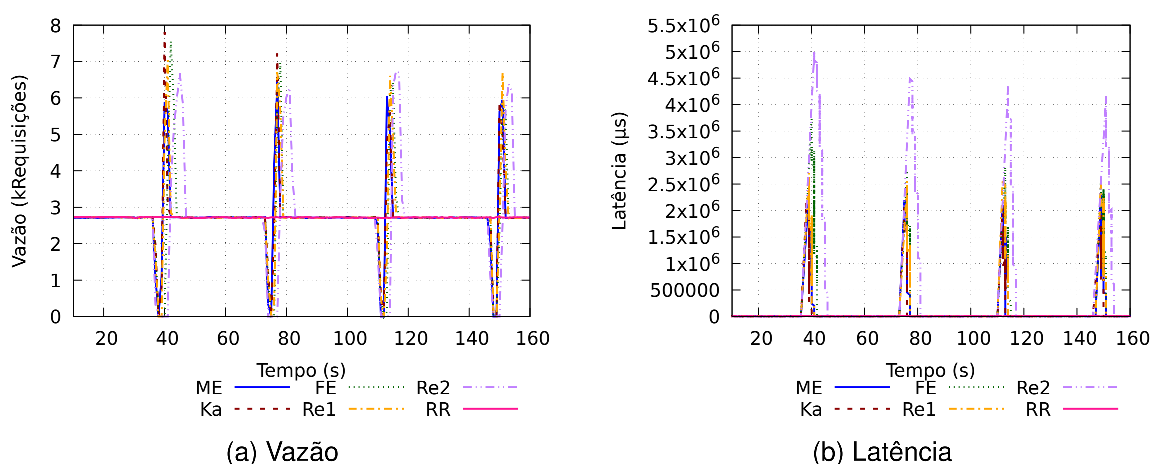


(b) 8 partições

Fonte: Elaborado pelo autor (2021).

O padrão de comportamento observado na carga de trabalho YCSB-D é similar à encontrada na carga YCSB-D. A vazão, como vista na Figura 25(a), apresenta os mesmos vales e picos, causados pelo tempo ocioso durante o reparticionamento e acumulo de requisições, respectivamente. Novamente, o algoritmo ReFENNEL (Re2) apresenta o maior tempo de reparticionamento, o que causa, como mostra a Figura 25(b), altas latências, que chegam a até 5 segundos. O algoritmo *Round-Robin* mantém uma vazão constante, e portanto não apresenta picos, na latência.

Figura 25 – Gráfico com métricas do sistema durante execução, observados para a carga YCSB-D com 4 partições e Δ_p = 100.000.



(a) Vazão (b) Latência

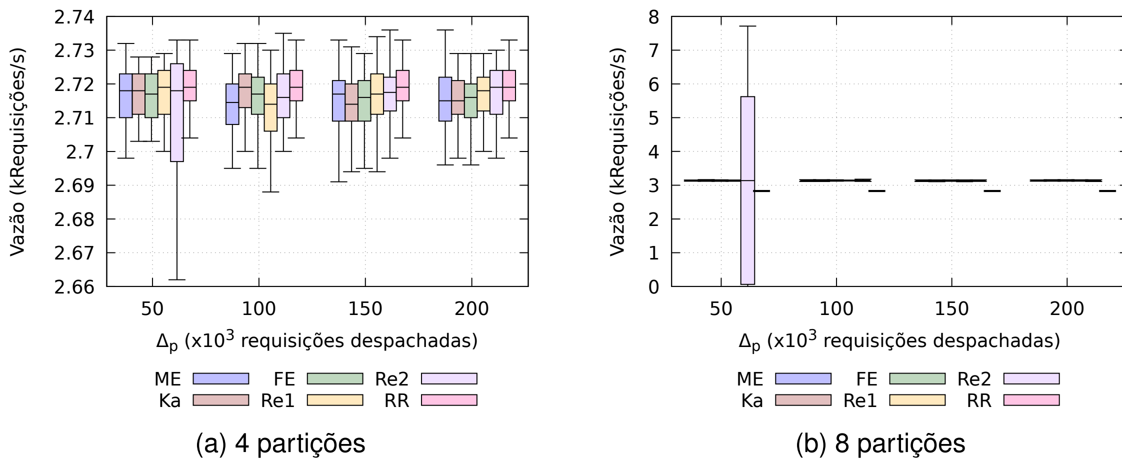
Fonte: Elaborado pelo autor (2021).

As observações da carga YCSB-D mostraram os mesmos padrões apresentados por YCSB-A, como mostra a Figura 26. Uma vez que os picos compensam os vales, a vazão de todos os algoritmos de particionamento permanece similar ao *Round-Robin*. O algoritmo ReFENNEL (Re2) continua como o mais lento, o que, com reparticionamentos frequentes, causa uma grande variação na vazão.

Mesmo com uma vazão similar ao *Round-Robin*, a variação da vazão faz com que a latência sofra uma variação ainda mais expressiva, assim como visto na carga de trabalho anterior. Na Figura 27 é possível ver como reparticionamentos frequentes causam uma grande variação na latência. O eixo *y* foi reduzido para melhor visualização dos valores, uma vez que a latência de algoritmos como ReFENNEL (Re1) atingem até 4 segundos. No cenário com 8 partições, o algoritmo METIS se destaca positivamente, sendo o algoritmo que causou menos impacto na latência do sistema, comparável ao *Round-Robin*.

Finalmente, o ponto de saturação para a carga YCSB-E. A Figura 28. É possível perceber como o sistema não apresenta aumento significativo na vazão pouco antes de 4.000 requisições por segundo para 4 partições, e pouco depois de 3.500 para 8 partições. A vazão apresentando uma redução em seu valor de acordo com o número de partições é uma tendência também observada no protótipo local. Isso acontece

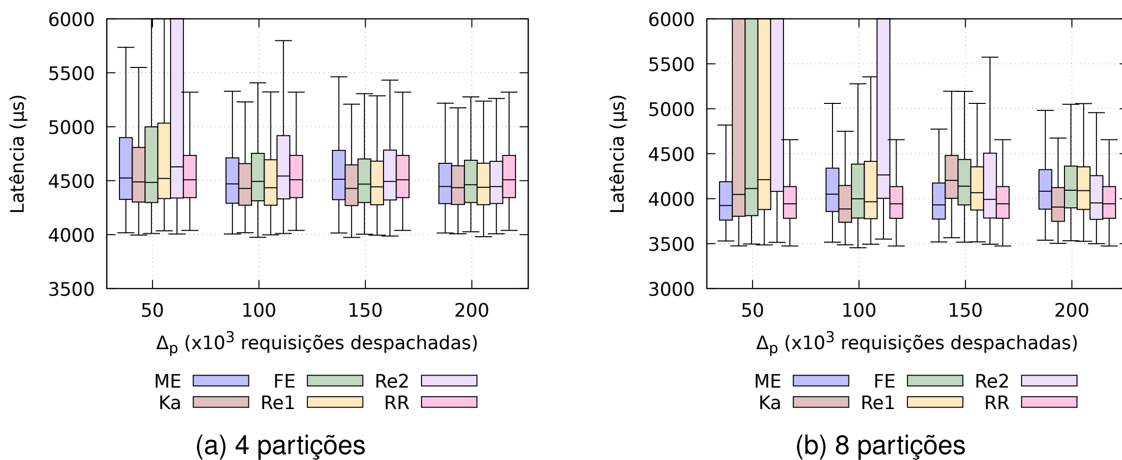
Figura 26 – Vazão observada com YCSB-D para diferentes intervalos de reparticionamento.



(a) 4 partições (b) 8 partições

Fonte: Elaborado pelo autor (2021).

Figura 27 – Latência observada com YCSB-D para diferentes intervalos de reparticionamento.



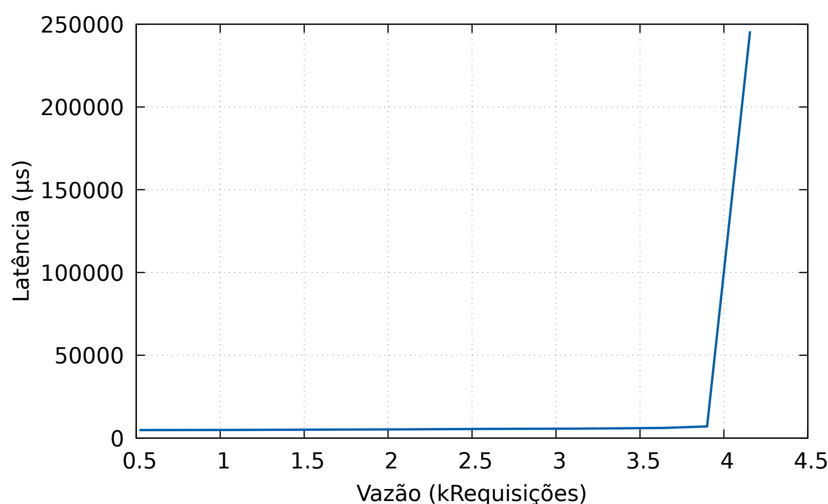
(a) 4 partições (b) 8 partições

Fonte: Elaborado pelo autor (2021).

