UNIVERSIDADE FEDERAL DE SÃO CARLOS

CENTRO DE CIÊNCIAS EXATAS E DE TECNOLOGIA

PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

PARALELIZAÇÃO DE ALGORITMOS DE BUSCA DE DOCUMENTOS MAIS RELEVANTES NA WEB UTILIZANDO GPUS

ROUSSIAN DI RAMOS ALVES GAIOSO

ORIENTADOR: PROF. DR. HERMES SENGER

São Carlos – SP

Janeiro/2019

UNIVERSIDADE FEDERAL DE SÃO CARLOS

CENTRO DE CIÊNCIAS EXATAS E DE TECNOLOGIA

PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

PARALELIZAÇÃO DE ALGORITMOS DE BUSCA DE DOCUMENTOS MAIS RELEVANTES NA WEB UTILIZANDO GPUS

ROUSSIAN DI RAMOS ALVES GAIOSO

Tese apresentada ao Programa de Pós-Graduação em Ciência da Computação da Universidade Federal de São Carlos, como parte dos requisitos para a obtenção do título de Doutor em Ciência da Computação, área de concentração: Sistemas de Computação
Orientador: Prof. Dr. Hermes Senger

São Carlos – SP

Janeiro/2019

UNIVERSIDADE FEDERAL DE SÃO CARLOS

Centro de Ciências Exatas e de Tecnologia
Programa de Pós-Graduação em Ciência da Computação

Folha de Aprovação

Assinaturas dos membros da comissão examinadora que avaliou e aprovou a Defesa de Tese de Doutorado do candidato Roussian Di Ramos Alves Gaioso, realizada em 13/02/2019:

Prof. Dr. Hermes Senger
UFSCar

Prof. Dr. Ricardo Rodrigues Ciferri
UFSCar

Prof. Dr. Wellington Santos Martins
UFG

Prof. Dr. Edson Norberto Cáceres
UFMS

Prof. Dr. Paulo Sergio Lopes de Souza
USP

Certifico que a defesa realizou-se com a participação à distância do(s) membro(s) Wellington Santos Martins, Edson Norberto Cáceres e, depois das arguições e deliberações realizadas, o(s) participante(s) à distância está(ao) de acordo com o conteúdo do parecer da banca examinadora redigido neste relatório de defesa.

Dedico este trabalho à minha mãe, ao meu pai, aos meus irmãos, a minha esposa, por todo apoio e compreensão.

AGRADECIMENTOS

Primeiramente agradeço à minha família, pais, irmãos, esposa, Luke, que me apoiou na realização deste trabalho. À minha esposa Carmen, agradeço por estar sempre presente nos momentos mais complicados desta trajetória e por me dar forças para continuar em frente. A sua companhia e compreensão foram fundamentais para realização deste trabalho.

Aos professores Hermes Senger, Hélio Guardia e Veronica Gil Costa, agradeço pelas orientações, ensinamentos, conselhos e tempos dedicados à me orientar. Obrigado pelo tempo investido, sempre direcionando meus estudos e ordenando meus pensamentos. A orientação, a paciência e a amizade do Professor Hermes foram cruciais para a conclusão deste trabalho.

Ao professor Wellington Martins e ao grupo de pesquisa LDA, agradeço pelas discussões e oportunidades que contribuíram para minha formação e qualificação profissional. Agradeço fortemente pelo apoio nos momentos mais complicados na geração deste trabalho.

Aos amigos que conquistei em São Carlos durante os anos que realizei o doutorado, Ana Braatz, Steve Tmat, Amir Jalilifard, Almas Awan, Sarosh Farjam, Iram Awan, Diego, Cédric, obrigado por me conceder momentos inesquecíveis, alegres e descontraídos. Isso tornou minha estadia em São Carlos e meus estudos muito mais fácil. Fica a minha gratidão.

Aos meus amigos Alexandre Ferreira, Juliano Geraldo, Juliana Moreira e a todos que sempre estiveram presentes de alguma forma, agradeço por serem verdadeiros companheiros.

Faça ou não faça. Tentativa não há.

Mestre Yoda - Star Wars Episódio 5 – O Império contra-ataca

RESUMO

As máquinas de busca estão enfrentando desafios de desempenho devido à grande quantidade de documentos e ao aumento de cargas de consultas no ambiente Web. O sucesso de uma máquina de busca está relacionado à capacidade do sistema de processamento de consultas de encontrar, em um curto intervalo de tempo, documentos que correspondam às necessidades de informações expressas nas consultas dos usuários. Apesar da grande quantidade de documentos, os usuários estão mais interessados em poucos documentos de resultados de uma consulta. Isso faz com que haja poucos documentos que são altamente relevantes na maioria das consultas. Os algoritmos de poda dinâmica DAAT vêm explorando a eficiência dos sistemas de processamento de consulta evitando perder tempo ao classificar documentos que provalvemente não são relevantes. Para lidar com a escala e a dinâmica do tráfego de consultas do usuário, o processamento de consulta precisa fazer o uso eficiente dos recursos do hardware.

O objetivo principal desta tese de doutorado é investigar o uso da computação paralela no processo de identificar os documentos mais relevantes a uma consulta realizando processamento na arquitetura GPU. Para isso, este trabalho apresenta estratégias de paralelização de algoritmos que visam a reduzir a latência de resposta de uma dada consulta e a aumentar a vazão das consultas. As propostas de paralelização são bem adequadas à categoria de algoritmos DAAT e aos algoritmos de poda dinâmica. Na categoria DAAT, estratégias de particionamento são oferecidas de modo que realizam uma investigação na localização das ocorrências de um mesmo documento na hierarquia de memória da GPU. No nível dos algoritmos de poda dinâmica, políticas de propagação de *threshold* entre os processadores são propostas e os impactos gerados na eficiência dos algoritmos paralelos são analisados. Para mostrar a eficiência na prática, as propostas paralelas foram implementadas e experimentadas na arquitetura da GPU Pascal e obtiveram um desempenho de $\sim 4\times$ a $\sim 40\times$ em relação aos algoritmos fundamentais.

Palavras-chave: Busca na Web, Processamento de Consultas, Algoritmos DAAT, Algoritmos de Poda, Algoritmo WAND, Algoritmo MaxScore, Algoritmos Paralelos, Arquitetura GPU

ABSTRACT

Search engines are facing performance challenges because of the large number of documents and the increase of query loads in the Web environment. The success of a search engine is related to the ability of the query processing system to find documents that match the needs of information expressed in user queries in a short time interval. Despite the large amount of documents, users are more interested in fewer results in a query. This causes few documents to be highly relevant in most queries. DAAT dynamic pruning algorithms have been exploring the efficiency of query processing systems, avoiding wasting time sorting documents that are not likely to be relevant. To handle the scale and dynamics of user query traffic, query processing needs to make efficient use of hardware resources.

The main objective of this doctoral thesis is to investigate the use of parallel computing in the process of identifying the most relevant documents to a given query in the GPU architecture. For this, strategies of parallelization of algorithms that aim to reduce the latency of response of a given query and to increase the flow of queries are proposed and evaluated in the GPU. The parallelization proposals are well suited to the category of DAAT algorithms and dynamic pruning algorithms. In the DAAT category, partitioning strategies are offered in a way that performs an investigation into the location of occurrences of the same document in the memory hierarchy of the GPU. At the level of dynamic pruning algorithms, threshold propagation policies among processors are proposed and the impacts generated on the efficiency of the parallel algorithms are analyzed. To verify efficiency in practice, the parallel proposals were implemented and tested in the Pascal GPU architecture and obtained a performance of $\sim 4\times$ to $\sim 40\times$ relative to the fundamental algorithms.

Keywords: Web Search, Query Processing, DAAT Algorithms, Pruning Algorithms, WAND Algorithm, MaxScore Algorithm, Parallel Algorithms, GPU

LISTA DE FIGURAS

LISTA DE TABELAS

SUMÁRIO

Capítulo 1

INTRODUÇÃO

A Web tornou-se um repositório universal de conhecimento que permite compartilhar informações como nunca visto antes. Contudo, procurar informações é difícil, caro e muitas vezes impossível. As máquinas de busca têm mudado o modo de acesso das pessoas à informação, tornando-o fácil, de modo que qualquer um pode obter informações em segundos, digitando algumas palavras em uma caixa de pesquisa (CAMBAZOGLU; BAEZA-YATES, 2015).

Máquinas de busca são sistemas que oferecem, a partir de interesses de usuários, o serviço de identificar documentos relevantes em coleções não estruturadas de documentos. Esses sistemas adotam uma estrutura de dados chamada de índice invertido capaz de organizar e obter as informações contidas na coleção de documentos a partir de consultas. Essa estrutura é constituída de vocabulário e listas invertidas, as quais contêm as ocorrências dos termos nos documentos.

Para identificar documentos a partir de uma consulta, é necessário que as máquinas de busca pesquisem nas listas invertidas dos termos da consulta. Duas técnicas básicas que são utilizadas pelas máquinas de busca se destacam: *Document-At-A-Time* (DAAT) e *Term-At-A-Time* (TAAT). Em DAAT, as listas são atravessadas simultaneamente, enquanto em TAAT somente uma lista invertida é processada por vez.

O processo de identificar os documentos mais relevantes na coleção de documentos a uma dada consulta é conhecido como processamento de consultas (CAMBAZOGLU; BAEZA-YATES, 2015). Comumente, esse problema consiste em percorrer as listas invertidas dos termos da consulta, aplicar técnicas de similaridade em cada documento dessas listas, ordenar os documentos em ordem decrescente de similaridade e selecionar os k documentos mais semelhantes à consulta, onde $k \in \mathbb{Z}$. A forma mais básica de processamento de consulta é chamada de processamento de consulta booleana, nas quais se destacam a conjuntiva (AND) e a disjuntiva

(OR). Em geral, as consultas disjuntivas têm sido tradicionalmente usadas na comunidade de recuperação de informação, ao passo que as conjuntivas são mais empregadas em máquinas de busca Web (DING; SUEL, 2011).

Uma dificuldade no processamento de consulta é o tamanho das listas invertidas. Os algoritmos de poda (*early termination*) são uma classe importante de algoritmos que tentam resolver esse problema, limitando o escopo das buscas. Quando todos os documentos são avaliados completamente pelo algoritmo de processamento de consulta, é dito que este é um algoritmo exaustivo, enquanto que aqueles que visam avaliar somente documentos com alta possibilidade de serem os mais relevantes à consulta, são conhecidos por algoritmos de poda.

À vista de grandes coleções de documentos que crescem constantemente, as máquinas de busca precisam realizar buscas de documentos para usuários que esperam, independente da complexidade e quantidade, baixos tempos de resposta. Enquanto isso, uma demanda crescente de consultas, cuja qualidade dos resultados é requisitada, chegam nessas máquinas de busca. Dessa forma, o processamento eficiente de consultas torna-se cada vez mais importante.

Nesses termos, vê-se que a eficiência do processamento de consultas afeta diretamente a satisfação dos usuários. O tempo de resposta elevado pode distrair os usuários e levá-los a emitir menos consultas do que o habitual, a reduzir o uso de máquinas de busca a longo prazo ou a mudar a máquina de busca utilizada (ARAPAKIS; BAI; CAMBAZOGLU, 2014). Nesses sistemas, a eficiência geralmente é medida por duas métricas complementares. Uma é a medida da velocidade na qual o processador de consultas recupera e apresenta os resultados ao usuário, conhecida por latência de resposta. A segunda métrica, vazão, é uma medida do número de consultas de diferentes usuários que podem ser respondidas simultaneamente pela máquina de busca dentro de um determinado período de tempo. Um sistema de busca eficiente deve ter uma latência de resposta baixa suficiente para não afetar o comportamento do usuário e, ao mesmo tempo, manter sua capacidade de operação sob intenso tráfego de consultas (CAMBAZOGLU; BAEZA-YATES, 2015).

Além da eficiência, as máquinas de busca são avaliadas pela eficácia nas buscas que, por sua vez, associa-se à eficiência. A eficácia no processamento de consultas é a combinação dos resultados de busca com o atendimento às informações que o usuário deseja. Técnicas para melhorar a eficácia das consultas aumentam consideravelmente o custo do processamento das consultas (BILLERBECK; ZOBEL, 2006). Por isso, diversas pesquisas têm buscado a eficiência no processamento das buscas com intuito de aumentar a qualidade dos resultados (TURTLE; FLOOD, 1995) (CARMEL et al., 2001) (BRODER et al., 2003) (DING et al., 2009) (DING; SUEL, 2011) (SHAN et al., 2012) (DIMOPOULOS; NEPOMNYACHIY; SUEL, 2013a) (TA-

DROS, 2015).

Modelos complexos de classificação que combinam técnicas de aprendizado de máquina são aplicados pelas máquinas de busca Web atuais para prover uma elevada qualidade nos resultados. No entanto, a aplicação em cada documento traz um alto custo computacional. Para evitar esse custo, a maioria das máquinas de busca adota classificação em duas fases. Na primeira fase, aplicam-se modelos simples precisos, como similaridade de cosseno (SALTON, 1971) e BM25 (ROBERTSON; WALKER; HANCOCK-BEAULIEU, 1995), em todos os documentos das listas invertidas dos termos da consulta para selecionar um pequeno subconjunto de documentos com potencial de relevância frente à consulta. Na segunda fase, uma função de ordenação baseada em aprendizado de máquina é aplicada somente no conjunto de documentos candidatos obtidos na primeira fase, gerando uma nova ordenação final que é apresentada ao usuário. Portanto, nesse contexto, a redução do tempo de processamento da primeira fase é extremamente necessária para alcançar melhores resultados no processamento de consulta com baixa latência de resposta.

1.1 Motivação

Devido à taxa elevada de crescimento do número de documentos na Web e do número de consultas emitidas por período de tempo, a escalabilidade das máquinas de busca tornou-se um verdadeiro desafio. Isso porque, a obtenção de documentos mais relevantes para uma consulta requer aplicação de modelos de similaridade em cada documento dos termos da consulta sob uma demanda crescente. O processamento paralelo tem potencial para reduzir significativamente o tempo de execução por consulta, tornando-se uma solução atraente para aquelas máquinas de busca que necessitam reduzir o tempo de resposta para grandes coleções.

Os processadores *manycore* são considerados processadores massivamente paralelos que oferecem alto desempenho, eficiente uso de energia e baixo custo. Um exemplo dessa categoria de processadores são as Unidades de Processamento Gráfico (GPU). GPUs são consideradas processadores com suporte eficiente para uma grande quantidade de *threads*. O alto poder computacional e a acessibilidade das GPUs tem levado a um uso crescente desta tecnologia.

Com o advento das GPUs, diversas pesquisas passaram a utilizar seus núcleos de processamento para melhorar a eficiência do processamento de consulta, oferecendo estratégias e algoritmos paralelos que se adequassem à arquitetura. No entanto, as GPUs têm uma arquitetura e organização de memória diferentes do que comumente se apresenta em arquiteturas com múltiplas CPUs, as quais impõem algumas restrições em termos de desenvolvimento de

algoritmos apropriados, exigindo novas soluções e novas abordagens de implementação. Embora trabalhos anteriores tenham produzido evoluções no desempenho no processamento de consulta, diversas restrições foram apresentadas para que o processamento de consulta fosse viabilizado na GPU, por exemplo, no tamanho das listas invertidas dos termos das consultas, no número de documentos retornados pelos algoritmos, na abordagem de percorrer as listas invertidas e no número de consultas processadas simultaneamente (DING et al., 2009; TADROS, 2015; HUANG et al., 2017; LIU; WANG; SWANSON, 2018).

A principal motivação desta tese é de explorar a arquitetura paralela da GPU de forma que aumente o desempenho do processamento de consulta, permitindo que consultas sejam processadas de maneira eficiente mesmo em um ambiente cujo o conjunto de documentos seja crescente e com um possível cenário de aumento de demanda de consultas. Embora o escopo desta tese não inclua melhorias na eficácia dos resultados, espera-se que os resultados divulgados possibilitam o uso de estratégias que aperfeçoem a qualidade do processamento. Em um processamento de consulta em duas fases, uma resposta rápida na primeira fase possibilita o uso de técnicas mais sofisticadas na segunda fase, o que potencialmente pode melhorar a qualidade.

1.2 Objetivos

Dadas uma consulta textual, uma coleção de documentos e uma máquina de busca, o objetivo geral desta presente tese de doutorado é investigar o uso de paralelismo nos algoritmos de processamento de consultas no que se refere à eficiência de respostas às requisições e demandas de consultas. Consideramos os algoritmos de poda DAAT para realizar essa investigação por serem algoritmos estados da arte na recuperação de informação.

A principal plataforma escolhida para o desenvolvimento desta tese foi a arquitetura *manycore* CUDA (*Compute Unified Device Architecture*), das GPUs da NVIDIA. No entanto, ao decorrer da pesquisa, partirmos do pressuposto que quaisquer outros processadores *manycores* que ofereçam características de paralelismo de granularidade grossa e fina ao mesmo tempo podem ser utilizados na investigação no problema de processamento de consultas.

Os objetivos específicos desta tese são:

- Analisar, desenvolver e avaliar estratégias de particionamento de dados entre os processadores, para realizarem o processamento de consultas de forma independente;
- Propor, implementar e avaliar estratégias de paralelização de algoritmos de processamento de consulta, para analisar o comportamento das soluções no contexto do aumento

na quantidade de documentos em uma determinada coleção;

- Propor, implementar e avaliar estratégias de paralelização de algoritmos de processamento de lote de consultas, com o intuito de melhorar a vazão do processamento e a utilização dos recursos de processamento da GPU;
- Propor, implementar e avaliar versões paralelas dos algoritmos WAND e MaxScore de granularidade fina que compartilham os recursos de um processador.

Para alcançar o objetivo deste trabalho, o corpus TREC ClueWeb09 de 50,2 milhões de documentos foi utilizado e indexado pela máquina de busca Terrier IR 4.2 (TERRIER, 2014) (OUNIS et al., 2005). Os algoritmos paralelos foram implementados na linguagem C/CUDA e executados em uma única NVIDIA GPU Titan Xp com 3840 CUDA cores. Para avaliar os algoritmos paralelos, algoritmos sequenciais fundamentais de poda, WAND (BRODER et al., 2003) e MaxScore (TURTLE; FLOOD, 1995), foram implementados e inseridos na máquina de busca Terrier IR e processados em um único núcleo do processador Intel(R) Core(TM) i7-5820K de 3.30GHz. Os algoritmos foram avaliados sob consultas booleanas sintéticas do tipo OR e AND e consultas de registros reais.

1.3 Contribuições

Abaixo segue o resumo das contribuições deste trabalho:

- Desenvolvimento de estratégias de particionamento de dados, homogênea e heterogênea, em hierarquia de memórias de processadores paralelos;
- Estratégias de paralelização de algoritmos, granularidade grossa e granularidade fina, para o problema de processamento de consultas que se adequam nas categorias de algoritmos DAAT e algoritmos de poda;
- Criação de políticas de propagação de *threshold*, limiar mínimo, para os algoritmos de poda: local, *safe R-Shared* e *safe WR-Shared*;
- Implementações em C/CUDA das propostas de paralelização do processamento de consulta para arquitetura *manycore* GPU;
- Avaliações de desempenho na latência de resposta das propostas de paralelização do problema de processamento de consulta em arquitetura *manycore* GPU. A proposta de

paralelização com a versão paralela do WAND alcançou $\sim 25\times$ de speedup com as partições homogêneas e $\sim 12\times$ com as heterogêneas;

- Estratégias de paralelização, granularidade grossa e granularidade fina, para o problema de processamento de lote de consultas;
- Implementações em C/CUDA das propostas de paralelização do processamento de lote de consultas para arquitetura *manycore* GPU;
- Avaliações de desempenho das propostas de paralelização do processamento de lote de consultas em arquitetura *manycore* GPU. A primeira proposta do processamento de lote de consultas com a versão paralela do MaxScore atingiu um speedup de $\sim 7\times$ com partições heterogêneas.

1.4 Organização da Tese

O restante desta tese é organizado em 7 capítulos:

- O Capítulo 2 fornece uma descrição dos conceitos básicos de busca na Web e detalhes dos algoritmos de processamento de consulta;
- O Capítulo 3 descreve os detalhes da arquitetura CUDA GPU e os principais modelos paralelos;
- O Capítulo 4 fornece propostas de paralelização de algoritmos para o processamento de consulta;
- O Capítulo 5 avalia as propostas de paralelização do processamento de consultas;
- O Capítulo 6 fornece propostas de paralelização de algoritmos para o processamento de lote de consultas;
- O Capítulo 7 avalia as propostas de paralelização do processamento de lote de consultas;
- O Capítulo 8 decorre dos trabalhos relacionados;
- O Capítulo 9 apresenta a conclusão desta tese de doutorado.

Capítulo 2

BUSCA NA WEB

As atuais máquinas de busca têm sido desafiadas a responder, em tempo cada vez menor, milhares de consultas por segundo sobre um ambiente de coleções com uma imensa quantidade de documentos que tende a crescer continuadamente. Diversas técnicas para otimizar as buscas têm sido propostas a fim de reduzir o impacto desse crescimento na eficiência e eficácia das máquinas de buscas.

Essas técnicas abrangem várias áreas em máquinas de busca Web, entre as quais podemos destacar: construção, compressão, particionamento e distribuição de índices invertidos, *caching*, métodos de ordenação de documentos e outras (WU; CHUANG; CHEN, 2008) (JIANG; YANG, 2016) (CUTTING; PEDERSEN, 1990) (ROJAS; GIL-COSTA; MARIN, 2013). Neste capítulo, os principais conceitos relacionados a máquinas de busca são descritos juntamente com as principais técnicas utilizadas para otimizar o processamento de busca.

2.1 Conceitos Básicos

O crescimento acelerado da quantidade de informações disponíveis tornou as máquinas de busca em serviços Web essenciais (WU; CHUANG; CHEN, 2008). Máquinas de busca são sistemas, que a partir de interesses de usuários, identificam documentos relevantes em coleções não estruturadas de documentos. Esses interesses são transmitidos para os sistemas computacionais em formato de consulta (*query*) que contém um conjunto de termos.

As máquinas de busca trabalham sobre um conjunto de documentos não estruturados, definido como coleção de documentos. Essas coleções variam drasticamente em tamanho. Os tipos de documentos de uma coleção podem ser variados, abrangendo páginas da Web, bibliografias, imagens, registros históricos, e-mails e outros. Normalmente, as máquinas de busca atuais

adotam uma estrutura de dados que organiza os documentos de modo que facilita a obtenção de informações a partir de consultas. Essa estrutura de dados, chamada de *índice invertido*, oferece um suporte rápido ao processamento de consultas (MANNING et al., 2008) (ZOBEL; MOFFAT, 2006) (CUTTING; PEDERSEN, 1990) (ROJAS; GIL-COSTA; MARIN, 2013).

O índice invertido permite encontrar de forma eficiente documentos que contêm os termos da consulta de entrada. Essa estrutura mapeia cada termo relevante (palavras ou itens atômicos de busca) existente em uma coleção com os documentos que o contêm. Dessa forma, duas estruturas são formadas: (1) vocabulário e (2) listas invertidas (listas de *postings*). A Figura 2.1 exemplifica um índice invertido com o vocabulário e as listas invertidas a partir de uma coleção com três documentos. Observa-se que uma lista de *stop words* é utilizada para remover do vocabulário as palavras extremamente comuns e que pouco contribuem para selecionar os documentos de acordo com a necessidade dos usuários. O uso de listas de *stop words* reduz drasticamente o tamanho das listas invertidas (MANNING et al., 2008).

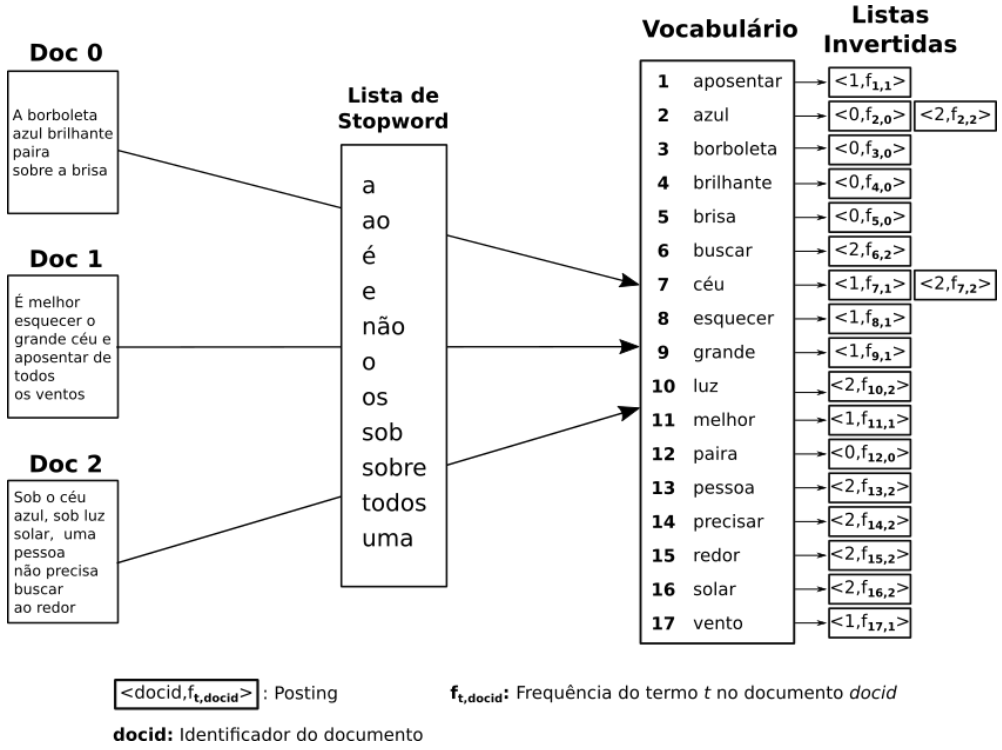

Figura 2.1: Índice invertido.

Na construção do índice invertido, todos os termos da coleção são aglutinados em um conjunto, chamado de vocabulário ou dicionário. O vocabulário permite o acesso aos dados relativos a um termo por meio de palavras-chave (palavras de interesse). Na literatura, duas formas de organizá-lo se destacam. Os termos podem ser dispostos de forma ordenada, em que uma estrutura de dados árvore, representada pela *B+ tree*, é a mais utilizada. A segunda forma de organização é realizada por meio de tabelas *hash* (SONG et al., 2015) (CUTTING; PEDER-

SEN, 1990) (CAMBAZOGLU; BAEZA-YATES, 2015).

Considerando uma coleção de N documentos, assume-se que cada documento possui um identificador único, nomeado de docID, entre 0 e $N-1$. Todos os termos no vocabulário possuem uma lista invertida, também chamada de lista de *postings*, que são compostos por *posting*. Os *postings* são estruturas de dados constituídos por um docID, frequência do documento ($f_{t,docID}$) e outras informações referentes ao termo e ao documento. A frequência do documento é o número de ocorrências de um termo t no documento identificado pelo docID. Com essa constituição, uma lista invertida contém todas as ocorrências do termo correspondente a ela na coleção de documentos (SONG et al., 2015) (MANNING et al., 2008) (JIANG; YANG, 2016).

A utilização do índice invertido pelos sistemas de busca Web traz diversas vantagens, entre elas destacam-se as seguintes: (1) obtenção direta de termos e documentos que os contêm; (2) remoção de informações redundantes, por exemplo, quando um termo ocorre diversas vezes em um mesmo documento, um único registro desse documento é mantido; (3) várias características relacionadas aos termos, aos documentos e às coleções podem ser armazenadas separadamente ou podem ser combinadas diretamente no índice invertido; e (4) possibilidade de processar em paralelo os documentos mais relevantes a uma dada consulta (SONG et al., 2015).

Em muitos sistemas de busca comerciais Web, o módulo responsável pela criação do índice invertido é o sistema de indexação, também conhecido por indexador. O índice invertido é reconstruído periodicamente devido a questões de manutenção (por exemplo, inserção de novos documentos), enquanto que consultas são continuadamente avaliadas. A redução do tempo na construção do índice invertido é crucial à qualidade desses sistemas de busca, pois evita-se que versões antigas continuem a ser utilizadas nas buscas de documentos.

Com intuito de obter uma eficiência aceitável comercialmente na construção, muitos algoritmos de construção buscam técnicas escaláveis que tentam minimizar o número de buscas em discos. As principais técnicas de construção distinguem-se pelo número de passos realizados para construir o índice invertido, duas são as mais utilizadas, sendo a técnica de único passo (*Single-pass in-memory indexing*) e a técnica de dois passos (*two-pass index construction*). Esta realiza a construção do vocabulário em um primeiro processamento para em seguida construir o índice invertido, enquanto a técnica de um passo constrói o vocabulário e as listas invertidas ao mesmo tempo (BAEZA-YATES; RIBEIRO-NETO et al., 1999) (MANNING et al., 2008) (CROFT; METZLER; STROHMAN, 2010).

2.2 Medidas de Qualificação de Máquinas de Buscas

As máquinas de busca em geral são avaliadas de duas formas. A primeira diz respeito à eficiência que geralmente está associada a duas métricas complementares. A métrica mais visível é o tempo de resposta, também conhecida como latência de resposta, experimentado por um usuário entre a emissão de uma consulta e o recebimento dos resultados. A capacidade de resposta do sistema em uma situação de tráfego intenso de consultas, quando muitos usuários simultâneos precisam ser suportados, é a segunda métrica complementar da eficiência, conhecida como vazão de consultas ou taxa de transferência. Um sistema de consultas eficiente deve poder continuar a operar sob intenso tráfego de consultas, mantendo um determinado pico de processamento (BÜTTCHER; CLARKE; CORMACK, 2016).

A segunda forma de avaliação, chamada de eficácia, de uma máquina de busca refere-se aos aspectos da qualidade da busca. Essa avaliação torna-se mais difícil de se medir do que a eficiência, uma vez que ela depende totalmente do julgamento humano. Além de ser eficiente, uma máquina de busca deve ser altamente eficaz na correspondência dos documentos retornados com as necessidades de informações identificadas. A ideia principal por de trás da medição de qualidade é a noção de relevância, em que um documento é considerado relevante para uma consulta se seu conteúdo, completo ou parcialmente, satisfaz a necessidade de informação representada pela consulta. Para determinar a relevância, um avaliador humano analisa um par de documento-assunto e atribui um valor de relevância. O valor de relevância pode ser binário (relevante ou não) ou gradual. Além disso, a eficácia pode captar outros aspectos da qualidade da busca, como a diversidade dos resultados. As duas medidas amplamente utizadas para avaliar a qualidade dos resultados das máquinas de busca (BAEZA-YATES; RIBEIRO-NETO et al., 1999) são: precisão e *recall* (revocação). Elas são definidas da seguinte forma.