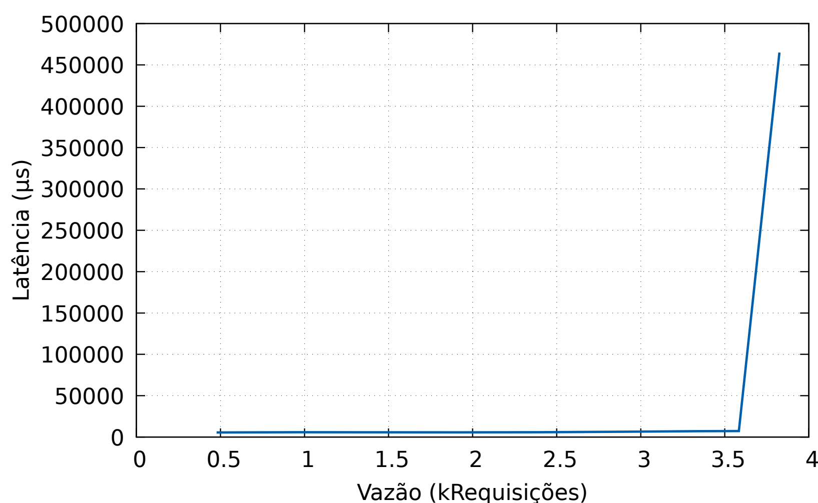
uma vez que com um maior número de partições, mais sincronizações acontecem em requisições multi-variáveis, e a carga YCSB-E é composta inteiramente por requisições multi-variáveis.

Os valores de vazão e latência por tempo são mostrados na Figura 29, e apresentam diferenças consideráveis às cargas de trabalho anteriores. A vazão, apresentada na Figura 29(a), segue o padrão de picos e vales. Devido aos padrões de acesso anteriores, os algoritmos de particionamento executavam sob grafos sem arestas, balanceando apenas vértices. Dessa vez, com a adição contínua de arestas causadas pelas partições multi-variáveis, a característica da redução do conjunto de arestas de corte passa a ter um papel central. Essa diferença parece favorecer consideravelmente algoritmos de *streaming*, que conseguem picos comparáveis aos algoritmos multiní-

Figura 28 – Gráfico representando a vazão (eixo x) pela latência (eixo y), observados para a carga YCSB-E.



(a) 4 partições



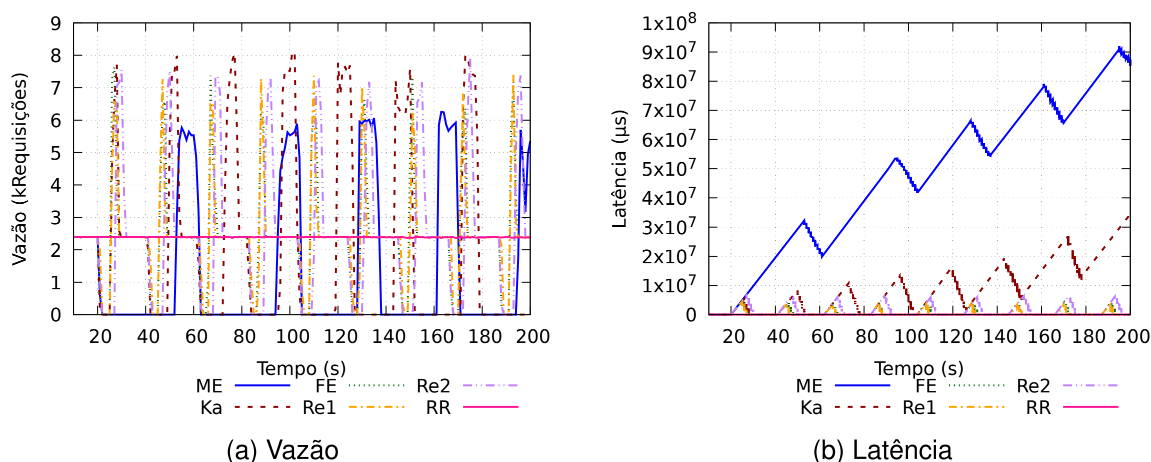
(b) 8 partições

Fonte: Elaborado pelo autor (2021).

vel, mas em um tempo expressivamente menor. Dos algoritmos multiníveis, o tempo de execução do METIS é bastante alto desde o primeiro reparticionamento, levando cerca de 20 segundos para concluí-lo. Esse tempo elevado de reparticionamento não é compensado por grandes picos. Enquanto picos de outros algoritmos chegam facilmente à 7.000 requisições processadas por segundo, os picos do METIS mal atingem 6.000, sugerindo um pior particionamento. O KaHIP começa com um tempo de execução comparável aos algoritmos de *streaming*, mas cresce conforme mais execuções são processadas, isso é, mais arestas são inseridas no grafo. Para os algoritmos de *streaming*, ReFENNEL (Re2) apresentou um tempo de execução levemente maior comparado aos demais, com valores máximos de vazão comparáveis. O desempenho do FENNEL e ReFENNEL (Re1) foram bastante similares, ambos atingindo altas vazões

com pouco tempo de execução.

Figura 29 – Gráfico com métricas do sistema durante execução, observados para a carga YCSB-E com 8 partições e Δ_p = 50.000.



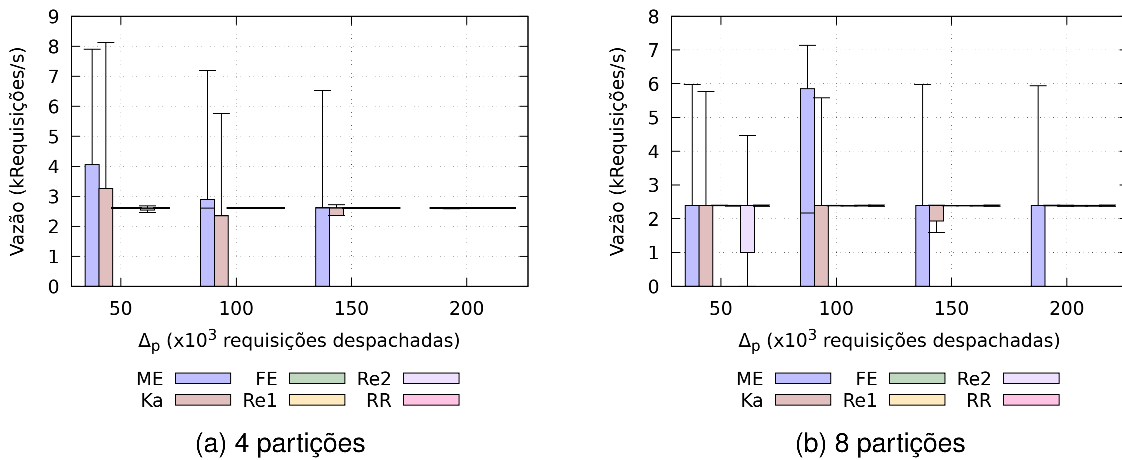
(a) Vazão (b) Latência

Fonte: Elaborado pelo autor (2021).

A latência mostrada na Figura 29(b) segue um padrão de escada. Esse padrão reflete a observação de que conforme uma maior quantidade de requisições é processada, isso é, vértices são adicionados ao grafo, maior é o tempo necessário para o reparticionamento de algoritmos multinível. METIS, o algoritmo mais lento, chega a atingir uma latência de mais de um minuto. O tempo de execução do KaHIP não cresce imediatamente como o apresentado pelo METIS, mas também tem um grande aumento com a adição de novas arestas, atingindo latências que ultrapassam 40 segundos. Os algoritmos de *streaming* apresentaram picos de latência de valores similares durante toda a execução, com uma mediana de aproximadamente 5.000 microssegundos, assim como o *Round-Robin*.

A Figura 30 mostra diagramas de caixa para a vazão apresentada pelo sistema na carga YCSB-E. Para algoritmos multinível, principalmente o METIS, a vazão possui uma grande variação. Isso é causado pelo grande tempo ocioso identificado anteriormente, causando uma grande quantidade de observações em ambos extremos dos valores observados para vazão. É possível ver situações onde o KaHIP também foi altamente desvantajoso, como na Figura 30(a) com Δ_p = 5.000. Apesar do KaHIP voltar mais rapidamente a ter uma vazão mais estável que o METIS, já em Δ_p = 100.000, a mediana da vazão do METIS é mais alta do que a mediana vista no KaHIP. Isso não necessariamente representa uma maior qualidade no reparticionamento. Uma vez que o algoritmo METIS apresenta uma vazão máxima menor, como visto na Figura29(a), o sistema demora mais para atingir o momento de reparticionamento. Isso permite com que ele apresente mais amostras de valores elevados de vazão, diferentemente do KaHIP e algoritmos de *streaming*, que atingem maiores picos de vazão mas por menos tempo, já que o próximo reparticionamento é iniciado mais rapidamente.

Figura 30 – Vazão observada com YCSB-E para diferentes intervalos de reparticionamento.

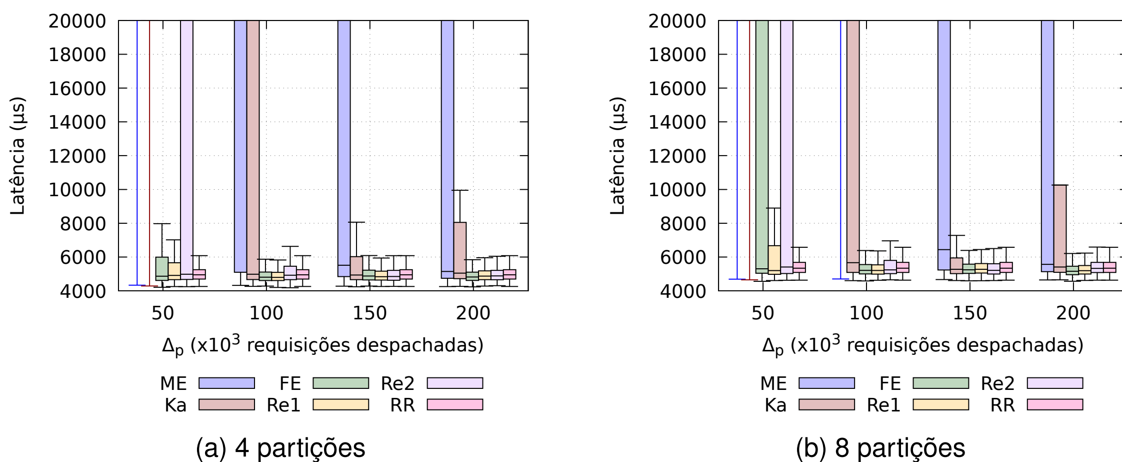


(a) 4 partições (b) 8 partições

Fonte: Elaborado pelo autor (2021).

A Figura 31 mostra o diagrama de caixas para a latência nos experimentos com a carga YCSB-E. O gráfico foi cortado para apresentar até a latência 20.000 microssegundos, uma vez que as altas latências apresentadas pelos algoritmos multinível, em especial pelo METIS, chegam à casa de minutos. A latência observada para algoritmos de *streaming* são comparáveis às latências observadas para o *Round-Robin*, chegando a valores ligeiramente menores em cenários com reparticionamento mais espaçado, como 8 partições e Δ_p = 200.000.

Figura 31 – Latência observada com YCSB-E para diferentes intervalos de reparticionamento.



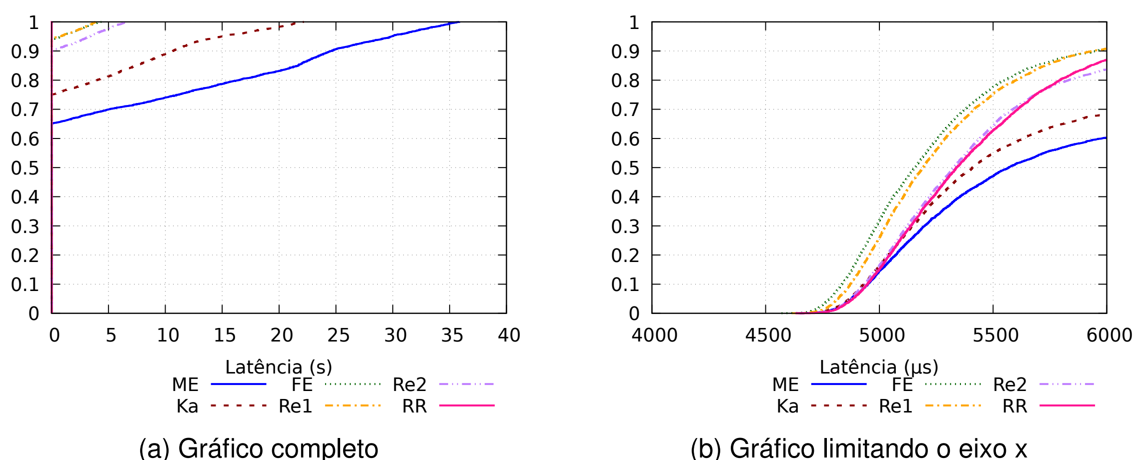
(a) 4 partições (b) 8 partições

Fonte: Elaborado pelo autor (2021).

O gráfico apresentado pela Figura 32 mostra a função de distribuição acumulada para YCSB-E com 8 partições, e Δ_p = 200.000. Ao ver o gráfico completo na Figura 32(a), é possível ver que 70% das requisições processadas pelo METIS foram respon-

didas dentro de 5 segundos. O KaHIP apresenta uma proporção levemente melhor, com 80% das requisições respondidas dentro de 5 segundos. Isso, porém, ainda é um valor extremamente alto. Os algoritmos de *streaming* têm por volta de 10% de suas requisições com o valor de aproximadamente 5 segundos. O gráfico na Figura 32(b) mostra o mesmo gráfico, mas com o eixo x limitado de 4.000 a 6.000 microssegundos. Essa aproximação permite perceber que as estratégias com os algoritmos de *streaming* FENNEL e ReFENNEL (Re1) responderam 80% em até 5.500 microssegundos. Para o *Round-Robin*, pouco mais de 60% das requisições foram respondidas em até 5.550 microssegundos. Isso mostra como existem ganhos na latência em uma quantia considerável de requisições, mas isso faz com que 10% das requisições atinjam a grande latência de quase 5 segundos. Usando o algoritmo ReFENNEL (Re2) quase não existem ganhos na latência em comparação ao *Round-Robin*, mas ainda assim apresenta o custo de latências acima de 5 segundos. Os algoritmos multinível apresentam o pior desempenho, apresentando latências consistentemente mais altas do que no *Round-Robin*.

Figura 32 – Gráfico de função de distribuição acumulada para latência de YCSB-E com 8 partições e Δ_p = 200.000.



(a) Gráfico completo (b) Gráfico limitando o eixo x

Fonte: Elaborado pelo autor (2021).

Os experimentos realizados com o protótipo distribuído mostram como o tempo de execução do algoritmo de particionamento tem um grande impacto na latência do sistema. Uma quantidade frequente de reparticionamento com algoritmos lentos resulta no tempo de resposta que pode atingir minutos, e uma vazão com variação extremamente alta. Quando o objetivo principal é balancear o número de acessos, o que representa cargas YCSB-A e YCSB-D, o uso de algoritmos de reparticionamento parece causar uma diminuição na vazão, ao mesmo tempo em que aumenta a latência. Esse impacto negativo, porém, fica consideravelmente menor com o aumento do valor de Δ_p. Isso vai de acordo com os experimentos realizados no protótipo local, onde particionamentos frequentes causaram um aumento no *makespan* do sistema.

O reparticionamento no protótipo distribuído não mostra um benefício na vazão ou latência quando comparados ao *Round-Robin*. Nesse cenário, os algoritmos multiníveis foram os que menos causaram variação de desempenho em relação ao *Round-Robin*, especialmente o algoritmo METIS, que manteve a latência entre intervalos pequenos mesmo na presença de reparticionamentos constantes. Dos algoritmos de *streaming*, o algoritmo ReFENNEL (Re2) mostrou o pior desempenho devido ao seu longo tempo de execução.

Durante a execução da carga YCSB-E, o foco do reparticionamento passa a ser o agrupamento de variáveis frequentemente acessadas em uma mesma partição, para reduzir a quantidade de sincronizações. Enquanto no protótipo local menores valores de Δ_p indicavam ganhos significativos na vazão, no protótipo distribuído reparticionamentos frequentes passam a apresentar piores resultados. O tempo de execução é um aspecto crucial que afeta o desempenho dos algoritmos de reparticionamento nessa carga. Algoritmos multiníveis, especialmente o METIS, passam a apresentar o pior desempenho, já que seu tempo de execução aumenta significativamente de acordo com a quantidade de arestas no grafo de trabalho, isso é, quantidade de requisições de múltiplas variáveis. Os algoritmos de *streaming*, por executarem muito mais rapidamente, não tem esse grande impacto negativo. Ambos FENNEL e ReFENNEL (Re1) apresentam cerca de 90% das requisições com uma latência ligeiramente menor quando comparada ao *Round-Robin*, mas em contrapartida causam latências de até 5 segundos em cerca de 5% das requisições. O algoritmo ReFENNEL (Re2) apresentou um grande ganho de desempenho no protótipo local, o que não se repetiu no protótipo distribuído. Apesar da boa qualidade do particionamento gerado por ele, o tempo de execução elevado em comparação aos demais algoritmos de *streaming* faz com que ele apresente latências maiores.

Ainda sobre a carga YCSB-E, em nenhum momento o sistema permaneceu estável, principalmente nos cenários com maior desempenho dos algoritmos de *streaming*. Todo pico de vazão foi imediatamente seguido de um vale causado pelo próximo reparticionamento. Uma vez que os algoritmos de particionamento FENNEL e ReFENNEL (Re1) resultaram em bons particionamentos e de maneira rápida, seria interessante ver como o sistema se comportaria caso conseguisse permanecer estável por alguns segundos após o reparticionamento, observar se a vazão permaneceria maior, ou se ela tenderia a voltar para a mesma vazão apresentada pelo *Round-Robin*, como ocorreu para as cargas YCSB-A e YCSB-D.

6 CONCLUSÃO

Algoritmos de particionamento balanceado de grafos são frequentemente usados com o propósito de oferecer melhor escalonamento em sistemas particionados. Apesar do ganho de popularidade, existe uma falta de entendimento em como eles se comparam uns aos outros, e qual seu desempenho ao serem usados para particionar o estado em réplicas em RMEP.

Nesse trabalho, buscou-se aprofundar os estudos de particionamento balanceado de grafos em uma réplica em RMEP. Foi definido um modelo de execução de RMEP que utiliza o particionamento de estado para extrair paralelismo, enquanto mantém o requisito de consistência. Depois, quatro algoritmos de particionamento balanceado de grafos foram avaliados usando o *benchmark* YCSB, e comparados com uma abordagem por *Round-Robin*. Cada algoritmo foi testado experimentalmente, mostrando-se como o número de partições e intervalos de reparticionamentos podem impactar o desempenho.

Apesar de reparticionamentos dinâmicos terem se mostrado efetivos, não existe uma "bala de prata". Nossos resultados sugerem um intervalo de reparticionamento espaçado para cargas de trabalho dominadas por requisições de variável única, e um intervalo curto para cargas de trabalho dominadas por requisições multi-variáveis. Essas descobertas são úteis para entender os prós e contras de cada algoritmo e podem dar apoio a outras estratégias, como políticas de escalonamento adaptativas.

6.1 TRABALHOS FUTUROS

Como próximos passos para avançar a pesquisa, serão estudadas diferentes configurações para o protótipo distribuído que podem permitir análises melhores do impacto do reparticionamento. Valores de 4kB se mostraram adequados para o protótipo local, mas podem aumentar consideravelmente o custo de rede e ordenação no protótipo distribuído. Com um tempo maior, também serão analisados cenários em que o sistema executa por mais tempo. Isso permitirá explorar valores maiores de Δ_p, que poderão permitir um melhor desempenho no uso dos algoritmos de particionamento, em especial para a carga YCSB-E, que mesmo com Δ_p = 200.000 permaneceu constantemente em reparticionamento.