A *precisão (P)* é a proporção de documentos recuperados que são relevantes:

$$P = \frac{\#(\text{documentos relevantes recuperados})}{\#(\text{documentos recuperados})} \tag{2.1}$$

A medida *recall (R)* é a proporção de documentos relevantes que são recuperados:

$$R = \frac{\#(\text{documentos relevantes recuperados})}{\#(\text{documentos relevantes})} \tag{2.2}$$

Em outras palavras, o *recall* indica a fração de documentos relevantes que aparece no conjunto de resultados, enquanto a precisão indica a fração do conjunto de resultados que é relevante. Intuitivamente, o *recall* mede o quanto de documentos são relevantes que a máquina

de busca está buscando para uma consulta e a precisão mede quanto de documentos não relevantes são rejeitados. A Figura 2.2 ilustra essas medidas por meio de uma classificação dos documentos da coleção em relevantes e não relevantes.

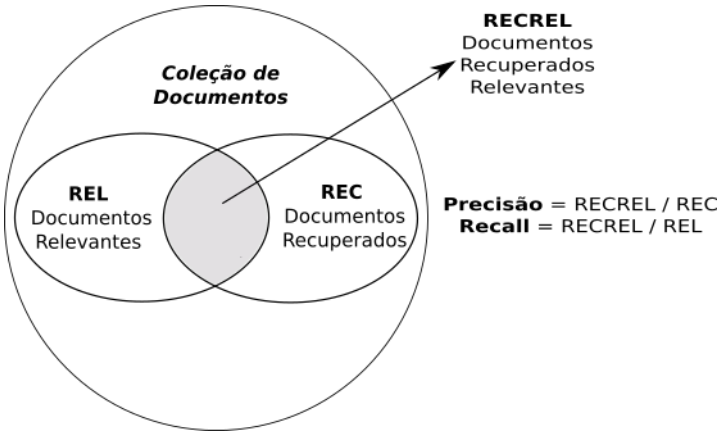

Figura 2.2: Precisão e *Recall* em um conjunto de documentos classificados em relevantes e não relevantes para uma dada consulta.

2.3 Medidas da Relação entre Termos e Documentos

Métodos de ordenação, também chamados de métodos de classificação, são usados pelas máquinas de busca para classificar os documentos mais similares a uma dada consulta. Os pesos dos termos são empregados nessa classicação e a forma específica de um peso é determinada pelo modelo de ordenação utilizado. Existem muitas variações desses pesos, mas todas elas são baseadas em uma combinação de dados estatísticos das relações entre termos, documentos e consulta (ZOBEL; MOFFAT, 2006). As principais medidas são destacadas:

1. N: número de documentos na coleção;
2. n: número de termos indexados da coleção;
3. $f_{t,d}$: frequência do termo t no documento d;
4. $f_{t,q}$: frequência do termo t na consulta q;
5. f_t: número de documentos em que o termo t ocorre;
6. idf_t: frequência inversa do documento definida como:

$$idf = \log \frac{N}{f_t} \tag{2.3}$$

Cada termo possui um único idf. O peso do idf é chamado de frequência inversa do documento, pois fornece pesos altos para termos que ocorrem com baixa frequência nos documentos. Assim, um termo que ocorre com alta frequência deve ter um peso menor do que um termo com baixa frequência nos documentos, ao proceder dessa forma, o idf pode ser traduzido como uma especificação da raridade dos termos na coleção.

2.4 Sistema de Processamento de Consultas

A qualidade de uma máquina de busca está relacionada diretamente com a habilidade do sistema de processamento de consultas de retornar, em tempo razoável ao usuário, os documentos que melhor combinam com a consulta de entrada. Embora existam diferenças entre diversos métodos de processamento de consulta, três ações são fundamentais: (1) a interpretação da consulta dada pelo usuário; (2) a avaliação, ordenação e recuperação de documentos dentro da coleção; e (3) a preparação e apresentação dos documentos recuperados (CAMBAZOGLU; BAEZA-YATES, 2015).

A interpretação de consultas envolve a transformação da consulta de entrada, expressa em palavras, em consulta intermediária, que é representada em uma linguagem interna do sistema de busca. Essa transformação é realizada através de aplicações de funções em série nas consultas no estilo de um *pipeline*. Entre essas funções, por exemplo, estão a normalização, a *lemmatization*, a*stemming*, a expansão, a correção, a eliminação de *stop words* e outras. As funções que são aplicadas nas consultas precisam ser as mesmas do que às aplicadas nos termos dos documentos na construção do índice invertido, para que seja fornecido consistência no processo de combinação entre documentos e termos da consulta (MANNING et al., 2008).

O processo de avaliar, ordenar e recuperar os documentos da coleção a cada consulta recebida possui um alto custo computacional. Esse processamento avalia todos documentos dos termos da consulta, atribuindo um peso para cada um deles. Em seguida, os documentos são ordenados de acordo com o peso atribuído e somente os k documentos com maiores pesos são retornados ao usuário, onde $k \in \mathbb{Z}$ é parametrizado no sistema. Existem diversos estudos que visam a melhorar a eficiência da ordenação de consultas (CAMBAZOGLU; BAEZA-YATES, 2015) (TURTLE; FLOOD, 1995) (CARMEL et al., 2001) (BRODER et al., 2003).

As coleções de documentos Web são extremamente extensas. Isso impossibilita que elas sejam mantidas em uma única máquina. Essa característica faz com que as coleções sejam particionadas em uma plataforma distribuída e organizada em arquitetura mestre-escravo (*master-slave*). Assim, cada nó de processamento constrói um índice invertido a partir de sua partição

das coleções de documentos (CAMBAZOGLU; BAEZA-YATES, 2015).

A máquina mestre (*broker machine*) envia as consultas de entrada para todos os nós de processamento (*slaves*). Cada nó de processamento avalia, ordena e envia à máquina mestre os k documentos mais relevantes. Ao receber os conjuntos de documentos, cada qual de origem diferente, uma agregação é realizada pelo sistema de processamento de consulta da máquina mestre. Otimizações diversas são aplicadas, adicionando mais informações aos resultados, por exemplo, usando modelos de aprendizado de máquina. Os filtros e as classificações podem ser executados novamente antes da apresentação dos resultados com a finalidade de aumentar a consistência das informações (ROJAS; GIL-COSTA; MARIN, 2013) (CAMBAZOGLU; BAEZA-YATES, 2015).

2.4.1 Modelos de Classificação

Estimar a relevância de documentos é um dos principais objetivos de um sistema de processamento de consulta. A partir de uma consulta, somente um pequeno conjunto de documentos relevantes é apresentado ao usuário, embora o sistema opere sobre uma grande coleção de documentos que cresce continuadamente. Os documentos são apresentados ao usuário em ordem decrescente de relevância. A relevância é estimada por meio de métricas específicas (CAMBAZOGLU; BAEZA-YATES, 2015).

O modelo *Boolean* é um modelo simples de recuperação do sistema de busca baseado na álgebra booleana. As consultas são do tipo booleano e todos documentos e a consulta são considerados como *bag-of-words*. Essa estratégia de classificação é baseada em uma decisão binária, ou seja, os pesos dos termos $w_{t,d} \in 0,1$, onde t é o termo da consulta e d é o identificador do documento. Os documentos recuperados retêm a combinação exata da especificação da consulta. Isso implica que esses documentos são equivalentes em relevância. As máquinas atuais de busca não se limitam a esse modelo. Consultas que possuem termos com alta frequência na coleção tornam o uso desse modelo inviável no ponto de vista da eficiência, pois o conjunto de documentos recuperados pode conter milhares de documentos (BAEZA-YATES; RIBEIRO-NETO et al., 1999) (LONG; SUEL, 2003) (MANNING et al., 2008).

A representação do conjunto de documentos em um espaço vetorial é definida pelo modelo de Espaço Vetorial (*Vector Space Model*) (SALTON, 1971), onde cada termo do vocabulário possui um peso para cada documento da coleção. Esse peso é usado para computar o grau de similaridade entre cada documento da coleção e os termos da consulta. Nesse modelo vetorial, um peso $w_{t,d}$ é um valor positivo e não-binário do termo t no documento d; e a consulta e todos documentos são considerados vetores no espaço vetorial. Assim, um vetor possui n pesos, onde

n é o número de termos indexados da coleção. Por simplificação, quando há ausência de um termo t em um documento ou consulta, o valor escalar que representa o peso desse termo é zero. Esse modelo leva os documentos a serem considerados como *bag-of-words*, cuja a ordem dos termos nos documentos não é considerada. Dada uma consulta, o modelo ordena os documentos em grau de similaridade. Essa similaridade pode ser quantificada pela *similaridade de cosseno*, definida pela Equação 2.4 (BAEZA-YATES; RIBEIRO-NETO et al., 1999) (ZOBEL; MOFFAT, 2006) (MANNING et al., 2008):

$$sim(q,d) = \frac{\overrightarrow{V}_q \cdot \overrightarrow{V}_d}{|\overrightarrow{V}_q| . |\overrightarrow{V}_d|} \tag{2.4}$$

onde o numerador representa o produto vetorial e o denumerador é o produto das normas dos vetores do documento e da consulta. A pontuação de cada documento é dada pela similaridade com a consulta. Os documentos são ordenados em ordem decrescente da similaridade e, assim, a pontuação resultante pode ser usada para selecionar os k documentos com as maiores pontuações, onde $k \in \mathbb{Z}$.

Os Modelos Probabilísticos (MARON; KUHNS, 1960) (ROBERTSON; JONES, 1976) (JONES; WALKER; ROBERTSON, 2000) estimam a relevância de cada documento de acordo com o interesse do usuário a partir de medidas de probabilidade, de modo que a relevância depende somente da consulta e dos documentos. Dentre esses modelos, destacam-se os modelos Okapi BM25 (ROBERTSON; WALKER; HANCOCK-BEAULIEU, 1995) (ROBERTSON; ZARAGOZA, 2009) e Divergence From Randomness (DFR) (AMATI; JOOST; RIJSBERGEN, 2003) pela predominância na literatura.

De acordo com Billerbeck e Zobel (BILLERBECK; ZOBEL, 2006), diversas técnicas para melhorar a eficácia das consultas podem ser aplicadas nas máquinas de busca. Essas técnicas utilizam funções mais complexas para obter ganhos na eficácia dos resultados. Tais funções comumente dependem da combinação de características dos elementos do conjunto de documentos. Para implementá-las de modo que se mantém o atendimento aos requisitos de eficiência, as máquinas de busca dividem o processamento da consulta em duas fases. Na fase inicial, é usada uma função simples de ordenação, tais como modelos probabilísticos e a técnica de similaridade de cosseno (SALTON, 1971) (ROBERTSON; WALKER; HANCOCK-BEAULIEU, 1995), para selecionar um conjunto de documentos candidatos. Na segunda fase, uma função de ordenação baseada em aprendizado de máquina é aplicada somente no conjunto de documentos candidatos obtidos na primeira fase, gerando uma nova ordenação final que é apresentada ao usuário.

A vantagem desse método de classificação de duas etapas é a diminuição do conjunto de documentos a serem reclassificados. Além da redução no custo computacional, essa limitação no conjunto de documentos traz uma série de implicações para o sistema de processamento de consultas. Uma vez que o conjunto de documentos candidatos possui várias ordens de magnitude menor que o tamanho da coleção de documentos, as informações obtidas dos documentos podem caber na memória principal e, assim, eliminar os acessos ao disco. Juntamente a isso, a quantidade de memória para armazenar os *postings* após o processamento da primeira fase torna-se um fator constante proporcional a k documentos, em vez de depender do número de documentos da coleção. Logo, o uso da memória na segunda fase não cresce de acordo com o tamanho da coleção. Com isso, tais métodos de classificação de duas fases são de grande importância tanto para eficácia quanto para eficiência dos sistemas de busca atuais (MACDONALD et al., 2013).

Um alto custo computacional é necessário para as listas de termos com alta frequência, tornando-se impraticável, em termos de eficiência, aplicação de modelos de classificação em todos os documentos das listas invertidas. Em função dessa dificuldade, diversas técnicas de otimização foram apresentadas na literatura para percorrer as listas invertidas dos termos. Essas técnicas são agrupadas em duas abordagens: *term-at-a-time* (TAAT) e *document-at-a-time* (DAAT).

2.4.2 Abordagens de Classificação de Documentos

As estratégias da abordagem TAAT processam uma única lista invertida de um termo da consulta de cada vez e mantêm em memória uma lista de pontuações proporcional ao número de documentos relacionados aos termos da consulta. Por exemplo, se a lista invertida de um termo da consulta contém 1 milhão de *postings* e uma outra possui 500 mil *postings* distintos, então 1,5 milhão de pontuações devem estar armazenadas em memória após o processamento das duas listas invertidas. Além disso, na computação dessa abordagem, não há como prever com exatidão quais documentos terão maior similaridade com a consulta até avaliar todos os documentos das listas invertidas dos termos da consulta.

As técnicas da abordagem DAAT avaliam completamente um único documento antes de avaliar um próximo documento em potencial, ou seja, um único documento por vez. Dessa forma, a categoria DAAT percorre simultaneamente todas as listas invertidas dos termos da consulta. Além disso, DAAT pode manter somente as maiores pontuações na memória durante a computação. A desvantagem da DAAT é o alto custo associado ao acesso aleatório das listas invertidas, pois um mesmo documento pode estar em posições diferentes nas listas

e, consequentemente, em discos diferentes. Os algoritmos de poda (algoritmos de *prunning*) se destacam entre as estratégias DAAT, ao ordenar os documentos sem realizar a avaliação em todos os documentos (TURTLE; FLOOD, 1995) (MANNING et al., 2008) (CAMBAZOGLU; BAEZA-YATES, 2015) (JIANG; YANG, 2016).

Essas abordagens, TAAT e DAAT, ordenam os documentos em relevância e retornam somente um conjunto de k documentos mais relevantes, onde $k \in \mathbb{Z}$. Se uma dessas estratégias garante os k documentos mais relevantes, mas não garante a correta ordenação dos documentos por pontuação, ela é dita *set-safe*. Se a ordenação é garantida, então ela é dita *rank-score* (CAMBAZOGLU; BAEZA-YATES, 2015).

2.4.3 Algoritmos de Poda

Algoritmos de Poda (Algoritmos de *Pruning*) buscam melhorar a eficiência (latência) de busca de documentos mais relevantes a uma dada consulta nas listas invertidas. Esses algoritmos são divididos em dois grupos: poda estática (*static pruning*) e poda dinâmica (*dynamic pruning*, também chamada *early termination*).

Nos métodos de poda estática, várias informações que têm pouco efeito na eficácia das buscas no índice invertido são removidas e um novo índice é criado. Usualmente, esse novo índice é muito menor que o original. Os métodos de poda estática são aplicados durante a construção do índice e são independentes de consultas. Por ter essas características, eles são chamados de métodos offline. Em contrapartida, os algoritmos de poda dinâmica são aplicados durante o processamento de cada consulta requisitada com o objetivo de reduzir o custo computacional do problema de identificar os k documentos mais relevantes a consulta. Os dois grupos de poda podem ser complementares, ou seja, podem ser aplicados juntos para obter uma maior eficiência (CARMEL et al., 2001) (ALTINGOVDE; OZCAN; ULUSOY, 2012) (BÜTTCHER; CLARKE, 2006) (CARMEL et al., 2001).