O protótipo local sugeriu um leve ganho de desempenho com Δ_p alto, o que não foi visto no protótipo distribuído. No protótipo distribuído as requisições foram enfileiradas durante o particionamento, e uma vez que o sistema não estava saturado, ele foi capaz de executar o excedente e ter sua vazão novamente equiparável à estratégia *Round-Robin*. É então estudada a possibilidade de analisar o sistema mais próximo do seu ponto de saturação, simulando um sistema de execução em lotes, ao contrário do sistema atual, que continuamente responde a requisições individuais, atendendo

uma carga de aproximadamente 70% da capacidade máxima de processamento.

Além disso, também foi elaborado um simulador para buscar entender melhor o comportamento visto durante a execução do modelo. No simulador, todos os custos de tempo real de execução são abstraídos, e é possível extrair informações mais detalhadas sobre a execução. Apesar de ter seu desenvolvimento concluído, as limitações de tempo fizeram com que fossem priorizados os estudos no protótipo, o que fez com que os experimentos no simulador não avançassem o suficiente para incluí-los no trabalho. A intenção é de que os estudos utilizando o simulador possam ser aprimorados, para explicar comportamentos percebidos no protótipo, como os sucessivos formatos de bacia apresentados na Figura 15.

Finalmente, é estudada a possibilidade de propor um algoritmo de *streaming* próprio, e adicionar a avaliação do seu desempenho junto aos comparativos dos algoritmos já estudados.

REFERÊNCIAS

ABBAS, Zainab; KALAVRI, Vasiliki; CARBONE, Paris; VLASSOV, Vladimir. Streaming graph partitioning: an experimental study. **Proceedings of the VLDB Endowment**, VLDB Endowment, v. 11, n. 11, p. 1590–1603, 2018.

ADONI, Hamilton Wilfried Yves; NAHHAL, Tarik; KRICHEN, Moez; AGHEZZAF, Brahim; ELBYED, Abdeltif. A survey of current challenges in partitioning and processing of graph-structured data in parallel and distributed systems. **Distributed and Parallel Databases**, Springer, p. 1–36, 2019.

ALCHIERI, Eduardo; DOTTI, Fernando; MENDIZABAL, Odorico M; PEDONE, Fernando. Reconfiguring parallel state machine replication. *In*: IEEE. 2017 IEEE 36th Symposium on Reliable Distributed Systems (SRDS). [*S.l.*: *s.n.*], 2017. P. 104–113.

ALCHIERI, Eduardo; DOTTI, Fernando; PEDONE, Fernando. Early scheduling in parallel state machine replication. *In*: PROCEEDINGS of the ACM Symposium on Cloud Computing. [*S.l.*: *s.n.*], 2018. P. 82–94.

AMER-YAHIA, Sihem; MARKL, Volker; HALEVY, Alon; DOAN, AnHai; ALONSO, Gustavo; KOSSMANN, Donald; WEIKUM, Gerhard. Databases and Web 2.0 panel at VLDB 2007. **ACM SIGMOD Record**, ACM New York, NY, USA, v. 37, n. 1, p. 49–52, 2008.

BATTITI, Roberto; BERTOSSI, Alan A. Greedy, prohibition, and reactive heuristics for graph partitioning. **IEEE transactions on computers**, IEEE, v. 48, n. 4, p. 361–385, 1999.

COELHO, P.; PEDONE, F. Geographic State Machine Replication. *In*: 2018 IEEE 37th Symposium on Reliable Distributed Systems (SRDS). [*S.l.*: *s.n.*], 2018. P. 221–230.

COOPER, Brian F; SILBERSTEIN, Adam; TAM, Erwin; RAMAKRISHNAN, Raghu; SEARS, Russell. Benchmarking cloud serving systems with YCSB. *In*: PROCEEDINGS of the 1st ACM symposium on Cloud computing. [*S.l.*: *s.n.*], 2010. P. 143–154.

CURINO, Carlo; JONES, Evan; ZHANG, Yang; MADDEN, Sam. Schism: a workload-driven approach to database replication and partitioning. **Proceedings of the VLDB Endowment**, VLDB Endowment, v. 3, n. 1-2, p. 48–57, 2010.

DAS, Sudipto; AGRAWAL, Divyakant; EL ABBADI, Amr. G-store: a scalable data store for transactional multi key access in the cloud. *In*: PROCEEDINGS of the 1st ACM symposium on Cloud computing. [*S.l.*: *s.n.*], 2010. P. 163–174.

FIDUCCIA, Charles M; MATTHEYSES, Robert M. A linear-time heuristic for improving network partitions. *In*: IEEE. 19TH design automation conference. [*S.l.*: *s.n.*], 1982. P. 175–181.

GAREY, Michael R; JOHNSON, David S; STOCKMEYER, Larry. Some simplified NP-complete problems. *In*: PROCEEDINGS of the sixth annual ACM symposium on Theory of computing. [*S.l.*: *s.n.*], 1974. P. 47–63.

GUO, Yong; HONG, Sungpack; CHAFI, Hassan; IOSUP, Alexandru; EPEMA, Dick. Modeling, analysis, and experimental comparison of streaming graph-partitioning policies. **Journal of Parallel and Distributed Computing**, Elsevier, v. 108, p. 106–121, 2017.

HENDRICKSON, Bruce; LELAND, Robert W. A Multi-Level Algorithm For Partitioning Graphs. **SC**, v. 95, n. 28, p. 1–14, 1995.

HERLIHY, Maurice P; WING, Jeannette M. Linearizability: A correctness condition for concurrent objects. **ACM Transactions on Programming Languages and Systems (TOPLAS)**, ACM New York, NY, USA, v. 12, n. 3, p. 463–492, 1990.

JAIN, Sachin; SWAMY, Chaitanya; BALAJI, K. Greedy algorithms for k-way graph partitioning. *In*: CITESEER. THE 6th international conference on advanced computing. [*S.l.*: *s.n.*], 1998. P. 100.

KAPRITSOS, Manos; WANG, Yang; QUEMA, Vivien; CLEMENT, Allen; ALVISI, Lorenzo; DAHLIN, Mike. All about Eve: execute-verify replication for multi-core servers. *In*: PRESENTED as part of the 10th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 12). [*S.l.*: *s.n.*], 2012. P. 237–250.

KARYPIS, George; KUMAR, Vipin. A fast and high quality multilevel scheme for partitioning irregular graphs. **SIAM Journal on scientific Computing**, SIAM, v. 20, n. 1, p. 359–392, 1998a.

KARYPIS, George; KUMAR, Vipin. A hypergraph partitioning package. **Army HPC Research Center, Department of Computer Science & Engineering, University of Minnesota**, 1998b.

KOTLA, Ramakrishna; DAHLIN, Michael. High throughput Byzantine fault tolerance. *In*: IEEE. INTERNATIONAL Conference on Dependable Systems and Networks, 2004. [*S.l.*: *s.n.*], 2004. P. 575–584.

KUMAR, K Ashwin; DESHPANDE, Amol; KHULLER, Samir. Data placement and replica selection for improving co-location in distributed environments. **arXiv preprint arXiv:1302.4168**, 2013.

LAMPORT, Leslie. Time, clocks, and the ordering of events in a distributed system. **Communications of the ACM**, ACM, v. 21, n. 7, p. 558–565, 1978.

LE, Long Hoang; FYNN, Enrique; ESLAHI-KELORAZI, Mojtaba; SOULÉ, Robert; PEDONE, Fernando. DynaStar: Optimized Dynamic Partitioning for Scalable State Machine Replication. *In*: IEEE. 2019 IEEE 39th International Conference on Distributed Computing Systems (ICDCS). [*S.l.*: *s.n.*], 2019. P. 1453–1465.

LI, Bijun; XU, Wenbo; ABID, Muhammad Zeeshan; DISTLER, Tobias; KAPITZA, Rüdiger. Sarek: Optimistic parallel ordering in byzantine fault tolerance. *In*: IEEE. 2016 12th European Dependable Computing Conference (EDCC). [*S.l.*: *s.n.*], 2016. P. 77–88.

LI, Bijun; XU, Wenbo; KAPITZA, Rüdiger. Dynamic State Partitioning in Parallelized Byzantine Fault Tolerance. *In*: IEEE. 2018 48th Annual IEEE/IFIP International Conference on Dependable Systems and Networks Workshops (DSN-W). [*S.l.*: *s.n.*], 2018. P. 158–163.

MARANDI, Parisa Jalili; BEZERRA, Carlos Eduardo; PEDONE, Fernando. Rethinking state-machine replication for parallelism. *In*: IEEE. 2014 IEEE 34th International Conference on Distributed Computing Systems. [*S.l.*: *s.n.*], 2014. P. 368–377.

MENDIZABAL, Odorico M; DE MOURA, Rudá ST; DOTTI, Fernando Luıs; PEDONE, Fernando. Efficient and deterministic scheduling for parallel state machine replication. *In*: IEEE. 2017 IEEE International Parallel and Distributed Processing Symposium (IPDPS). [*S.l.*: *s.n.*], 2017. P. 748–757.

NISHIMURA, Joel; UGANDER, Johan. Restreaming graph partitioning: simple versatile algorithms for advanced balancing. *In*: PROCEEDINGS of the 19th ACM SIGKDD international conference on Knowledge discovery and data mining. [*S.l.*: *s.n.*], 2013. P. 1106–1114.

PACACI, Anil; ÖZSU, M Tamer. Experimental Analysis of Streaming Algorithms for Graph Partitioning. *In*: PROCEEDINGS of the 2019 International Conference on Management of Data. [*S.l.*: *s.n.*], 2019. P. 1375–1392.

QUAMAR, Abdul; KUMAR, K Ashwin; DESHPANDE, Amol. SWORD: scalable workload-aware data placement for transactional workloads. *In*: PROCEEDINGS of the 16th International Conference on Extending Database Technology. [*S.l.*: *s.n.*], 2013. P. 430–441.