Algoritmos de poda dinâmica têm sido propostos para melhorar a eficiência no processamento de consultas (TURTLE; FLOOD, 1995) (CARMEL et al., 2001) (BRODER et al., 2003) (BÜTTCHER; CLARKE, 2006) (JONASSEN; BRATSBERG, 2011) (ALTINGOVDE; OZCAN; ULUSOY, 2012) (SHAN et al., 2012) (DIMOPOULOS; NEPOMNYACHIY; SUEL, 2013a). A ideia comum dessas propostas é tentar reduzir o custo computacional, evitando o cálculo de similaridade dos documentos que não estarão na classificação dos resultados finais. Em particular, as propostas deste trabalho se baseiam em dois algoritmos fundamentais de poda dinâmica: WAND (BRODER et al., 2003) e MaxScore (TURTLE; FLOOD, 1995). A Subseção 2.4.3.1 e a Subseção 2.4.3.2 detalham os algoritmos MaxScore e WAND, respectiva-

mente.

2.4.3.1 Algoritmo Sequencial MaxScore

O algoritmo de poda MaxScore (TURTLE; FLOOD, 1995) processa os documentos das listas invertidas no estilo da abordagem DAAT, um documento em um dado instante. Para obter os k documentos mais relevantes, o algoritmo mantém os seguintes dados em memória: (1) a pontuação máxima de cada termo em sua lista invertida (*maxscore*) ; (2) um ponteiro que indica o próximo documento a ser avaliado (*nextDoc*) em cada lista de *posting*; (3) um conjunto de termos essenciais (*essentialTerms*); (4) uma estrutura de dados heap mínimo (*heap-min*) e (5) um limite inferior (*threshold*, o menor valor entre os elementos contidos no *heap-min*).

O algoritmo atua da seguinte forma. Inicialmente, os termos da consulta são ordenados em ordem crescente a suas pontuações máximas (*maxscores*). Essa ordenação é utilizada para fazer a soma acumalada dos *maxscores* na obtenção do conjunto dos termos essenciais, que terão suas listas invertidas percorridas pelo algoritmo. Todos os documentos a serem avaliados pelo algoritmo estão contidos nas listas invertidas dos termos desse conjunto de termos essenciais. Por outro lado, os documentos que estão presentes somente nos termos que não estão incluídos nesse conjunto são totalmente ignorados.

O conjunto dos termos essenciais é atualizado a cada iteração, sendo removidos ou adicionados os termos da consulta de acordo com o valor do *threshold*. Um termo t_i estará contido nesse conjunto se a soma acumulada, de forma crescente, dos *maxscores*, incluindo a do próprio t_i, for maior que o valor do *threshold*. Dessa forma, o algoritmo atualiza, de maneira indireta, os possíveis documentos a serem avaliados até o certo momento. Entre esses documentos selecionados, o algoritmo sempre seleciona o menor docID e realiza a avaliação. O algoritmo finaliza quando todos os documentos do conjunto dos essenciais forem analisados ou o conjunto dos essenciais encontra-se vazio (JONASSEN; BRATSBERG, 2011) (SHAN et al., 2012).

O Algoritmo 1 apresenta o algoritmo MaxScore. As listas de termos e o próximo documento, em cada lista de *posting*, são inicializados nas linhas de 1 a 4. Na linha 5, os termos são colocados em ordem crescente de acordo com os *maxscores* dos termos e o resultado é colocado na lista *essentialTerms*. Posteriormente, a lista *essentialTerms* é atualizada pela função *UpdateEssentialTerms* que realiza o processo a partir do valor inicial do *threshold*. Se a consulta de entrada é do tipo *OR*, então o *threshold* será zero. Para consulta *AND*, o *threshold* terá a soma dos *maxscore* dos termos da consulta de entrada. O termo contido na posição *essentialTerms[i]* somente é removido da lista *essentialTerms* pela função *UpdateEssentialTerms* em duas condições. A primeira ocorre quando $\sum(essentialTerms[1..i].maxscore) < \theta$ e a segunda,

quando não há documentos a serem analisados do termo *essentialTerms[i]*. Assim, o algoritmo mantém a lista *essentialTerms* atualizada somente com os termos que possuem contribuição suficiente para terem prováveis documentos na lista dos top-k documentos mais relevantes.

O menor docID das listas invertidas contidas na estrutura de dados *essentialTerms* é obtido e avaliado completamente - linha 8 a 12. Se a pontuação alcançada for maior que o *threshold*, o documento é inserido na lista dos k documentos mais relevantes e o *threshold* é atualizado, linha 13. Os ponteiros dos próximos documentos a serem avaliados (*nextDoc*) são ajustados para o próximo docID maior que o último docID avaliado. Após a avaliação de um documento, a lista *essentialTerms* é atualizada na linha 19, pois o valor do *threshold* pode ter sido modificado na linha 13. No final do algoritmo, quando todos os documentos identificados estiverem na lista dos documentos mais relevantes (*topkDocs*), uma ordenação é realizada nessa lista, linha 22, antes de retorná-los.

Algoritmo 1 Algoritmo Sequencial MaxScore

Input: [1] os termos da consulta $< t_1, ..., t_n >$; [2] o número de documentos recuperados k; [3] *threshold* θ;
Output: Lista dos top-k documentos $result[1..k]$;

```
1:  for i ← 1; i ≤ n; i++ do
2:      terms[i].term ← t_i
3:      terms[i].nextDoc ← NextDoc(t_i, 0)
4:  end for
5:  essentialTerms ← Sort(terms)
6:  UpdateEssentialTerms(essentialTerms, terms, θ)
7:  while essentialTerms ≠ ∅ and essentialTerms[0].nextDoc ≠ NoMoreDocs do
8:      for i ← 1; i ≤ n; i++ do
9:          if essentialTerms[0].nextDoc = terms[i].nextDoc then
10:             score ← score + FullScore(terms[i])
11:         end if
12:     end for
13:     θ ← ManagerTopkDocs(topkDocs, score, essentialTerms[0].nextDoc)
14:     for i ← 1; i ≤ n; i++ do
15:         if terms[i].nextDoc ≤ essentialTerms[0].nextDoc then
16:             terms[i].nextDoc ← NextDoc(t_i, essentialTerms[0].nextDoc + 1)
17:         end if
18:     end for
19:     UpdateEssentialTerms(essentialTerms, terms, θ)
20:     score ← 0
21: end while
22: result ← Sort(topkDocs)
```

2.4.3.2 Algoritmo Sequencial WAND

De forma semelhante ao *Maxscore*, o algoritmo WAND proposto em (BRODER et al., 2003) processa as listas invertidas das consultas ao estilo da abordagem DAAT. A posição das listas dos termos da consulta são mantidas em ordem crescente ao próximo docID a ser avaliado de cada lista. Assim, cada vez que um docID é avaliado, a ordem das listas é verificada para saber se foi alterada.

O algoritmo WAND utiliza as maiores pontuações (*upper bound*) dos termos da consulta em suas listas invertidas para efetuar um processamento de duas etapas de avaliação. Na primeira

etapa, uma avaliação simples e parcial é realizada para selecionar um documento candidato e ignorar documentos com baixa contribuição nos termos da consulta. Após essa seleção, aplica-se um modelo similaridade no documento candidato e compara-se a pontuação obtida com as pontuações dos elementos contidos na lista dos k documentos mais relevantes na segunda etapa da avaliação. Por ter uma avaliação preliminar simples que apresenta um baixo custo computacional comparado com a aplicação de um modelo de similaridade, o algoritmo evita realizar o cálculo do modelo de similaridade em todos os documentos da consulta e, assim, alcança um ganho de desempenho.

A estrutura de dados indicada no trabalho original do WAND (BRODER et al., 2003) para manter os k documentos mais similares à consulta é a estrutura heap mínimo (*heap-min*). O menor elemento da lista dos k documentos mais relevantes, também considerada como *threshold*, é utilizada para realizar a seleção dos documentos candidatos obtidos na primeira fase da avaliação. Essa avaliação preliminar é efetivada por meio de um processo de seleção de documentos candidatos.

Seguindo a ordenação crescente dos próximos docIDs a serem avaliados, o primeiro termo cuja a soma acumulada dos *upper bounds* excedem o *threshold* é selecionado como *termo pivô*. O próximo documento a ser avaliado na lista invertida desse termo é considerado como documento candidato e chamado de *documento pivô*, ou simplemente pivô. Então, após a seleção do pivô, verifica-se se todos os termos ordenados e anteriores ao termo pivô possuem a ocorrência do docID representada pelo pivô. Se confirmada essa condição, o documento é passado para a segunda fase do processamento, na qual uma avaliação completa é realizada. A pontuação completa obtida desse documento é comparada com o *threshold* para ser inserida na lista dos k documentos mais relevantes. Por último, o algoritmo atualiza os próximos documentos a serem avaliados e continua com o processo até todos os documentos serem avaliados ou não ter mais o pivoteamento.

O Algoritmo 2 apresenta o processamento original do algoritmo WAND. As listas de termos e o próximo documento a ser avaliado, em cada lista de *posting*, são inicializados nas linhas de 1 a 4. A ordem dos termos é alterada, pelas chamadas da função *Sort* nas linhas 5 e 27, de forma crescente ao próximo documento a ser avaliado em cada lista invertida. O documento pivô é mantido em cada iteração do laço de repetição da linha 7 pelas chamadas da função *FindPivotTerm* nas linhas 6 e 28.

O documento candidato selecionado é avaliado completamente por um modelo de similaridade entre as linhas 9 e 13, se possuir ocorrências em todos os termos anteriores ao termo pivô na lista ordenada de termos. Essa verificação é realizada na condição da linha 8. Caso não

tenha as ocorrências, os próximos docIDs a serem avaliados são atualizados e um novo pivô é selecionado. Caso uma avaliação completa seja realizada no docID pivô, então a pontuação é passada para a função *ManagerTopkDocs*, linha 14, com o intuito de realizar o gerenciamento do documento na lista de k documentos.

Algorithm 2 Algoritmo Sequencial WAND

Input: [1] os termos da consulta $< t_1, ..., t_n >$; [2] o número de documentos recuperados k; [3] *threshold* θ
Output: Lista dos top-k documentos $result[1..k]$;

```
1:  for i ← 1; i ≤ n; i++ do
2:      terms[i].term ← t_i
3:      terms[i].nextDoc ← NextDoc(t_i, 0)
4:  end for
5:  Sort(terms)
6:  pTerm ← FindPivotTerm(terms, θ)
7:  while pTerm.nextDoc ≠ NoMoreDocs do
8:      if pTerm.nextDoc = terms[0].nextDoc then
9:          for i ← 1; i ≤ n; i++ do
10:             if pTerm.nextDoc = terms[i].nextDoc then
11:                 score ← score + FullScore(terms[i])
12:             end if
13:         end for
14:         θ ← ManagerTopkDocs(topkDocs, score, pTerm.nextDoc)
15:         for i ← 1; i ≤ n; i++ do
16:             if pTerm.nextDoc = terms[i].nextDoc then
17:                 terms[i].nextDoc ← NextDoc(t_i, pTerm.nextDoc + 1)
18:             end if
19:         end for
20:     else
21:         for i ← 1; i ≤ n; i++ do
22:             if terms[i].nextDoc < pTerm.nextDoc then
23:                 terms[i].nextDoc ← NextDoc(t_i, pTerm.nextDoc)
24:             end if
25:         end for
26:     end if
27:     Sort(terms)
28:     pTerm ← FindPivotTerm(terms, θ)
29:     score ← 0
30: end while
31: result ← Sort(topkDocs)
```

O trabalho proposto em (DING; SUEL, 2011), chamado *Block-Max WAND (BMW)*, baseou-se no algoritmo WAND para obter os k documentos mais relevantes. O algoritmo utiliza métodos de compactação no índice invertido, por exemplo, o método *New-PFD* (YAN; DING; SUEL, 2009). As listas invertidas são divididas em blocos de 64 ou 128 docIDs e cada bloco pode ser descompactado separadamente. Dessa forma, o *BMW* estabelece a criação e utilização de dados adicionais que são acoplados ao índice invertido (*block-max index*). Esses dados referem-se às maiores pontuações de documentos dentro de cada bloco, ou seja, o *upper bound* (UB) de cada bloco, ao invés de um único valor por termo. Na fase de escolha do termo pivô, o algoritmo faz o uso desse valor local ao bloco atual ao invés de um valor global. Essa estratégia faz com que o algoritmo alcance mais eficiência que a versão do WAND ao saltar um bloco (conjunto de documentos) em um único passo.

2.5 Considerações Finais do Capítulo

Este capítulo apresentou os principais conceitos da recuperação de informação, relacionados aos algoritmos escolhidos para serem analisados neste trabalho, ou seja, algoritmos de poda dinâmica DAAT. Esses algoritmos foram escolhidos por se adaptarem bem ao crescimento das coleções de documentos e por serem passíveis de melhorias na eficiência do processamento de consultas. As principais características desses algoritmos, o WAND e o MaxScore, foram destacadas. Essas soluções são eficientes no processamento de consultas sem que haja perdas na acurácia dos resultados. Além disso, essas soluções são bases para diversas estratégias de otimização (ROJAS; GIL-COSTA; MARIN, 2013) (CARMEL et al., 2001) (DING; SUEL, 2011) (BÜTTCHER; CLARKE, 2006) (JONASSEN; BRATSBERG, 2011) (ALTINGOVDE; OZCAN; ULUSOY, 2012) (SHAN et al., 2012) (DIMOPOULOS; NEPOMNYACHIY; SUEL, 2013a).

Capítulo 3

ARQUITETURAS PARALELAS

O consumo de energia crescente e as altas temperaturas das operações dos sistemas computacionais tornaram insustentável o aumento de desempenho das arquiteturas de um único processador que, por mais de 40 anos, dominaram os sistemas computacionais. Esse cenário tem impulsionado a indústria a procurar novas soluções. A integração de várias unidades de processamento com baixa frequência e consumo de energia reduzido em um único chip se tornou uma solução atraente ao permitir um contínuo aprimoramento de desempenho dos processadores.

A disseminação dessa solução teve um impacto direto no desempenho dos softwares. A adaptação de um programa para fazer melhor uso do paralelismo interno ao nível de instrução de um processador era realizada pelos compiladores e os desenvolvedores de softwares não precisavam se preocupar. Contudo, a exploração do paralelismo pelo compilador em vários núcleos de um processador é uma tarefa difícil e muitas vezes não eficiente, pois é necessário identificar fluxos independentes de instruções de um programa que podem ser executados em paralelo. Além disso, antes, os avanços na tecnologia dos processadores significavam avanços na velocidade do clock, de modo que havia um aumento automático na aceleração dos softwares. Agora, no entanto, essa afirmação não é mais verdadeira, pois os avanços na tecnologia induzem a um maior paralelismo e não somente ao aumento de velocidade do clock. Portanto, para fazer o uso eficiente da computação nessas novas arquiteturas, se faz necessária a exploração de técnicas de paralelismo de forma eficaz, ou seja, o uso de vários processadores para trabalhar em uma única tarefa (HERLIHY; SHAVIT, 2011).

A partir dessa solução de integrar vários núcleos em um único chip, duas abordagens de arquitetura se destacaram: *multicore* e *manycore*. A abordagem *multicore* integra até algumas dezenas de núcleos altamente complexos com grandes caches em um único processador. Embora essa arquitetura vise a aumentar o desempenho dos núcleos para computação com baixa latência, buscando manter o desempenho de programas sequenciais, ela permite a execução de

múltiplos fluxos de instruções independentes. A programação paralela nessas arquiteturas é um desafio por conter núcleos de processamento assíncronos, em que as atividades podem ser atrasadas sem aviso por interrupções, execução antecipada, falhas de cache e outros eventos. Para facilitar a programação, diversas APIs oferecem recursos que expõem regiões paralelas no código para os compiladores na tentativa de adaptar o programa na arquitetura paralela dos *multicores*, de modo que aproveite ao máximo o paralelismo interno da plataforma. A API de memória compartilhada OpenMP se destaca ao facilitar a descrição ao compilador de como o trabalho deve ser compartilhado entre as *threads* que serão executadas em diferentes núcleos de processamento e como solicitar o acesso a dados compartilhados.

A segunda abordagem, *manycore*, comumente usa milhares de núcleos simples com caches muito pequenas e interfaces com alta largura de banda para vários blocos de memória. Esse tipo de arquitetura possui a característica de sacrificar o desempenho sequencial de um único núcleo de processamento, com a orientação de aumentar a vazão de fluxos de instruções, beneficiando aplicações paralelas. A principal arquitetura que representa essa abordagem é a Unidade de Processamento Gráfico (GPU). Apersar de que, historicamente, as GPUs se apresentaram como aceleradores gráficos, recentemente elas evoluíram para processadores de propósito geral, que têm impulsionado pesquisas em diversas áreas (GREEN; MCCOLL; BADER, 2012; LIMA et al., 2015; BUSATO; BOMBIERI, 2015; MATSUMOTO; NAKASATO; SEDUKHIN, 2011; CHEN; HUO; AGRAWAL, 2012). Esse interesse de aplicações não gráficas em GPUs se deve essencialmente por duas considerações, ao custo/benefício e à evolução da velocidade.

O poder computacional da GPU tem excedido o das CPUs. Por exemplo, a NVIDIA Titan Xp possui o desempenho teórico de 12 TFLOPS, enquanto a CPU Intel i7-5820K alcança um desempenho de 172,40 GFLOPS. Ao mesmo tempo, o mercado de jogos gera vendas de alto volume de placas gráficas, o que mantém o preço baixo das GPUs em comparação aos outros hardwares especializados. Além disso, impulsionado pelo mercado de jogos, o desempenho da GPU tem avançado mais rapidamente do que a taxa de crescimento da CPU. Essa evolução do desempenho das GPUs se deve principalmente ao paralelismo explícito oferecido pelo hardware gráfico.

A arquitetura CUDA NVIDIA tem se destacado entre as GPUs, sendo amplamente utilizada por oferecer um hardware de alto poder computacional e um conjunto de APIs que abrangem uma ampla gama de recursos no que diz respeito à decomposição de problemas e à expressão de paralelismo. A NVIDIA usa um termo especial, *compute capability*, para descrever as versões dos hardwares da GPU. Atualmente, a *compute capability* está na versão 7.5 com a arquitetura nomeada de Turing que possui, além de milhares de processadores de precisão simples,

processadores especializados na técnica de *ray-tracing* (*RTX Cores*). Isso mostra o quanto tem evoluído a arquitetura CUDA. Por isso, este capítulo foca nos detalhes da arquitetura CUDA GPU.

3.1 Unidade de Processamento Gráfico (GPU)

As Unidades de Processamento Gráfico (GPUs) são arquiteturas massivamente paralelas especializadas e projetadas, originalmente, como coprocessadores cujo propósito era renderizar gráficos. No entanto, desde de 2006, essas unidades se tornaram aceleradores poderosos para computação de propósito geral (GPGPU).

Uma visão de arquitetura abstrata de uma GPU é ilustrada na Figura 3.1. A arquitetura básica da GPU consiste em um conjunto de Multiprocessadores de Stream (SMs), cada um contendo vários Processadores de Stream (SPs). Embora todos os SPs (*thin core*, granularidade fina) dentro de um SM (*fat core*, granularidade grossa) executem as mesmas instruções, essa execução é realizada em diferentes conjuntos de dados, de acordo com o modelo SIMD (*Single Instruction, Multiple Data*).

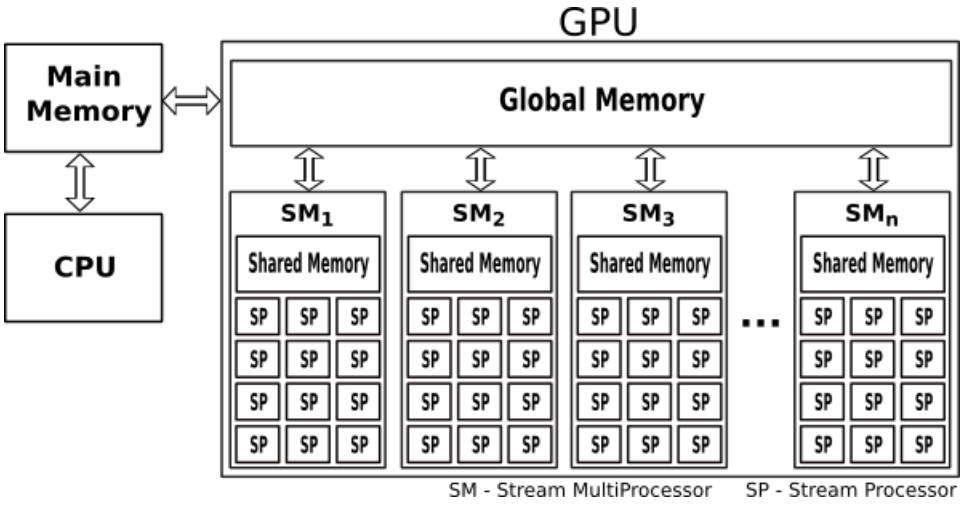

Figura 3.1: Visão geral de uma arquitetura GPU.

A quantidade de SPs e o número de SMs em uma GPU diferem de um modelo para outro. A GPU suporta milhares de *threads* concorrentes leves e, ao contrário das *threads* da CPU, a sobrecarga de criação e trocas de contexto de *threads* é insignificante quando comparado com a CPU. As *threads* que são executadas em cada SM são organizadas em grupos que compartilham os mesmos recursos de computação, por exemplo, registradores e memória compartilhada. Um grupo de *threads* é dividido em várias unidades escalonáveis, chamadas de *warp*, que são

escalonadas dinamicamente dentro do SM.

Devido à natureza SIMD, se uma *thread* de uma *warp* espera por uma operação de memória, toda *warp* irá esperar até que a operação seja efetuada. Neste caso, o SM seleciona outra *warp* que esteja pronta para ser executada e faz a troca de contexto de *warp*, ou seja, retira todas as *threads* da *warp* ociosa de execução e adiciona uma *warp* para execução. Esse técnica é chamada de latência escondida de memória.

A memória global da GPU é uma memória do tipo *off-chip* com alta largura de banda e alta latência. Sua capacidade é medida em gibabytes. Para que a latência escondida de memória seja efetiva, é importante ter mais *threads* agrupadas do que o número de SPs dentro de um SM e ter acessos a endereços consecutivos da memória global pelas *threads* de uma *warp* que possam ser facilmente aglutinadas.

Um segundo tipo de memória oferecida pela GPU é a memória compartilhada (*shared memory*). Diferente da memória global, caracterizada pelo tipo *off-chip*, a memória compartilhada é *on-chip*, localizada dentro dos SMs, fornecendo rápido acesso aos SPs. Dessa forma, a memória compartilhada de um SM é acessível por todos os SPs do SM. O tamanho é pequeno, medido em kilobytes, e possui baixa latência. A utilização pode ser feita como cache controlada pelo desenvolvedor ou por forma automática pela API CUDA (software).

A CPU comunica-se com a GPU e vice-versa através de uma rede de interconexão. Embora essa conexão comumente seja implementada pela tecnologia *PCIExpress*, novas tecnologias têm apresentado maior eficiência na transferência de dados, como por exemplo, a tecnologia *NVlink* (FOLEY, 2014). Essa tecnologia expôs uma velocidade de 19.2 GB/s de transferência de dados entre CPU e GPU, aproximando da velocidade de comunicação da CPU com a memória principal, eliminando, dessa forma, o principal gargalo de comunicação encontrado na programação dessas arquiteturas.

O modelo de programação da GPU requer que a parte da aplicação que exige o uso intensivo de computação seja acelerada pela GPU. Essa parte é mapeada na GPU através de funções especiais, nomeadas de funções *kernels*. Dessa maneira, o fluxo geral de um programa segue algumas etapas. Primeiro, o programa em execução na CPU aloca memória na memória global da GPU e copia os dados que estão localizados na memória principal para essa área alocada. Então, o código da GPU (função *kernel*) pode ser iniciado na GPU (processo intitulado como lançamento de *kernel*). O código do *kernel* é executado em paralelo na GPU e os resultados são copiados de volta para a memória principal da CPU. Uma nova iteração pode ocorrer ou o programa da CPU pode desalocar a memória alocada anteriormente na GPU e finalizar.

A programação na GPU expõe o paralelismo através das funções *kernels*. As instruções dessas funções são paralelizadas no estilo do paralelismo de dados SPMD (*Single Program Multiple Data*). Durante a implementação, o desenvolvedor pode configurar o número de *threads* a ser usado em cada função *kernel*. Todas as *threads* de um *kernel* são organizadas em grupos, chamados de bloco de *threads*. Por sua vez, esses blocos são organizados em uma estrutura maior, nomeada de grade de *threads* (*thread grid*), como mostrado na Figura 3.2. Quando um *kernel* é lançado, os blocos de *threads* dentro da grade são distribuídos nos SMs ociosos, enquando as *threads* de um bloco são mapeadas para os SPs do SM designado.

As *threads* de um bloco são divididas em *warps*, uma unidade escalonável usada pelos SMs, deixando que a GPU decida em qual ordem e quando executar cada *warp*. *Threads* que pertencem a blocos diferentes não podem se comunicar explicitamente e precisam utilizar a memória global para compartilhar seus resultados. *Threads* de um bloco são executadas pelos SPs de um único SM e podem se comunicar através da memória compartilhada do SM. Além disso, cada *thread* dentro de um bloco tem seus próprios registradores e memória local privada e usa um índice global do bloco de *threads* e um índice de *thread* local dentro de um bloco de *threads*, para identificar exclusivamente seus dados.

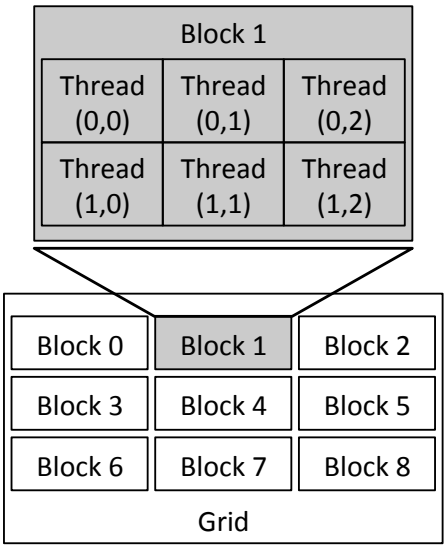

Figura 3.2: Organização das *threads* na GPU.

3.2 Modelos Paralelos

Plataformas heterogêneas modernas de computação de alto desempenho têm buscado apoio nos processadores *manycore*. Contudo, a utilização eficiente da computação e comunicação nessas plataformas transformou-se em um fator de impacto para desenvolvedores de software, pois a obtenção de desempenho nessas arquiteturas depende muitas vezes dos detalhes da própria arquitetura.

O uso de modelos computacionais paralelos permite ter uma previsão independente da plataforma tanto do custo de tempo quanto do custo de comunicação de um algoritmo paralelo. Para tanto, um modelo computacional deve apresentar uma certa estabilidade de abstração e detalhe da arquitetura, de modo a refletir o comportamento real desses algoritmos paralelos (MEYER; SANDERS; SIBEYN, 2003). Porém, um dificuldade fundamental ao uso generalizado de máquinas paralelas para computação de propósito geral é a falta de um modelo padrão amplamente aceito de computação paralela que não deve interferir no processo de projetar e implementar algoritmos.

As máquinas sequenciais não alteram muito de uma arquitetura para outra, já que todas estão em conformidade com um modelo de von Neumann (também conhecido como modelo RAM - Máquina de Acesso Aleatório). Cada máquina sequencial pode ter seus próprios recursos para melhorar o desempenho, mas ainda atende a esse modelo. Enquanto isso, os programas paralelos desenvolvidos em uma máquina paralela geralmente não apresentam portabilidade para serem empregados em outras máquinas paralelas, assim exigem grandes modificações antes que possam ser eficientemente empregados.

Este trabalho não tem o foco em um modelo específico de computação paralela, contudo esta seção irá apresentar os principais modelos paralelos com o intuito de facilitar o entendimento das propostas deste trabalho.

3.2.1 Parallel Random Acess Machine (PRAM)

O modelo teórico utilizado para computadores sequenciais (SISD, "*Single Instrution, Sigle Data*") é conhecido como máquina de acesso aleatório (RAM). A versão paralela do modelo RAM constitui um modelo abstrato da classe de processadores paralelos de memória global, chamada de modelo PRAM (KARP; RAMACHANDRAN, 1991). Esse modelo é básico para análise teórica da computação paralela por ignorar os detalhes da rede de interconexão processador-memória. Além disso, a abstração considera que cada processador pode acessar qualquer localização de memória em cada ciclo de máquina, independentemente do que outros

processadores estão fazendo (SKILLICORN; TALIA, 1998).

A Figura 3.3 ilustra uma visão abstrata da organização da arquitetura PRAM, em que um número p de processadores têm acesso a uma memória global com m posições. O modelo PRAM é considerado altamente teórico, em vista do acesso direto e independente a cada localização de memória permitida para cada processador em único ciclo de máquina. Dessa forma, o uso desse modelo permite concentra-se na realização da computação, sem qualquer preocupação com problemas relacionados aos acessos à memória. Na prática, o acesso a memória global teria que ser realizado por uma rede de interconexão. Assim, o acesso seria limitado pela largura de banda da rede de interconexão, gastando diferentes tempos ao acessar localizações diferentes.

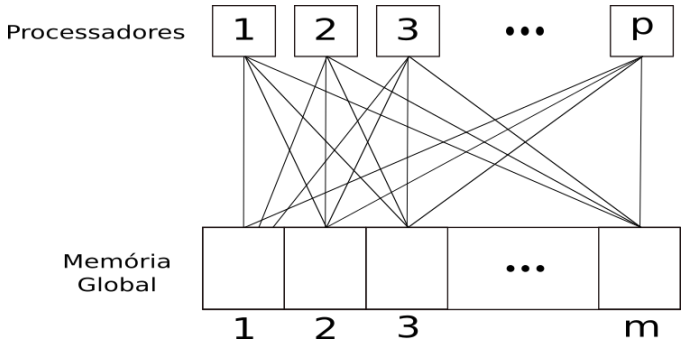

Figura 3.3: Visão conceitual da máquina paralela de acesso aleatório (PRAM).

Nesse modelo há vários processadores trabalhando em paralelo, a operação de sincronização da PRAM pode resultar em acessos para uma mesma localidade na memória global. A realização dessas concorrências, leitura e escrita, divide o modelo PRAM nas seguintes variantes (SKILLICORN, 2005):

1. Leitura Exclusiva e Escrita Exclusiva (EREW): não permite qualquer tipo de concorrência em uma mesma localidade;

2. Leitura Concorrente e Escrita Exclusiva (CREW): aceita somente acessos concorrentes de leitura. Assim, todos os processadores participantes em uma mesma localidade obtêm o mesmo valor;

3. Leitura Exclusiva e Escrita Concorrente (ERCW): permite concorrência de acessos somente de escrita;

4. Leitura Concorrente e Escrita Concorrente (CRCW): a concorrência dos dois tipos de acesso à memória (leitura e escrita) é permitida. Enquanto na leitura, um mesmo valor é passado para todos os processadores, na escrita há várias formas de tratar a concorrência.