SAKOUHI, Chayma; KHALDI, Abir; GHEZAL, Henda Ben. An overview of recent graph partitioning algorithms. *In*: THE STEERING COMMITTEE OF THE WORLD CONGRESS IN COMPUTER SCIENCE, COMPUTER ... PROCEEDINGS of the International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA). [*S.l.*: *s.n.*], 2018. P. 408–414.

SANDERS, Peter; SCHULZ, Christian. Think locally, act globally: Highly balanced graph partitioning. *In*: SPRINGER. INTERNATIONAL Symposium on Experimental Algorithms. [*S.l.*: *s.n.*], 2013. P. 164–175.

SCHNEIDER, Fred B. Implementing fault-tolerant services using the state machine approach: A tutorial. **ACM Computing Surveys (CSUR)**, ACM, v. 22, n. 4, p. 299–319, 1990.

SCIASCIA, Daniele. **Libpaxos 3**. [*S.l.*: *s.n.*], 2013. http://libpaxos.sourceforge.net/paxos_projects.php. Accessed: 2021-04-10.

TSOURAKAKIS, Charalampos; GKANTSIDIS, Christos; RADUNOVIC, Bozidar; VOJNOVIC, Milan. Fennel: Streaming graph partitioning for massive scale graphs. *In*:

PROCEEDINGS of the 7th ACM international conference on Web search and data mining. [*S.l.*: *s.n.*], 2014. P. 333–342.

WHITE, Brian; LEPREAU, Jay; STOLLER, Leigh; RICCI, Robert; GURUPRASAD, Shashi; NEWBOLD, Mac; HIBLER, Mike; BARB, Chad; JOGLEKAR, Abhijeet. An Integrated Experimental Environment for Distributed Systems and Networks. *In*: "USENIX Association". "PROC. OF the Fifth Symposium on Operating Systems Design and Implementation". Boston, MA: [*s.n.*], dez. 2002. P. 255–270.

Apêndices

APÊNDICE A – CÓDIGO DESENVOLVIDO

Todo código desenvolvido está armazenado no repositório aberto *Github*. O código desenvolvido para esse trabalho está dividido em dois repositórios, um para o simulador, e outro para os protótipos local e distribuído. Ambos um arquivo "README.txt"na raiz do projeto, com instruções para compilação e execução do código.

A.1 SIMULADOR

Como mencionado nas conclusões, foi desenvolvido um simulador para testar a qualidade do particionamento de cada algoritmo em um cenário ideal, que acabou não sendo incluído no estudo devido a limitações de tempo. Apesar disso, o programa foi essencial para as primeiras analises, assim como para elencar os algoritmos que seriam comparados no trabalho. O repositório pode ser acessado através do seguinte *link*: https://github.com/gabrieltron/psmr-simulator

O intuito do simulador é comparar não apenas diferentes algoritmos de particionamento balanceado de grafos, mas também diferentes políticas de escalonamento em Replicação Máquina de Estados Paralela. Seu desenvolvimento modular procura oferecer uma interface que permita a implementação de diferentes políticas de escalonamento com facilidade, em um ambiente pronto para realizar análises de desempenho.

O simulador é dividido em 7 módulos, são eles:

- **Entrada** - Lê arquivos de configuração que customizam a execução.
- **Simulador** - Classe principal que orquestra a interação entre as demais. Simula a execução das requisições.
- **Random** - Módulo com geração de diferentes padrões de aleatoriedade para gerar requisições.
- **Gerador de requisições** - Gera requisições que serão usadas na simulação.
- **Gerenciador** - Módulo que implementa a política de escalonamento a ser avaliada.
- **Log** - Mantém um log da execução para avaliação posterior.
- **Saída** - Mostra dados da execução, como tempo total, taxa de utilização das *threads*, quantidade de sincronizações necessárias, dentre outras informações.

Mais detalhes podem ser encontrados no arquivo "README.txt"na raíz do projeto. Ao final do desenvolvimento desse trabalho, a *branch* principal encontrava-se no *commit* de *hash* 6d6e2daf4fd95a5fae66fcae1374095ef7b0d075.

A.2 PROTÓTIPOS

Os protótipos, tanto local quanto distribuído, são os programas analisados com mais profundidade durante a avaliação experimental. O repositório contendo o código pode ser acessado através do seguinte link: https://github.com/gabrieltron/kvpaxos.

Ambos protótipos foram divididos nos seguintes módulos:

- **Compressor** - Responsável com comprimir e descomprimir os valores armazenados durante a escrita e leitura, respectivamente.
- **Contantes** - Módulo contendo constantes que são utilizadas em vários locais do programa.
- **Grafo** - Módulo responsável por gerenciar a estrutura e também implementar as operações realizadas no Grafo.
- **Escalonador** - Módulo que implementa o escalonador de requisições, que envia as requisições para as *threads* trabalhadoras, gerencia as sincronizações necessárias, mantém registro da quantidade de requisições despachadas para iniciar o reparticionamento. A implementação das *threads* trabalhados também se encontra nesse módulo.
- **Armazenamento** - Implementação do sistema chave-valor utilizando um *hashmap* concorrente.
- **Tipos** - Módulo contendo a definição de tipos que são necessários para a construção de mensagens. Esse módulo é feito principalmente para re-declarar tipos utilizados pela biblioteca LibPaxos, que são necessários para o programa, mas suas definições não são acessíveis em tempo de *linkagem* devido ao sistema de compilação da biblioteca.
- **Réplica** - Classe que implementa o recebimento de requisições.

  Em adição a esses módulos, o protótipo distribuído conta com os seguintes módulos, não presentes no protótipo local:

- **Evclient** - Implementa um cliente que estabiliza a conexão do cliente com as réplicas que compõe o sistema de Replicação Máquina de Estados Paralela. A intenção é ter um paralelo ao *Evreplica*, classe responsável pela conexão das réplicas, já presente na biblioteca LibPaxos.
- **Cliente** - Classe que implementa o envio de requisições para as réplicas.

  Cada versão do protótipo encontra-se em uma *branch* diferente. Enquanto o protótipo local pode ser encontrado na *branch simplified-replica*, o protótipo distribuído está na *branch master*. Mais detalhes sobre a implementação, como

dependências do projeto, como compilar e executar, podem ser encontrados nos arquivos "README.txt"presentes na raiz de cada *branch*. Ao final do desenvolvimento desse trabalho, a *branch master* encontrava-se no *commit* de *hash* f386821ef37956ef3f7d8d976a529dce302bd2ff, e a *branch simplified-replica* no *commit* de *hash* 7352090ca9b3d3d05a431a89b5c7c4a9c6b81b3a.

APÊNDICE B – ARTIGOS

O desenvolvimento dessa pequisa resultou na publicação de dois trabalhos. A primeira publicação aconteceu na décima edição do evento *Computer on the Beach*, e tem como foco o modelo de execução por particionamento de estado discutido nesse trabalho. A segunda publicação foi feita na vigésima edição do evento **Escola Regional de Alto Desempenho da Região Sul (ERAD/RS)**, e tem como foco a discussão da necessidade de algoritmos de escalonamento em RMEP, e o simulador previamente mencionado.

B.1 XI COMPUTER ON THE BEACH

Proposta para Reparticionamento de Estado em Replicação Máquina de Estado Paralela

João Gabriel Trombeta[1], **Odorico Machado Mendizabal**[1]

[1]Departamento de Informática e Estatística – Universidade Federal de Santa Catarina
Caixa Postal 476 – 88040-900 – Florianópolis – SC – Brasil

joao.gabriel.trombeta@grad.ufsc.br, odorico.mendizabal@ufsc.br

Abstract. *State Machine Replication is a widely used technique to provide fault tolerance and strong consistency. In this approach, all commands are executed sequentially throughout the replicas. Aiming to improve the system's throughput, enhanced versions were proposed, where independent commands can be executed in parallel. One arising challenge, though, is how to balance the workload between threads, while avoiding the need of synchronization between them. This extended abstract proposes a technique to schedule requests in a dynamic and efficient manner, using a workload graph to keep track of access patterns and graph partitioning to decompose and dispatch workload to the worker threads.*

Resumo. *Replicação Máquina de Estados é uma técnica amplamente utilizada para prover tolerância a falhas e consistência forte. Nela, todos os comandos são executados sequencialmente pelas réplicas. Buscando melhorar a vazão do sistema, versões aprimoradas foram propostas, onde comandos independentes podem ser executados em paralelo. Um crescente desafio, porém, é como balancear a carga de trabalho entre threads, e também evitar a necessidade de sincronização entre elas. Esse resumo estendido propõe uma técnica para escalonar requisições em uma maneira dinâmica e eficiente, usando um grafo de trabalho para manter registro de padrões de acesso e particionamento de grafos para decompor e despachar a carga de trabalho para threads trabalhadoras.*

1. Introdução

Diversos serviços na Internet possuem rigorosos requisitos de disponibilidade. Nesse contexto, a Replicação Máquina de Estado (RME) [Schneider 1990, Lamport 1978] é uma técnica bem estabelecida que provê tolerância a falhas e garantia de consistência forte (linearizabilidade). Em RME, réplicas do sistema inciam no mesmo estado inicial e executam os mesmos comandos na mesma ordem, de maneira determinística, fazendo com que todas atinjam os mesmos estados durante a execução.

A implementação clássica de RME possui a limitação de que comandos precisam ser executados de maneira sequencial, não tomando proveito de arquiteturas multiprocessadores. Replicação Máquina de Estado Paralela [Kotla and Dahlin 2004, Kapritsos et al. 2012, Marandi et al. 2014, Mendizabal et al. 2017] foi então desenvolvida, onde comandos não conflitantes podem ser executados simultaneamente, aumentando a vazão de requisições processadas sem violação de consistência. Dois comandos

são ditos não conflitantes caso a escrita de um não ocorra no mesmo dado em que a leitura ou escrita do outro, caso contrário são conflitantes. Para maximizar o ganho de desempenho, faz-se necessário explorar o balanceamento do trabalho entre as threads, assim como minimizar o impacto causado pela necessidade de sincronização que se dá pela existência de comandos conflitantes [Alchieri et al. 2017].