A maneira mais simples é permitir a escrita de um valor se todos os processadores estão tentando escrever um mesmo valor. De forma arbitrária, um valor pode ser escolhido dentre os acessos para ser armazenado. Em um modo prioridade, o valor escolhido pertence ao processador com a maior prioridade.

3.2.2 Modelo *Bulk Synchronous Parallel* (BSP)

A máquina abstrada do modelo *Bulk Synchronous Parallel* (BSP) (VALIANT, 1989) consiste em uma coleção de processadores com uma memória local, conectados por uma rede de interconexão, como mostrado na Figura 3.4. Esse modelo presume que a rede de interconexão seja capaz de entregar mensagens de ponto a ponto com um custo uniforme. Isto significa que o custo de acessar a memória de qualquer processador é independente da localidade do processador. Desse modo, leva-se a rede de interconexão ser considerada como uma caixa preta, onde a conectividade da rede é escondida no seu interior.

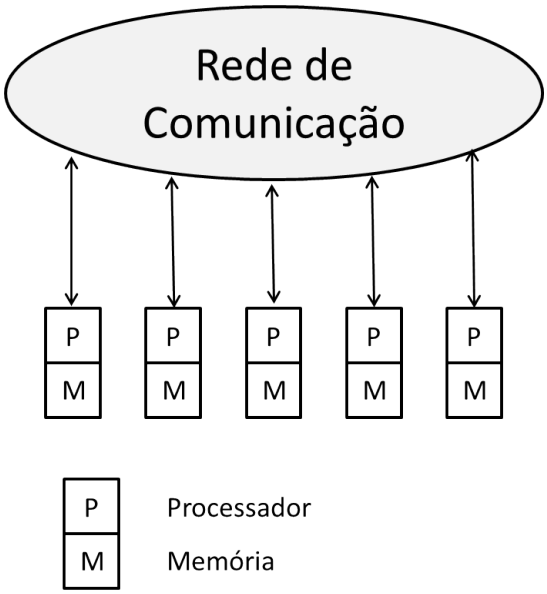

Figura 3.4: Modelo Bulk Synchronous Parallel (BSP)

A execução de um programa no modelo BSP segue como uma sequência de superpassos. Dentro de um superpasso, um processador pode atualizar seus valores locais, no entanto, as memórias remotas não são atualizadas até que o fim do superpasso seja alcançado. Nesse ponto, os processadores são sincronizados, as memórias são atualizadas e a execução continua no próximo superpasso. Dessa forma, ao ler um valor remoto, os processadores recebem o valor do superpasso anterior e, ao gravar valores remotos, não há confirmação até o final do superpasso.

O desempenho da comunicação do modelo BSP é medido por dois parâmetros: L e g. O

parâmetro L implica na latência da memória, isto é, o número mínimo de unidades de tempo entre superpassos sucessivos. Após cada período de L unidades de tempo, uma verificação global é feita para determinar se o superpasso foi completado por todos os componentes. Se tiver completado, a máquina prossegue para o próximo superpasso. Caso contrário, o próximo período de L unidades é alocado ao superpasso inacabado. O parâmetro g define a largura de banda básica da rede de interconexão, quando está em uso contínuo. Esses parâmetros são determinados experimentalmente por cada computador paralelo (MAGGS; MATHESON; TARJAN, 1995).

Essa abstração das máquinas paralelas leva esse modelo a ser altamente escalável e fácil de programação, pois torna-se previsível o tempo de finalização dos superpassos de um programa no modelo BSP. O cálculo do tempo de um superpasso segue da seguinte maneira. Se o tempo de computação é w e o número máximo de mensagens enviadas ou recebidas por um processador é h, então o tempo de uma superetapa é dado por:$w + hg + L$. Com essas características, o BSP é considerado uma generalização do modelo PRAM. Quando a arquitetura BSP tem o valor de g muito pequeno ($g = 1$), ela torna-se uma PRAM (MAGGS; MATHESON; TARJAN, 1995) (SKILLICORN; TALIA, 1998).

3.2.3 Modelo BSP/CGM

O modelo BSP considera que a computação em uma máquina com p processadores é realizada por cada processador de forma que um problema de tamanho n é dividido em granularidade grossa (*coarse grained*), ou seja, $n \gg p$. O modelo *Coarse-Grained Multicomputer (CGM)* é um caso especial do modelo BSP, em que há uma adição estritamente na comunicação, transformando as comunicações em granularidade grossa com tamanho $O(\frac{n}{p})$, onde $O(\frac{n}{p}) \gg 1$.

Uma máquina paralela é considerada no modelo BSP/CGM um conjunto de p processadores, cada qual com uma memória local de tamanho $(\frac{n}{p})$, interconectados por uma rede arbitrária de comunicação ou por uma memória compartilhada.

O modelo BSP/CGM é adequado para projetar algoritmos paralelos independentes da arquitetura. O algoritmo consiste de uma sequência de superpassos ou rodadas. Um superpasso contém ou um número de etapas de computação ou um número de etapas de comunicação, acompanhado por uma barreira de sincronização global. Nesse modelo, o custo de comunicação é dado pelo número de superpassos requeridos. Cada rodada de comunicação consiste em enviar $O(\frac{n}{p})$ dados e receber $O(\frac{n}{p})$ dados. Encontrar um algoritmo ótimo no modelo BSP/CGM requer a redução do número de rodadas de comunicação. A Figura 3.5 mostra um conjunto de pares de processador-memória envolvido em superpassos de computação e comunicação sepa-

rados por barreiras de sincronização (CHEETHAM et al., 2003) (ALVES; CÁCERES; DEHNE, 2002) (DEHNE; FABRI; RAU-CHAPLIN, 1996) (VALIANT, 1990).

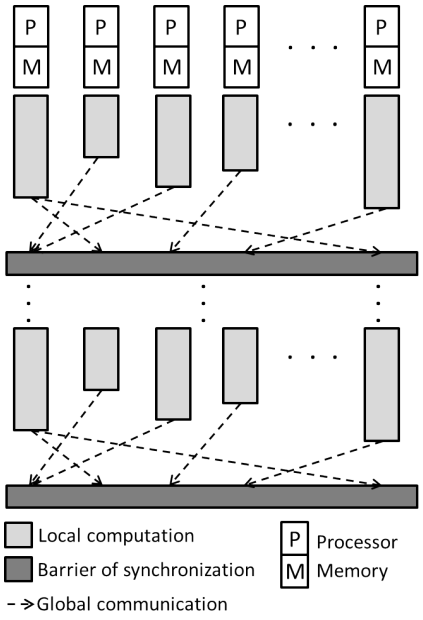

Figura 3.5: Execução de um programa no modelo BSP/CGM.

3.3 Considerações Finais do Capítulo

Este capítulo detalhou as principais características da arquitetura CUDA GPU e os principais modelos paralelos. A arquitetura CUDA GPU foi escolhida para ser a plataforma principal deste trabalho por apresentar um hardware com alto poder computacional, de baixo custo, totalmente programável e massivamente paralelo. Além disso, a plataforma oferece programação de paralelismo de granularidade fina que permite oportunidades de paralelização ao nível de algoritmo.

Capítulo 4

PROPOSTA DE PARALELIZAÇÃO DO PROCESSAMENTO DE CONSULTA EM GPUS

Este trabalho discute o uso da computação paralela na GPU para identificação de documentos mais relevantes a uma consulta através de soluções paralelas. A eficiência neste processo é o requisito principal a ser ponderado na análise do comportamento das novas soluções de paralelização aqui oferecidas. Ademais, algoritmos sequenciais fundamentais que se adaptam bem ao tamanho da coleção de documentos são explorados nestes algoritmos paralelos.

Neste capítulo, as soluções são condicionadas à latência do processamento de consulta. Para isso, são detalhadas duas novas estratégias de particiomento de documentos entre os processadores paralelos que são bem adequadas à abordagem DAAT. Uma paralelização de algoritmo que explora o particionamento dessas estratégias é apresentada neste capítulo. Duas propostas de versões paralelas dos algoritmos fundamentais de classificação de documentos aqui oferecidas são utilizadas: algoritmo WAND (BRODER et al., 2003) e algoritmo MaxScore (TURTLE; FLOOD, 1995).

4.1 Escopo do Trabalho

A diversidade de abordagens de implementação existentes em recuperação de informação na Web dificulta qualquer comparação de desempenho de consultas (TURTLE; FLOOD, 1995). A fim de reduzir essa dificuldade, esta seção destaca as considerações em recuperação de informação e em computação paralela que definem o escopo das estratégias e das propostas aqui oferecidas.

4.1.1 Processamento de Consultas em um Único Nó

As atuais máquinas de busca comerciais utilizam clusters com centenas de servidores. Esses servidores são organizados ao estilo do paradigma mestre-escravo (ROJAS; GIL-COSTA; MARIN, 2013), sendo que cada qual é responsável por buscar informações em um subconjunto dos documentos.

Neste presente trabalho, o foco está no processamento dentro de cada máquina, pois ao maximizar o rendimento local de cada máquina, é maximizado o rendimento global. Essa consideração impele à ponderação das propostas a serem utilizadas sobre qualquer técnica de balanceamento de carga, por exemplo, uma simples técnica de *round-robin* ou uma mais avançada que considera o balanceamento de carga de cada nó.

Os documentos podem ser divididos em dois cenários a partir de ambientes mestre-escravo: *(1)* os documentos distribuídos entre os nós de processamento e, assim, cada qual irá ter um índice invertido distinto; e *(2)* o índice invertido compartilhado de tal forma que o vocabulário é segmentado. Contudo, iremos considerar o primeiro cenário, pois é o mais utilizado nas máquinas de busca comerciais e na literatura (MANNING et al., 2008).

4.1.2 Estratégia de Avaliação: *Document-at-a-time* - DAAT

Atualmente, as listas invertidas dos termos da consulta são percorridas pelos algoritmos da categoria DAAT na maioria das máquinas de busca comerciais (BÜTTCHER; CLARKE; CORMACK, 2016; CAMBAZOGLU; BAEZA-YATES, 2015). Isso se deve, principalmente, por dois fatores:

1. Ao atravessar as listas de todos os termos das consultas ao mesmo tempo, as estratégias DAAT ficam passíveis de melhorias de eficiência em relação a outras estratégias;

2. Procurando identificar somente os k documentos mais relevantes referentes à consulta, mantêm-se no máximo os k documentos com as maiores pontuações até o momento da computação em memória.

Uma classe importante de algoritmos de otimização nas estratégias DAAT são os dos algoritmos sequenciais de poda dinâmica. Esses algoritmos alcançam um rápido processamento de consulta ao evitar o cálculo da similaridade de documentos com baixo potencial para estar na classificação dos top-k documentos à consulta. Em particular, as propostas deste trabalho focam em dois algoritmos que fazem parte do estado da arte no processamento sequencial de

consultas com poda dinâmica: *WAND* (BRODER et al., 2003) e *MaxScore* (TURTLE; FLOOD, 1995).

O algoritmo WAND armazena a maior pontuação de cada lista de *posting* dos termos, faz uma seleção a cada iteração de um documento com alta aproximidade com os termos da consulta e realiza duas avaliações, uma parcial rápida e outra completa. Isso lhe permite evitar aplicações desnecessárias de modelos de similaridade em muitos documentos que seriam avaliados por um algoritmo exaustivo. Diversos trabalhos propuseram aprimoramentos baseados nesse algoritmo (DING; SUEL, 2011) (DIMOPOULOS; NEPOMNYACHIY; SUEL, 2013b) (ROSSI et al., 2013) (SHAN et al., 2012).

O segundo algoritmo, MaxScore, divide as listas de *postings* dos termos da consulta em dois conjuntos, essenciais e não essenciais, a partir de um limiar atualizado dinamicamente de acordo com o aumento do número de documentos avaliados. Os documentos candidatos, aqueles com alto potencial de estarem no resultado final dos top-k documentos, são obtidos somente das listas essenciais e os documentos restantes são ignorados. Assim, o algoritmo obtém eficência no processamento de consulta. Há várias versões do MaxScore baseado em ambas as estratégias de percorrer as listas invertidas, DAAT e TAAT (STROHMAN; TURTLE; CROFT, 2005) (ZHU et al., 2008) (JONASSEN; BRATSBERG, 2011).

4.1.3 Abstração da Arquitetura Manycore da GPU

No nível de processamento, considera-se a organização das unidades de processamento na GPU, onde há uma hierarquia de processadores. Os primeiros processadores, considerados como *fat cores*, Multiprocessadores (SMs), possuem as características de processadores de granularidade grossa (*coarse-grained*) e isso os leva a serem assíncronos, podendo executar fluxo de instruções diferentes (paralelismo de tarefas). Cada Multiprocessador é constituído por um conjunto de unidades de processamento mais simples, considerados como *thin cores*, Processadores de Fluxo (SP), que são o segundo nível da hierarquia. Esses processadores de fluxo são notados como granularidade fina que compartilham os mesmos recursos dentro do SM e trabalham de forma síncrona no mesmo fluxo de instrução (SIMD).

A partir dessa consideração, podemos definir formalmente a hierarquia de processadores da GPU da seguinte forma:

1. Um conjunto de processadores de granularidade grossa (*coarse-grained*) $P = \{p_1, p_2, ..., p_s\}$, onde s é o número de Multiprocessadores;

2. Cada processador de granularidade grossa p_i, onde $1 \leq i \leq s$, possui um conjunto de uni-

dades de processamento de granularidade fina (*fine-grained*) $Core[1..b]$ que compartilham os mesmos recursos, onde b é o número de SP em cada SM.

A memória disponível na GPU é distribuída em diversas localidades na arquitetura da GPU, criando uma hierarquia de memória em que o tamanho juntamente com velocidade de acesso faz com que cada nível de memória seja distinto. O conjunto de processadores de granularidade grossa está interconectado à memória global da GPU, o nível mais baixo da hierarquia, enquanto as unidades de processamento de granularidade fina estão interconectados pela memória compartilhada, um nível intermediário, que se encontra dentro dos SMs. Por sua vez, cada SP possui acesso a uma memória local, constituída de registradores.

A GPU trabalha de forma cooperativa com a CPU. De forma padrão, ao iniciar um programa na arquitetura CPU-GPU, os dados inicialmente estão localizados na memória RAM e a CPU envia os dados à memória da GPU e o processamento para a GPU quando requisitado explicitamente no programa.

Neste trabalho, considera-se que o algoritmo principal é executado na CPU e as porções de códigos paralelos (*kernels*) são enviadas à GPU. Consideramos que as listas invertidas já estejam localizados na memória da GPU, assim ignora-se o tempo de transferência de grandes quantidade de dados. Essa direção de localidade permite apresentar eficiências dos algoritmos independentemente de qualquer tecnologia de rede de interconexão.

Há diversos trabalhos que discutem o desempenho de transferências através de tecnologias alternativas de rede de interconexão, além da traditional *Peripheral Component Interconnect Express* (PCIe), que apresenta largura de banda semelhante à da memória RAM (BUONO et al., 2017).

4.2 Investigação da Disposição dos Dados

Esta seção apresenta propostas que exploram a disposição de dados frente aos processadores paralelos que visam a melhorar a localidade dos dados na hierarquia de memória disponibilizada pela arquitetura da GPU. O resultado dessa investigação é a geração de duas estratégias de particiomento de dados entre os processadores de granularidade grossa apresentadas nas Subseções 4.2.1 e 4.2.2. Por meio dessas estratégias, é proposta uma estratégia de paralelização para o processamento de consulta e duas estratégias de paralelização para o processamento de lote de consultas, sendo que estas são apresentadas no Capítulo 6.

Dada uma consulta, os algoritmos sequenciais de poda dinâmica DAAT obtêm os top-k

documentos mais relevantes, atravessando todas as listas invertidas dos termos da consulta ao mesmo tempo para avaliar todas as ocorrências de um mesmo docID nas listas invertidas de todos os termos da consulta. Então, para executar qualquer algoritmo dessa abordagem, um processador precisa ter acesso a todas listas invertidas dos termos da consulta.

Neste intuito de abordar os algoritmos DAAT, são propostas partições que contêm partes de todas as listas invertidas da consulta para explorar a localidade de memória dos processadores. Esse particionamento é efetivado de maneira que cada partição tenha o tamanho $\theta(\frac{N_m}{|P|})$ documentos, onde N_m é o tamanho da maior lista invertida dentre os termos e o P é o conjunto de processadores de granularidade grossa. Assim, as listas invertidas dos termos da consulta são particionadas entre os processadores de granularidade grossa, de modo que cada partição contenha todas as listas invertidas dos termos da consulta. Dessa forma, garante-se que as partições se adequem nas memórias internas dos processadores de granularidade grossa. Esse particionamento é demonstrado na Figura 4.1. Seguindo essa ideia de particionamento, são propostas duas estratégias, Particiomento Homogêneo e Particionamento Heterogêneo, que são apresentadas nas Subseções 4.2.1 e 4.2.2, respectivamente.

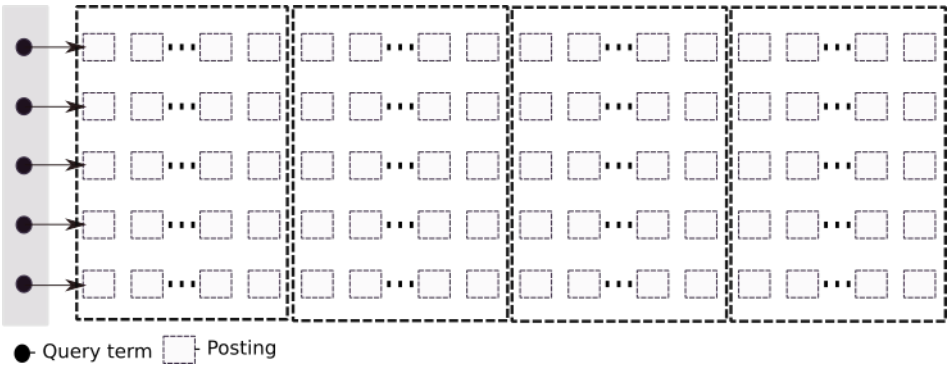

Figura 4.1: Divisão das listas invertidas.

4.2.1 Particionamento Homogêneo

Na primeira estratégia de particionamento, as listas de *postings* dos termos da consulta são divididas em partições homogêneas com o mesmo número de identificadores de documentos (docIDs). Os documentos são particionados de acordo com o número de multiprocessadores, como descrito na Figura 4.2.

Essa estratégia objetiva simplicidade e maximização dos processadores através do melhor balancemento de carga. Contudo, ela tem uma desvantagem implícita. Nos algoritmos DAAT, os documentos são avaliados completamente em uma única vez, ou seja, todas as ocorrências de um documento são avaliadas conjuntamente. Com as partições homogêneas, um dado identifi-

cador de documento pode aparecer em mais de uma lista de *postings* e em posições diferentes. Dessa forma, há possibilidade de que as ocorrências de um mesmo documento estejam em partições distintas e, assim, somente a avaliação parcial dos docIDs é realizada por um determinado processador. Nesse caso, os documentos relevantes podem ser descartados do resultado dos top-k documentos, fazendo com que essa estratégia tenha impacto na acurácia dos algoritmos. Por exemplo, na Figura 4.2, as ocorrências dos docIDs 6, 7 e 10 estão distribuídas nas partições dos SM_0 e SM_1. Essa distribuição faz com que os SMs consigam obter somente avaliação parcial desses docIDs.

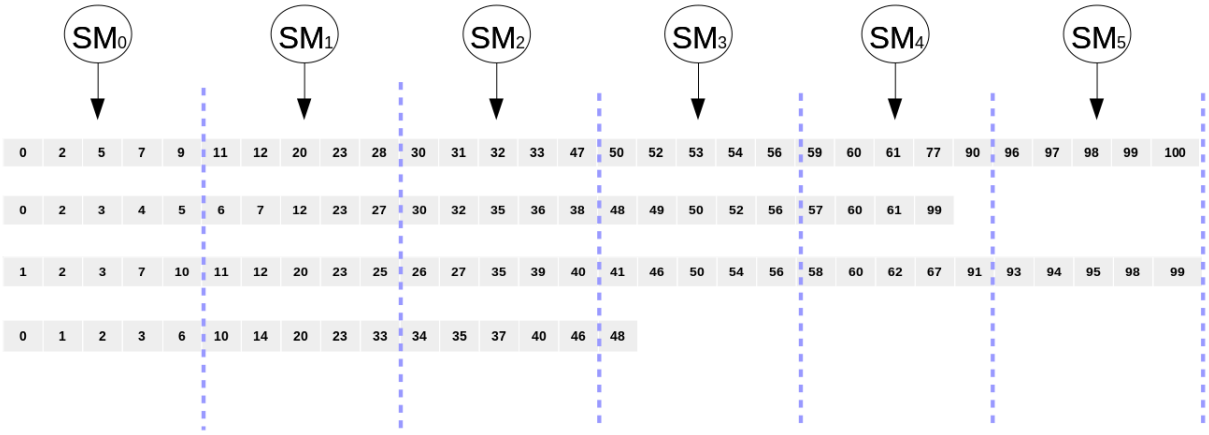

Figura 4.2: Estratégia de Particionamento Homogênea entre os SMs.

4.2.2 Particionamento Heterogêneo

A segunda estratégia propõe dividir as listas de *postings* em segmentos de acordo com intervalos de docIDs. Para isso, um processamento é efetuado nas listas de *postings* de maneira que todos segmentos tenham o mesmo intervalo de docIDs. Dessa forma, os segmentos poderão ter tamanho diferentes. Por essa razão, essa estratégia é nomeada de particionamento heterogêneo.

O particionamento é realizado a partir de uma sequência de fases. Na primeira fase, são obtidas as posições das partições homogêneas a partir do número de processadores, como mostrado na Figura 4.2, processo idêntico ao particionamento homogêneo. Posteriormente, na segunda fase, a partiçao de documentos é delimitada pelos documentos localizados nas extremidades das partições homogêneas. O docID inicial de uma partição é o maior docID localizado na primeira posição de fora da extremidade inicial (posição mais à esquerda) da partição de cada lista invertida adicionando-se uma unidade no seu valor. Enquanto o docID final é o maior valor de docID localizado na posição da extremidade final (posição mais à direita) da partição de cada lista invertida. As Figuras 4.3 e 4.4 detalham esse processo. Ao alcançar as faixas de docIDs, todas as ocorrências de um documento estarão localizadas em uma única partição,

evitando assim avaliações parciais.

A estratégia heterogênea pode produzir partições com tamanhos diferentes, como demonstrado no exemplo da Figura 4.4. Observa-se que neste exemplo houve desbalanceamento de carga entre os processadores, no caso da partição do SM_5, somente um docID é processado. Apesar desse desbalanceamento de carga, essa estratégia foi projetada para garantir os mesmos resultados dos algoritmos sequenciais.

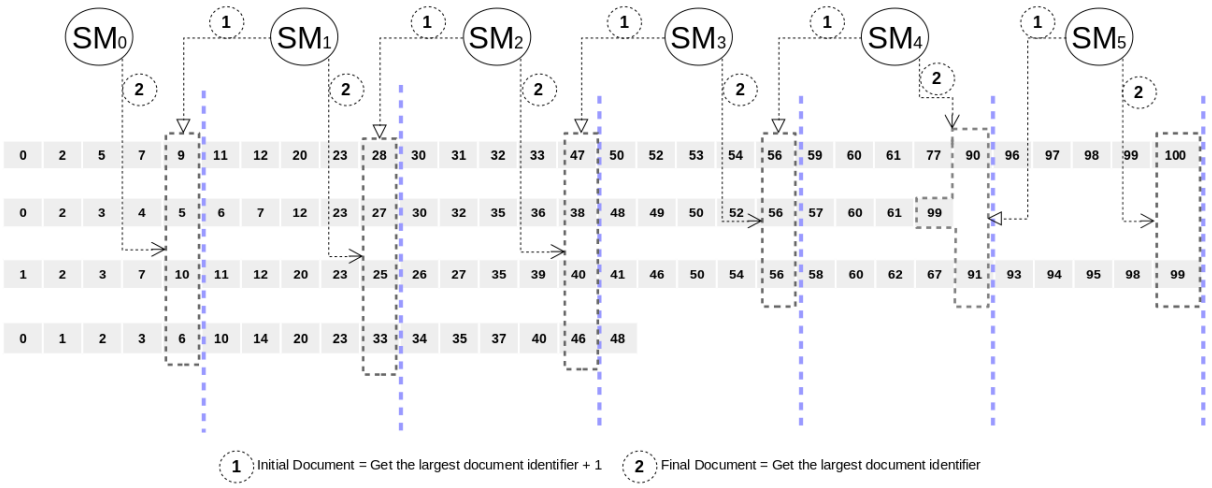

Figura 4.3: Obtenção dos documentos iniciais e finais.

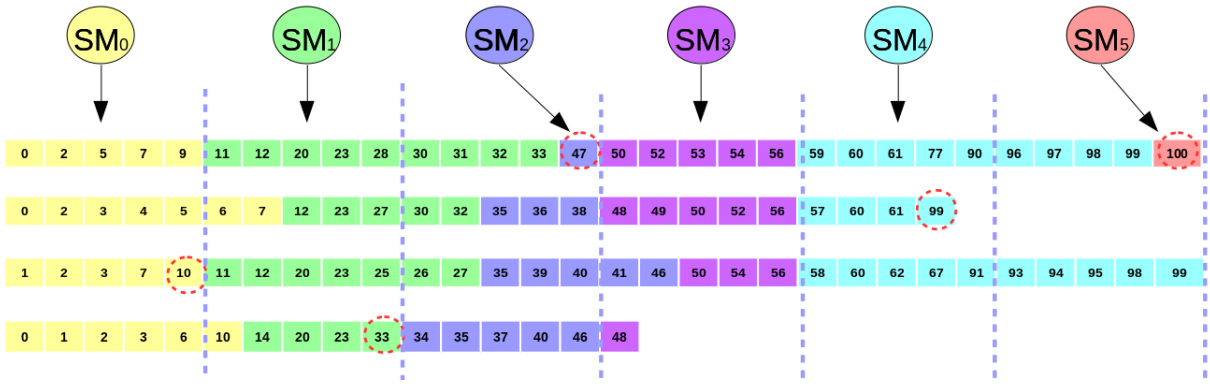

Figura 4.4: Partições heterogêneas geradas pela segunda fase.

Todas as estratégias de particionamento, homogênea e heterogênea, são realizadas dinamicamente para cada consulta de entrada e não requisitam qualquer processamento na construção do índice invertido, ou seja, não necessitam de processamento prévio à consulta. Isso significa que nenhuma informação extra é necessária no índice invertido para realizar os particionamentos. Por trabalhar com posições, ao obter os docIDs inicial e final, essas estratégias apresentam vantagens ao não exigir intersecções de listas invertidas e nem qualquer algoritmo de busca, por exemplo, a busca binária que é a mais utilizada nas soluções de intersecção. Na prática, os tamanhos das faixas das partições em cada lista invertida não são conhecidos ao processar

consultas, somente os identificadores de documentos iniciais e finais.

4.3 Proposta de Paralelização

Nesta seção, é apresentada uma proposta de paralelização do processo de identificação de documentos relevantes a partir de consultas sobre as estratégias de particionamento de dados. Essa proposta trabalha com a abstração da GPU apresentada na Seção 4.1. Essa abstração permite que o algoritmo seja facilmente modelado para outras arquiteturas paralelas *manycore* que possuem as características destacadas na abstração.

A proposta apresenta dois níveis de paralelismo. O primeiro nível é a execução concorrente, realizada pelos processadores de granularidade grossa, de algoritmos em diferentes porções das listas invertidas que foram geradas pelas estratégias de particionamento. No segundo, cada instância do algoritmo de processamento de consulta e do algoritmo de junção é realizada em paralelo pelos processadores de granularidade fina.

A proposta de paralelização do processamento de consulta é mostrada no Algoritmo 3. A paralelização de granularidade grossa ocorre com as chamadas de função nas linhas 1 e 4. A função *ParallelMatchProcessing* representa a execução de algoritmos paralelos de processamento de consulta, ao passo que a função *ParallelMerge* efetua a combinação de todos os resultados dos top-k documentos para obter somente os k documentos mais relevantes em paralelo. O Algoritmo 3 segue em um modelo inspirado no modelo BSP, podendo ser considerado com duas superetapas de processamento. A primeira superetapa executa a similaridade e a segunda combina os resultados das partições.

A avaliação dos documentos dos termos da consulta é alcançada na linha 1. Observa-se que até nesta linha não foram definidas as partições das listas invertidas. Isso leva à geração das partições a ser realizada pelos processadores de granularidade grossa em tempo de execução das instâncias do algoritmo representado pela função *ParallelMatchProcessing*.

Os k documentos identificados por cada processador pela função *ParallelMatchProcessing* são mantidos em uma estrutura de dados local que é compartilhada somente com os processadores de granularidade fina de um mesmo processador. Ao finalizar o processamento de uma partição, cada processador de granularidade grossa mantém os documentos identificados de uma partição em uma estrutura de dados global, representada pela variável *parcial_result*. Essa estrutura conterá todas as listas de top-k documentos de cada partição processada. A posição i da estrutura *parcial_result* contém os k documentos mais relevantes à consulta resultante da execução do processador P_i sobre a partição i.

A barreira de sincronização na linha 2 garante que todos os resultados parciais sejam conseguidos antes de iniciar a combinação dos resultados. Depois de todos os processadores completarem as buscas sobre suas partições, os resultados parciais são combinados em paralelo para se obter somente uma única lista de top-k documentos relevantes. O algoritmo nomeado de *ParallelMerge* efetua essa combinação pela chamada da linha 4. A obtenção de uma lista de top-k documentos relevantes global é alcançada pelas $\theta(\log(|P|))$ chamadas do algoritmo *ParallelMerge*.

Algorithm 3 Proposta de Processamento Paralelo de Consulta.