O balanceamento de carga em uma réplica individual deve levar em consideração a distribuição do acesso aos dados, que pode sofrer alterações com o tempo, e evitar sincronização entre threads, que degrada a performance do sistema. Trabalhos recentes estudam problemas de otimização semelhantes em contextos distribuídos (e.g., [Taft et al. 2014, Le et al. , Curino et al. 2010]). Resultados apresentados em [Curino et al. 2010, Le et al.] indicam que o uso de grafos de dependência auxilia no reparticionamento do estado entre as réplicas distribuídas. Espera-se que o uso dessa técnica possa ser aplicado localmente em réplicas em RME Paralela, visando reduzir sincronizações e balancear a carga entre threads.

Nesse resumo é proposta uma técnica em que a execução de requisições referentes a cada dado é delegada a uma única thread, de maneira dinâmica e completamente transparente. Durante a execução, de acordo com o número de requisições feitas a um dado e os conflitos por elas causados, a associação entre dados e threads pode ser reestruturada, visando um melhor equilíbrio de trabalho e, consequentemente, aumento na vazão do sistema.

2. Solução Proposta

Seguindo o modelo de Replicação Máquina de Estado Paralela, assume-se que cada réplica possui *k* threads trabalhadoras. Além disso, o sistema possui uma estrutura que mapeia um dado da aplicação para a thread responsável por executar as requisições que o acessam, assim como um grafo, que é utilizado para recalcular a associação entre os dados e threads buscando rebalanceamento.

2.1. Grafo de trabalho

A réplica armazena, de maneira global, um grafo não direcionado. Cada vértice (v, w) é um elemento do conjunto de vértices V, onde o dado v da aplicação foi acessado w vezes. Cada aresta pode ser descrita como (x, y, p), onde os vértices x e y estão conectados se e somente se x e y executaram comandos conflitantes p vezes, sendo $p \geq 1$. Essa estrutura foi escolhida para que, durante a fase de reparticionamento, algoritmos conhecidos em grafos possam ser usados para otimizar o reparticionamento.

Uma implementação de grafo eficiente é fundamental para que a técnica apresente um bom desempenho, uma vez que ele é acessado frequentemente e por múltiplas threads. Estruturas de dados concorrentes vêm sendo tema de estudos recentes, incluindo grafos, idealmente a estrutura deve ser livre de espera e ter tempo de acesso à vértices e arestas constante. Algumas implementações sendo estudadas foram propostas em [Chatterjee et al. 2019, Kallimanis and Kanellou 2016, Escobar et al. 2019].

2.2. Execução de requisições

Uma requisição r é acessada pela thread associada aos dados manipulados por r, seguido por uma atualização no grafo de trabalho para registrar a ocorrência do acesso e, possivelmente, conflito.

Caso requisições acessem um único dado, elas podem ser executadas de maneira paralela. Uma sincronização é necessária quando uma requisição acessa múltiplos dados, sendo pelo menos dois desses mapeados para threads diferentes. Para demonstrar a razão, suponha como exemplo uma thread t_1, responsável pelas requisições referentes ao dado a, e t_2, responsável pelas requisições em b. Suponha também o comando $write(x, y)$, que escreve o valor y em x, e $swap(w, z)$, que troca os valores de w e z. Ao receber a sequência de comandos $write(a, 2)$, $write(b, 3)$ e $swap(a, b)$, é necessário que o comando $swap(a, b)$ seja executado após t_1 e t_2 processarem suas requisições de escrita, exigindo sincronização entre t_1 e t_2 para garantir consistência. Na ocorrência de uma situação como a demonstrada cria-se uma barreira.

Para que seja feito o processamento da requisição que requer sincronização, é necessário que todas as threads envolvidas tenham processado todas as requisições que antecedem a barreira. Note que neste caso, algumas threads ficam ociosas, aguardando pela execução dessas requisições. Quando todas as threads estiverem prontas, apenas uma executa o comando e atualiza o grafo de trabalho. Em seguida, todas as threads envolvidas na sincronização podem prosseguir.

A consistência garantida pela sincronização entre threads é trivialmente atingida caso todos os dados acessados estejam associados a uma única thread, uma vez que existe a garantia de que todas as requisições envolvendo os múltiplos dados serão executadas de maneira sequencial.

A Fig. 1 mostra um exemplo de estado do sistema, onde a thread $t1$ é responsável por executar requisições sobre os dados x e y, enquanto $t2$ é responsável por operações sobre w e z. É possível observar que r_1, r_2, r_4 e r_5 acessam dados mapeados a uma única partição. A requisição r_3, entretanto, acessa dados das partições de t_1 e t_2, sendo adicionada nas filas de ambas as threads.

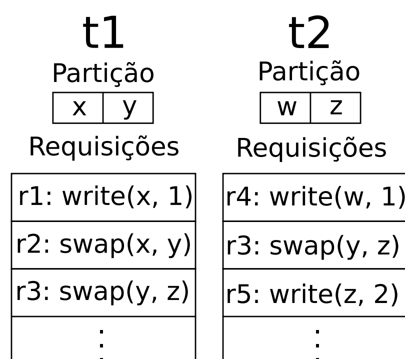


Figure 1. Exemplo de estado do sistema.

As requisições $r1$ e $r4$ podem ser executadas em paralelo, como demonstrado na Fig. 2. $t1$ pode executar $r2$, pois ambos os dados estão em sua partição, mas $t2$ não pode executar $r3$, pois precisa de um dado que está em $t1$. $t2$ então sinaliza para $t1$ que está aguardando, e quando chega a vez de $t1$ executar $r3$ ela a faz, uma vez que já recebeu o aviso de que $t2$ está aguardando. $t1$ executa $r3$ e comunica $t2$, que pode continuar a executar comandos em sua fila. Cada thread, após executar uma requisição, atualiza o grafo de trabalho.

Decorrente do modelo de execução descrito, não existe concorrência ao acessar

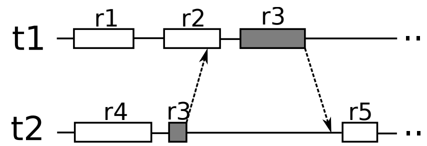


Figure 2. Execução das requisições representadas na Fig. 1.

um dado, removendo a necessidade de travas no processamento de requisições e tornando-as livres de espera. Requisições que envolvem uma única variável serão executadas apenas pela única thread a que está designada, e requisições que envolvem múltiplas variáveis serão executadas por uma única thread, caso as variáveis estejam todas mapeadas a uma mesma thread, ou todas as threads envolvidas sincronizarão de forma que apenas uma execute a requisição.

O reparticionamento é iniciado depois de disparado um certo gatilho, nele os dados adquiridos durante a execução são utilizados para criar um novo mapeamento que melhor equilibra a carga de execução das threads. Para evitar que durante a transição para o novo mapeamento duas threads executem requisições em um mesmo dado, é feita uma barreira para esperar que todas as threads finalizem requisições pendentes, depois disso a barreira é liberada e o novo mapeanto entra em vigor.

2.3. Reparticionamento dos dados

É necessário diminuir a quantidade de sincronizações entre threads, uma vez que causam ociosidade até que todas as threads atinjam a barreira. Não existindo a necessidade de sincronização, menos threads permanecerão ociosas, acarretando em um melhor aproveitamento dos recursos do sistema. Logo, é ideal que dados frequentemente acessados em conjunto sejam delegados à mesma thread.

A quantidade de sincronizações durante a execução de requisições poderia ser trivialmente resolvida fazendo com que todas as requisições sejam executadas por uma única thread, o que seria o mesmo que uma implementação de RME tradicional, onde o processamento ocorre de maneira puramente sequencial e recursos da máquina são subutilizados. Existe a necessidade de reduzir a quantidade de comandos que requerem sincronização, mas, simultaneamente, é preciso distribuir a execução de forma a não sobrecarregar algumas threads enquanto outras são subutilizadas.

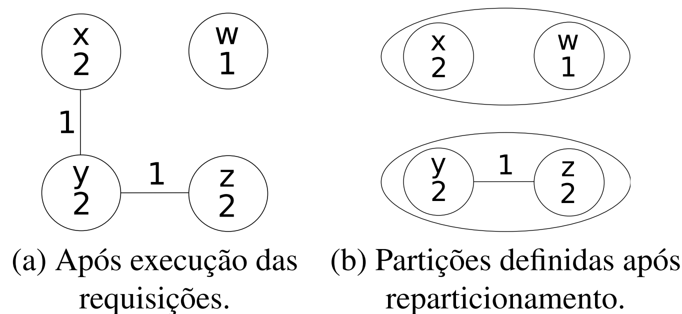
Outro fator que pode desbalancear o trabalho entre threads são aspectos da semântica da aplicação. Em redes sociais, por exemplo, um vídeo que repentinamente tornou-se viral terá um grande pico em seu acesso, sobrecarregando a thread responsável por seus dados. É necessário que o reparticionamento seja feito de maneira dinâmica, para levar em consideração as mudanças causadas por aspectos inerentes à aplicação que não podem ser previstos de maneira estática.

O reparticionamento é, portanto, um problema de otimização, onde os dados devem ser distribuídos de maneira a diminuir a necessidade de sincronizações e, ao mesmo tempo, balancear o trabalho entre threads de maneira homogênea. Tendo em vista que se quer particionar os dados em k threads com a restrição do balanceamento de carga, é possível reduzir esse problema para o problema de particionamento de um grafo em k partições com restrições, onde a restrição é o balanceamento de carga entre as partições.

Especificamente, o problema pode ser descrito como o corte mínimo em k partições, que consiste em encontrar uma forma de cortar um grafo em k partições de modo que a soma dos pesos das arestas cortadas (que atravessam partições) seja o menor possível, com a adição da restrição que a soma dos pesos dos vértices em cada partição seja similar.

O grafo de trabalho é modelado de forma que seja possível usá-lo como entrada em um algoritmo de corte mínimo. O corte será feito preferencialmente em arestas de menor peso, separando dados que conflitaram poucas vezes, e manterá arestas de alto peso, isso é, permanecerão conectados na mesma partição dados que conflitaram muitas vezes. Além disso, a soma dos valores dos vértices em cada componente do grafo devem ser similares, como o valor de um vértice é a quantidade de vezes que ele foi acessado, a quantidade de acesso aos dados em cada partição serão similares.

Depois de aplicado o algoritmo de particionamento no grafo de trabalho, os dados são redistribuídos entre as threads. Ao final do reparticionamento, cada vértice será designado a uma única partição, e cada partição será designada a uma única thread. A Fig. 3(b) mostra um possível resultado do reparticionamento do grafo representado na Fig. 3(a). Nesse exemplo o grafo foi cortado em duas partições, uma de tamanho 3 e outra de tamanho 4, removendo uma aresta de peso 1 no processo. Após o reparticionamento, uma partição composta por x e w é designada a uma das threads, e uma composta por y e z à outra. O corte mínimo admite mais de uma solução, o grafo poderia ser cortado de outras maneiras que resultariam em partições de mesmo tamanho e com a soma dos pesos das arestas removidas com o mesmo valor.



(a) Após execução das requisições. (b) Partições definidas após reparticionamento.

Figure 3. Exemplo de grafo de trabalho.

O reparticionamento representa um aspecto crítico da execução, é necessário que ele não interfira no processamento de requisições e que consiga reagir a mudanças na carga de trabalho em tempo hábil. Técnicas eficientes de particionamento são bastante pesquisadas por sua utilidade em diversas áreas, elas podem ser divididas em algoritmos de particionamento estático, em fluxo, e para bancos de dados de grafos distribuídos [Sakouhi et al. 2018]. Algoritmos para o particionamento de grafos em fluxo são uma possibilidade a ser considerada, por possuírem menor complexidade e menor consumo de memória, e ainda assim produzirem boas partições. Além disso, algoritmos online [Stanton and Kliot 2012] também são promissores, por adaptar as partições dinamicamente de acordo com as mudanças feitas no grafo, sem a necessidade de recalcular as partições por inteiro.

3. Considerações finais

Por permitir certo grau de paralelismo, Replicação Máquina de Estado Paralela trás grandes ganhos de desempenho em comparação a implementações tradicionais de RME, porém o sucesso desta técnica depende da carga de trabalho das aplicações e das estratégias de balanceamento de carga. Aqui é proposta uma técnica onde dados são divididos em partições e cada thread é responsável por uma partição, para que requisições sejam distribuídas entre as threads de forma a balancear o trabalho e reduzir a necessidade de sincronizações.

Um aspecto da proposta que ainda deve ser explorado é o gatilho que dará início ao reparticionamento, algumas opções são ser disparado por um cronômetro ou ser disparado após um desbalanceamento acima do aceitável ser detectado.

É esperado que ao implementar o método de reparticionamento aqui proposto o sistema experiencie um aumento na vazão de requisições, mesmo que os padrões de acesso a dados da aplicação sofram alterações durante sua execução.

References

[Alchieri et al. 2017] Alchieri, E., Dotti, F., Mendizabal, O. M., and Pedone, F. (2017). Reconfiguring parallel state machine replication. In *2017 IEEE 36th Symposium on Reliable Distributed Systems (SRDS)*, pages 104–113. IEEE.

[Chatterjee et al. 2019] Chatterjee, B., Peri, S., Sa, M., and Singhal, N. (2019). A simple and practical concurrent non-blocking unbounded graph with linearizable reachability queries. In *Proceedings of the 20th International Conference on Distributed Computing and Networking*, pages 168–177.

[Curino et al. 2010] Curino, C., Jones, E., Zhang, Y., and Madden, S. (2010). Schism: a workload-driven approach to database replication and partitioning. *Proceedings of the VLDB Endowment*, 3(1-2):48–57.

[Escobar et al. 2019] Escobar, I. A., Alchieri, E., Dotti, F. L., and Pedone, F. (2019). Boosting concurrency in parallel state machine replication. In *Proceedings of the 20th International Middleware Conference*, pages 228–240.

[Kallimanis and Kanellou 2016] Kallimanis, N. D. and Kanellou, E. (2016). Wait-free concurrent graph objects with dynamic traversals. In *19th International Conference on Principles of Distributed Systems (OPODIS 2015)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik.

[Kapritsos et al. 2012] Kapritsos, M., Wang, Y., Quema, V., Clement, A., Alvisi, L., and Dahlin, M. (2012). {All about Eve}: execute-verify replication for multi-core servers. In *Proceedings of the 10th Symposium on Operating Systems Design and Implementation*, pages 237–250.

[Kotla and Dahlin 2004] Kotla, R. and Dahlin, M. (2004). High throughput byzantine fault tolerance. In *International Conference on Dependable Systems and Networks, 2004*, pages 575–584. IEEE.

[Lamport 1978] Lamport, L. (1978). Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565.

[Le et al.] Le, L. H., Fynn, E., Eslahi-Kelorazi, M., Soulé, R., and Pedone, F. Dynastar: Optimized dynamic partitioning for scalable state machine replication.

[Marandi et al. 2014] Marandi, P. J., Bezerra, C. E., and Pedone, F. (2014). Rethinking state-machine replication for parallelism. In *2014 IEEE 34th International Conference on Distributed Computing Systems*, pages 368–377. IEEE.

[Mendizabal et al. 2017] Mendizabal, O. M., Moura, R. S. T. D., Dotti, F. L., and Pedone, F. (2017). Efficient and deterministic scheduling for parallel state machine replication. In *2017 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 748–757.

[Sakouhi et al. 2018] Sakouhi, C., Khaldi, A., and Ghezal, H. B. (2018). An overview of recent graph partitioning algorithms. In *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA)*, pages 408–414. The Steering Committee of The World Congress in Computer Science, Computer . . . .

[Schneider 1990] Schneider, F. B. (1990). Implementing fault-tolerant services using the state machine approach: A tutorial. *ACM Computing Surveys (CSUR)*, 22(4):299–319.

[Stanton and Kliot 2012] Stanton, I. and Kliot, G. (2012). Streaming graph partitioning for large distributed graphs. In *Proceedings of the 18th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 1222–1230.

[Taft et al. 2014] Taft, R., Mansour, E., Serafini, M., Duggan, J., Elmore, A. J., Aboulnaga, A., Pavlo, A., and Stonebraker, M. (2014). E-store: Fine-grained elastic partitioning for distributed transaction processing systems. *Proceedings of the VLDB Endowment*, 8(3):245–256.

B.2 XX ESCOLA REGIONAL DE ALTO DESEMPENHO DA REGIÃO SUL

Simulador para medição de paralelismo em algoritmos de escalonamento para Replicação Máquina de Estados Paralela

João Gabriel Trombeta[1], **Odorico Machado Mendizabal**[1]

[1]Departamento de Informática e Estatística – Universidade Federal de Santa Catarina
Caixa Postal 476 – 88040-900 – Florianópolis – SC – Brasil

joao.gabriel.trombeta@grad.ufsc.br, odorico.mendizabal@ufsc.br

Resumo. *Ao desenvolver um novo algoritmo de escalonamento de requisições para Replicação Máquina de Estados Paralela, é difícil mensurar seu grau de paralelismo sob diferentes cargas de trabalho e configurações, ou compará-lo com técnicas existentes. Neste trabalho é proposto um simulador que abstrai custos de uma implementação real, para que sejam analisados possíveis ganhos de desempenho decorrentes exclusivamente das estratégias de escalonamento.*

1. Introdução

Replicação Máquina de Estado (RME) [Lamport 1978, Schneider 1990] é uma técnica utilizada para garantir tolerância a falhas em sistemas distribuídos, ao mesmo tempo que garante consistência forte. Em RME, todas as réplicas partem do mesmo estado inicial e executam as mesmas requisições na mesma ordem, garantindo assim que todas atravessem os mesmos estados. A execução sequencial representa uma limitação no desempenho do sistema, abrindo espaço para implementações de RME Paralela [Kotla and Dahlin 2004], que visam extrair paralelismo na execução de requisições, sem comprometer a consistência. Com o surgimento dessa técnica, estratégias para balancear a execução de requisições entregues às réplicas ganham importância.

Diversas técnicas foram propostas para o escalonamento de requisições em RMEP. CBase [Kotla and Dahlin 2004] utiliza um grafo para manter controle das dependências entre as requisições, em [Mendizabal et al. 2017] mapas de bits anotam informação de dependência entre comandos, enquanto lotes de requisições são utilizados para aumentar o taxa de ocupação nas *threads* executoras. Outras técnicas separam as requisições em classes de conflito e definem o relacionamento entre as classes [Alchieri et al. 2018]. Uma das dificuldades ao desenvolver uma nova técnica, porém, é medir o potencial ganho em paralelismo que a técnica pode extrair e como compará-la com técnicas já existentes, sem que haja a necessidade de implementá-la em um sistema completo.

Nesse trabalho é descrito um simulador de execução de requisições em RMEP, para analisar o paralelismo obtido por algoritmos de escalonamento, abstraindo demais custos de uma implementação real. Durante a simulação, requisições são geradas e enviadas para o algoritmo de escalonamento, que registra dados da simulação para posterior análise. As etapas da execução são modularizadas de forma que é possível editar aspectos pontuais da simulação para representar diferentes ambientes e comportamentos. Através dos dados levantados, é possível medir o potencial paralelismo obtido por políticas de escalonamento, observar como diferentes parâmetros afetam o desempenho, e comparar técnicas, antes de implementá-las em um sistema real.

2. Funcionamento do simulador

O uso do simulador passa pelas configurações iniciais, geração de requisições, execução e saída de dados. A Figura 1 mostra a arquitetura do simulador.

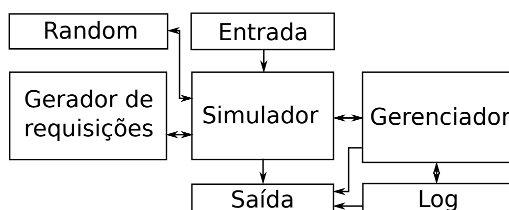


Figura 1. Arquitetura do simulador

As configurações do simulador, como número de chaves no sistema, quantidade e distribuição de requisições, são especificadas em um arquivo de entrada. É no arquivo de configuração que são definidos comportamentos que podem variar entre diferentes simulações, visando a análise da execução sob diferentes parâmetros.

Requisições podem ser geradas de maneira aleatória ou importadas de um arquivo externo. Existem dois tipos de requisições, requisições que acessam uma única chave, e requisições que acessam múltiplas chaves. Caso sejam importadas de um arquivo externo, o caminho de um arquivo definindo uma lista de requisições é esperado. Se for necessário gerar requisições aleatórias, o simulador chama o módulo `Random`, como demonstrado na Figura 1, para buscar geradores de números pseudo-aleatórios que satisfaçam a distribuição desejada.

Ao optar por gerar requisições aleatórias, em requisições de chave única é possível especificar a quantidade de requisições e a distribuição a ser utilizada durante a geração. Em requisições que acessam múltiplas chaves, é preciso definir a quantidade de requisições, o número mínimo e máximo de chaves envolvidas, e a distribuição utilizada para escolher o tamanho e as chaves envolvidas nas requisições. Atualmente estão implementadas as distribuições fixa, uniforme e binomial.

Durante a execução da simulação é assumido que todas as requisições tenham o mesmo tempo de execução, definido como uma unidade de tempo. Seguindo a arquitetura demonstrada na Figura 1, o Simulador interage com o módulo `Gerenciador`, que deve ser implementado estendendo uma classe de mesmo nome. É preciso que a nova classe implemente o método `execute_requests`, que deve realizar o escalonamento das execuções. Durante o escalonamento, é possível interagir com o `Log` para administrar o estado de cada $thread$ simulada.

Após a execução, o simulador escreve em um arquivo informações sobre as $threads$ simuladas. São apresentados o instante de tempo em que as *threads* terminaram de executar a última requisição, a quantidade de tempo em que estiveram ociosas e a porcentagem total que esse tempo representa. Essas informações são obtidas através do `Log`, utilizado durante o escalonamento. Além das informações coletadas pelo `Log`, o `Gerenciador` pode, através do método `export_data`, escrever dados adicionais. As requisições utilizadas durante a execução podem ser exportadas e reutilizadas em simulações futuras.

3. Análise de uso: comparando técnicas

Para mostrar o simulador em funcionamento, foram implementadas variações do `Gerenciador` para duas técnicas de escalonamento: CBase e corte em grafo. Dada a estrutura modular da ferramenta, outras estratégias de escalonamento podem ser facilmente Dada a estrutura modular da ferramenta, outras estratégias de escalonamento podem ser facilmente incorporadas ao simulador futuramente.

incorporadas ao simulador futuramente.

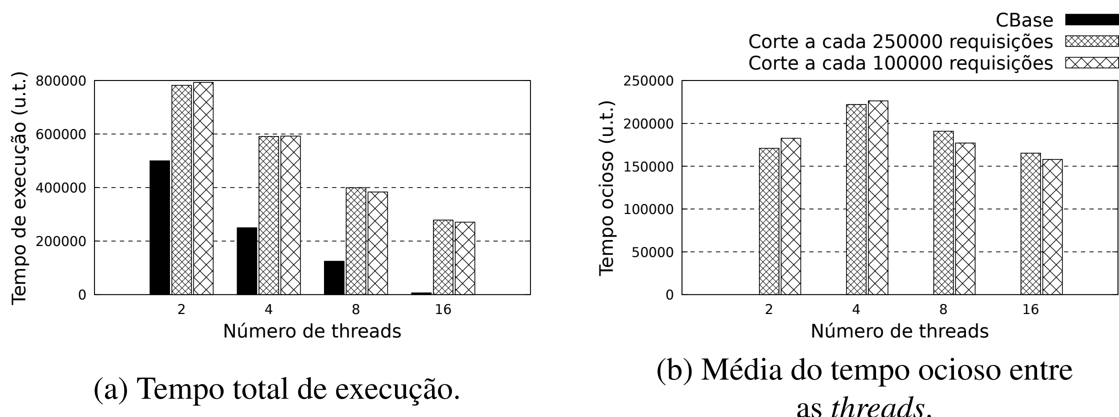
O CBase [Kotla and Dahlin 2004] cria um grafo de dependências, em que vértices são requisições e existe uma aresta (x, y) se, e somente se, a execução de x antecede y. $Threads$ acessam o grafo e executam as requisições representadas por vértices fonte, e depois disso atualizam o grafo, removendo o vértice. Esse algoritmo é considerado ótimo em exploração de paralelismo, porém implementações reais apresentam um grande custo devido ao acesso concorrente ao grafo e à detecção de conflitos [Mendizabal et al. 2017].

O corte mínimo em grafos é uma técnica de balanceamento de carga (e.g. [Hendrickson and Kolda 2000]). O grafo aqui é modelado de forma que os vértices são chaves, seu peso é a quantidade de vezes que foram acessadas, e uma aresta (a, b, x) diz que as chaves a e b foram acessados por uma mesma requisição x vezes, onde $x > 0$. O corte resulta em conjuntos disjuntos de chaves, chamados de partições, cada uma atribuída a uma $thread$. A execução de requisições é feita em paralelo por todas as $threads$, cada uma executando as requisições nas chaves em suas partições. Caso uma requisição acesse chaves que estão em partições diferentes, as $threads$ que possuem as chaves acessadas sincronizam a execução do comando, onde apenas uma executa o comando, e as demais ficam bloqueadas. O número de partições é definido como o mesmo número de *threads*, e inicialmente as chaves são distribuídas pelas partições utilizando *Round-Robin*. A biblioteca METIS [Karypis and Kumar 1998] foi utilizada para realizar o corte mínimo.

As simulações foram configuradas com 1.000 chaves e 1.000.000 de requisições, sendo que 75% das requisições acessam uma única chave e as demais acessam múltiplas chaves. Para gerar requisições de um único valor, foi utilizada a distribuição binomial na escolha da chave; para requisições que acessam múltiplas chaves, o número de chaves pode variar de 2 a 8, de acordo com uma distribuição binomial, e a escolha das chaves segue uma distribuição uniforme. Duas situações do corte em grafos foram exploradas, uma com reparticionamento a cada 250.000 requisições, e outra com reparticionamento a cada 100.000 requisições. Foram simulados cenários com 2, 4, 8 e 16 *threads*.

A Figura 2(a) mostra o gráfico do tempo necessário para a execução das requisições, enquanto a Figura 2(b) exibe o tempo médio em que as *threads* ficam ociosas, indicados em unidades de tempo (u.t.). A ociosidade é computada pela *thread* a cada instante em que aguarda pela sincronização com outra(s) *thread(s)*. Apesar do alto custo de implementação, o CBase é ideal em termos de exploração do paralelismo, o que causa um baixo tempo de execução e nenhuma ociosidade. Nos escalonamentos utilizando corte mínimo, o tempo em que as *threads* permanecem ociosas é maior, devido à necessidade de sincronizações. Isso reflete no tempo total de execução, maior do que o obtido com o CBase. É possível perceber como a frequência de reparticionamento pouco afetou o tempo de execução ou a ociosidade, e que uma frequência maior pode, por vezes, deteriorar o desempenho. O simulador tem como objetivo levantar informações sobre

número de sincronizações, ociosidade das *threads* e tempo de execução total de maneira simples, para que sejam considerados durante a concepção de novos algoritmos ou em comparações entre diferentes técnicas e configurações.



(a) Tempo total de execução.

(b) Média do tempo ocioso entre as *threads*.

Figura 2. Gráficos obtidos a partir das informações de saída do simulador.

4. Considerações finais

Com a introdução de Replicação Máquina de Estados Paralela, é necessário que sejam exploradas técnicas de escalonamento de requisições que consigam extrair melhor paralelismo do modelo. É apresentando, então, um simulador que abstrai custos de uma implementação real, para que durante o concebimento de um novo algoritmo seja possível analisar o potencial paralelismo da técnica sob diferentes configurações. Pode-se, também, comparar algoritmos existentes, como um instrumento de análise.

Agradecimentos

O presente trabalho foi realizado com o apoio do Conselho Nacional de Desenvolvimento Científico e Tecnológico - CNPq - Brasil.

Referências

Alchieri, E., Dotti, F., Marandi, P., Mendizabal, O., and Pedone, F. (2018). Boosting state machine replication with concurrent execution. In *2018 Eighth Latin-American Symposium on Dependable Computing (LADC)*.

Hendrickson, B. and Kolda, T. G. (2000). Graph partitioning models for parallel computing. *Parallel computing*, 26(12):1519–1534.

Karypis, G. and Kumar, V. (1998). A fast and high quality multilevel scheme for partitioning irregular graphs. *SIAM Journal on scientific Computing*, 20(1):359–392.

Kotla, R. and Dahlin, M. (2004). High throughput byzantine fault tolerance. In *International Conference on Dependable Systems and Networks, 2004*.

Lamport, L. (1978). Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*.

Mendizabal, O. M., De Moura, R. S., Dotti, F. L., and Pedone, F. (2017). Efficient and deterministic scheduling for parallel state machine replication. In *2017 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*.

Schneider, F. B. (1990). Implementing fault-tolerant services using the state machine approach: A tutorial. *ACM Computing Surveys (CSUR)*, 22(4):299–319.