Input: [1] os termos da consulta $< t_1, ..., t_n >$; [2] o número de documentos a serem identificados k; [3] um conjunto de P processadores; [4] o número i de cada processador $p_i \in P$, onde $1 \leq i \leq P$;
Output: Lista dos top-k documentos $result[1..k]$;
1: Each processor $p_i \in P$ **in parallel**
   $parcial\_result[i] \leftarrow$ ParallelMatchProcessing($< t_1, ..., t_n >, k$) {Cada p_i processa $\theta(\frac{N_m}{|P|})$ docIDs e returna os top-k docIDs.}
2: sync_barrier()
3: **for** $j \leftarrow |P|; j > 0; j \leftarrow \frac{j}{2}$ **do**
4: Each processor p_i **in parallel**, where $i < j$:
      ParallelMerge($parcial\_result[1..(j * k)]$) {Cada p_i realiza uma junção de duas listas ordenadas dos top-k docIDs.}
5: sync_barrier()
6: **end for**
7: $result \leftarrow parcial\_result[1]$

O número de partições depende da maior lista invertida da consulta. Então, para o pior caso, a classificação é realizada com a complexidade de tempo de $\theta(\frac{N_m}{|P|})$, onde N_m é a maior lista da consulta, pela chamada da função *ParallelMatchProcessing* na linha 1. A junção das listas dos documentos mais relevantes é realizada com $\log(|P|)$ chamadas da função *ParallelMerge* que pode executar, no pior caso e com um algoritmo mais simples de *merge sort* citado em (CORMEN et al., 2009), em tempo de $\theta(k\log(k))$, onde k é uma constante que define o número de documentos retornados. Assim, pode-se dizer que o Algoritmo 3 possui uma complexidade de tempo de $\theta(\frac{N_m}{|P|} + \log(|P|))$.

4.4 Algoritmos Paralelos de Poda Dinâmica

Esta seção concentra-se em explorar o paralelismo nos algoritmos de poda dinâmica que apresentam características semelhantes e são bem trabalhados na literatura. Em particular, esse trabalho foca nos algoritmos WAND (BRODER et al., 2003) e MaxScore (TURTLE; FLOOD, 1995).

Para classificar os documentos dos termos da consulta, os algoritmos de poda utilizam algumas informações que são comuns entre eles para realizar saltos nas listas de *postings*. O entendimento dessas informações foi útil para alcançar estratégias eficientes. Dentre elas, podem ser destacadas:

- A maior pontuação de impacto de cada lista de *postings* do índice invertido, chamada de *upper bound* ou *maxscore*, é armazenada no dicionário do índice invertido e empregada para descartar documentos na avaliação completa de suas pontuações;
- Uma estrutura de dados de ordenação (em ambos os casos os trabalhos originais oferecem o heap mínimo - *min-heap*), mantém os top-k documentos durante as avaliações;
- A menor pontuação contida na estrutura de dados que armazena os top-k documentos, chamada de *threshold*, é atualizada dinamicamente de acordo com inserções sucessivas nessa estrutura e, assim, esse valor, conforme vai aumentando, define quais documentos serão avaliados completamente ao proceder com o processamento.

Os algoritmos sequenciais que são considerados neste trabalho, MaxScore e WAND, foram detalhados no Capítulo 2 pelos Algoritmos 1 e 2, respectivamente. Esses algoritmos percorrem todas as listas invertidas dos termos da consulta buscando as ocorrências de um mesmo documento. Isso acontece, pois eles fazem parte dos algoritmos DAAT. Essa característica faz com que esses algoritmos sejam de natureza sequencial e, desse modo, dificulta propostas paralelas eficientes dessa classe de algoritmos, como mencionado no trabalho de (DING et al., 2009).

Diante da dificuldade da paralelização desses algoritmos de processamento de consulta, este trabalho criou versões paralelas de granularidade fina dos algoritmos WAND e MaxScore que são apresentadas nos Algoritmos 4 e 5, respectivamente. Essas versões paralelas do processamento de consulta são adequadas na proposta de paralelização de granularidade grossa da classificação de documentos dada uma consulta, representada pela chamada da função *ParallelMatchProcessing*.

Em ambas as versões paralelas dos algoritmos de poda, os top-k documentos são mantidos em uma estrutura de ordenação, *topkDocs*. Embora não tenhamos fixado uma implementação dela, é proposto que a manutenção dessa estrutura seja realizada em paralelo pelos processadores de granularidade fina. O método *ManageTopkDocs* presente nas propostas representa as manutenções necessárias para inserir e excluir um elemento, mantendo sempre a propriedade de ordenação.

O núcleo da proposta de paralelização e das versões paralelas aqui oferecidas é composto por duas estratégias que focam na melhoria da eficiência dos algoritmos. Uma das estratégias trata das políticas de propagação de *threshold* que visam a impactar o desempenho da classificação sem afetar na acurácia dos resultados. A segunda estratégia está relacionada à realização das estratégias de particionamento por cada processador antes de iniciar a classificação dos documentos.

4.4.1 Políticas de Propagação de *Threshold*

A eficiência dos algoritmos de poda relaciona-se diretamente ao valor do *threshold* que possui um aumento contínuo no decorrer do processamento das listas invertidas. Quanto maior é o valor do *threshold*, mais documentos são descartados e mais rápida é a execução dos algoritmos.

No contexto paralelo, em cada partição, um processador de granularidade grossa pode conservar localmente uma lista dos top-k documentos e o *threshold*, como demonstrado na Figura 4.5. Embora essa ideia de implementação seja a mais básica para que se evite barreiras de sincronização e acessos exclusivos a variáveis compartilhadas em uma arquitetura paralela, ela não é a mais eficiente, pois o valor do *threshold* não tem o mesmo crescimento para todos os processadores e, com isso, menos documentos são ignorados. Por isso, este trabalho propõe propagar os valores dos *thresholds* encontrados pelos processadores com o intuito de obter mais eficiência no processamento de documentos, de forma que não degrade as acurácias dos resultados.

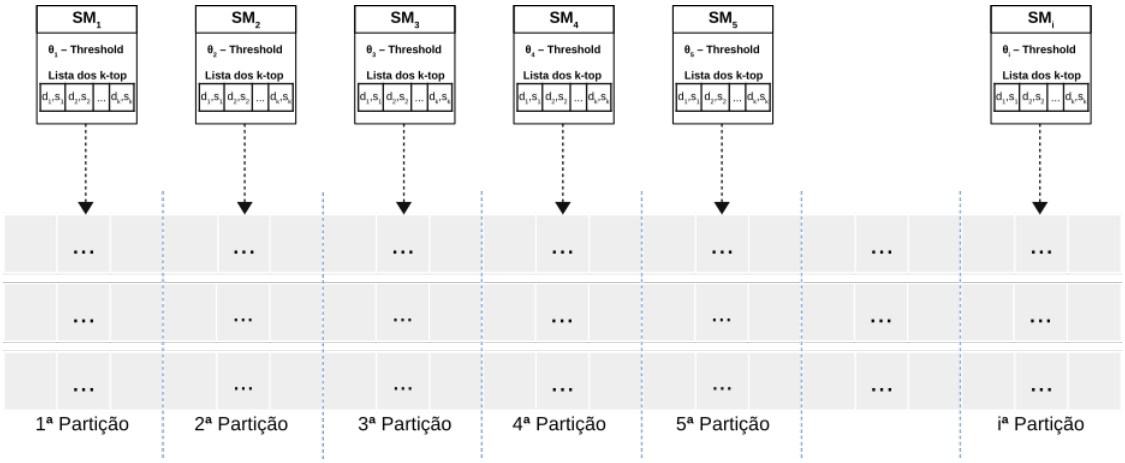

Figura 4.5: Manutenção das estruturas de dados dos algoritmos de poda.

Como a propagação do *threshold* é crítica ao desempenho do WAND, são oferecidas três implementações de políticas de propagação de *threshold* que se seguem:

1. Com um *threshold* local, cada processador não possui a sobrecarga de propagar seu valor para os demais. No entanto, trabalhando com valores mínimos locais, é provável que os processadores ignorem menos documentos;

2. A segunda política compartilha o valor do *threshold* por meio de uma variável global. Cada processador compartilha seu valor de *threshold* local com outros processadores por meio dessa variável global. Porém, com um máximo global sendo compartilhado, é possível que um valor muito alto seja encontrado (e compartilhado com os demais proces-

sadores) logo no início da busca e, com isso, outros processadores ignorem documentos que podem ser relevantes, mas têm seu valor inferior ao *threshold* propagado. Assim, esta estratégia retarda a propagação do *threshold* até que a lista esteja cheia. Como o *threshold* global é inicializado com o valor zero, cada processador pode ler com segurança seu valor para avaliar seus documentos, ou seja, ela garante que no momento em que os processadores lêem a variável compartilhada, existem pelo menos k documentos nas listas. Por esse motivo, chamamos essa política de *Safe-Read Shared (Sh-R)*. No entanto, uma possível desvantagem dessa estratégia é o atraso para compartilhar o *threshold*, causando um lento incremento do *threshold* global no início do processamento dos algoritmos.

3. Para evitar um possível prejuízo decorrente do atraso de propagação do *threshold* da política anterior, uma terceira política é proposta. Essa política adia o compartilhamento dos valores dos *threshold* até que os processadores tenham preenchido uma determinada quantidade mínima de documentos na lista dos documentos mais relevantes. Embora essa estratégia possa antecipar o compartilhamento do *threshold*, ainda há risco de perder documentos relevantes, por isso essa quantidade de documentos deve ser alta suficiente para que se reduza a possibilidade de propagar um valor alto do *threshold* logo no início do processamento. Para alcançar essa possibilidade, foi definido neste trabalho que cada processador adia as operações de leitura e de escrita na variável global até preencher pelo menos a metade da lista de top-k documentos. Essa estratégia é chamada de *Safe-Write-Read Shared (Sh-WR)*.

4.4.2 Realização do Particionamento das Listas Invertidas

As estratégias de particionamento, detalhadas na Seção 4.2, são executadas em paralelo pelos processadores de granularidade fina, por meio de operações de redução paralela. A decisão de realizar essas operações de forma dinâmica, além de eliminar qualquer armazenamento de informações extras no índice invertido, permite flexibilidade na execução das consultas.

Esse procedimento de particionamento é definido nas versões paralelas dos algoritmos de poda, Algoritmos 4 e 5, pela função nomeada de *RangeDocs* na linha 1. O comportamento dessa função depende da estratégia de particionamento escolhida. Se a escolha do particionamento for a estratégia heterogênea, as faixas de documentos em cada lista dos termos da consulta de uma partição terão os mesmos docIDs de limites, inicial e final, retornados pela chamada de *RangeDocs*. Enquanto com a estratégia homogênea, os limites da partição em cada lista serão distintos.

4.4.3 Algoritmo Paralelo baseado no WAND

A primeira proposta paralela de algoritmo de poda é baseada na lógica sequencial do WAND (BRODER et al., 2003), detalhada pelo Algoritmo 4. Ele mantém um ponteiro (*terms*) para cada lista de *posting* dos termos da consulta e para o próximo documento (*nextDoc*) a ser avaliado em cada lista. O algoritmo começa inicializando, em paralelo, esses ponteiros das listas e as faixas da partição em cada lista de *posting*, ou seja, o menor e o maior docIDs (d_{min}, d_{max}) de cada lista na partição. Isso pode ser visto na linha 1, onde são chamadas as funções *NextDoc* e *RangeDocs*. Esta adquire os docIDs das faixas da partição de acordo com a estratégia de particionamento utilizada. A partir desses docIDs obtidos, a função *NextDoc* alcança o próximo docID a ser avaliado.

As iterações sobre os documentos da partição são conservadas sequencialmente e realizadas através do laço de repetição da linha 4. A cada iteração, um documento é selecionado como pivô para ser analisado e classificado. Quatro métodos que auxiliam nessa classificação são utilizados: *FindPivotTerm*, *FullScore*, *NextDoc*, *ThresholdSharingStrategy* e *Sort*.

O primeiro método, *FindPivotTerm*, seleciona um termo como termo pivô, termo que possui o próximo documento a ser avaliado, a partir do *threshold* recebido. Essa seleção é realizada sequencialmente, enquanto que o restante dos métodos é executado em paralelo por processadores de granularidade fina que possuem identificadores menores ou iguais à quantidade de termos da consulta.

A aplicação do modelo de similaridade é efetuada em todas as ocorrências do docID do termo pivô pela função *FullScore* em paralelo e os resultados são retornados para a variável *score* que representa a pontuação total das classificações do documento. Após a obtenção da pontuação total, o valor obtido e o docID são enviados à função *ManageTopkDocs* que irá gerenciar o novo documento na lista dos documentos mais relevantes a partir da pontuação total do documento e do *threshold* corrente.

As invariantes do algoritmo sequencial WAND são mantidas pela função *NextDoc*, que obtém o docID maior ou igual ao valor passado por argumento nas chamadas da função nas linhas 8 e 10. O último método auxiliar, *Sort*, faz a ordenação crescente ou decrescente de acordo com o conjunto recebido. Se a lista for um conjunto de termos, linhas 2 e 12, então a ordenação é feita de modo crescente a partir do próximo docID candidato a ser avaliado de cada lista. Caso seja os top-k documentos, linha 16, a ordenação é realizada de modo decrescente diante das pontuações dos documentos.

A chamada do método *ThresholdSharingStrategy* na linha 14 verifica as condições da política

de propagação de *threshold* escolhida e retorna o valor atualizado do *threshold* global. Essa chamada garante que o maior valor do *threshold* é propagado sem impactar diretamente na acurácia dos resultados dos processadores.

As iterações do laço 4 finalizam quando o próximo documento a ser avaliado, representado por *pTerm.nextDoc*, possuir um identificador maior que docID definido pela chamada da função *RangeDocs* para a lista de *postings* do termo pivô. Após essa finalização, os top-k documentos identificados desta partição são ordenados de forma decrescente e retornados para a variável global *result*.

Algorithm 4 Processamento de Consulta em Paralelo baseado no WAND - ParallelMatch-Processing.

Input: [1] os termos da consulta $< t_1, ..., t_n >$; [2] o número de documentos a serem retornados k; [3] um conjunto de $Core[1..b]$ processadores, onde $Core[1..b] \in P$; [4] o número j de cada processador $\varepsilon_j \in Core$, onde $1 \leq j \leq b$;
Output: Lista dos top-k documentos $result[1..k]$;

1: Each processor $\varepsilon_j \in Core[1..b]$ **in parallel** processes, where $j \leq n$:
   $terms[j].term \leftarrow t_j$
   $terms[j].(d_{min}, d_{max}) \leftarrow$ RangeDocs($< t_1, ..., t_n >$, b, $|P|$) {Obtém as faixas de docIDs (d_{min}, d_{max}) nos *postings* dos $< t_1, ..., t_n >$.}
   $terms[j].nextDoc \leftarrow$ NextDoc(t_j, $terms[j].d_{min}$)
2: Sort($terms$) **in parallel**
3: $pTerm \leftarrow$ FindPivotTerm($terms$, θ)
4: **while** $pTerm.nextDoc \leq pTerm.d_{max}$ **do**
5: **if** $pTerm.nextDoc == terms[0].nextDoc$ **then**
6: Each processor $\varepsilon_j \in Core[1..b]$ **in parallel** processes, where $j \leq n$:
        $score \leftarrow$ FullScore($terms$)
7: $\theta \leftarrow$ ManageTopkDocs($topkDocs$, $score$, $pTerm.nextDoc$) **in parallel**
8: Each processor $\varepsilon_j \in Core[1..b]$ **in parallel** processes, where $j \leq n$:
        $terms[j].nextDoc \leftarrow$ NextDoc(t_j, $pTerm.nextDoc + 1$), where $terms[j].nextDoc == pTerm.nextDoc$
9: **else**
10: Each processor $\varepsilon_j \in Core[1..b]$ **in parallel** processes, where $j \leq n$:
        $terms[j].nextDoc \leftarrow$ NextDoc(t_j, $pTerm.nextDoc$), where $terms[j].nextDoc < pTerm.nextDoc$
11: **end if**
12: Sort($terms$) **in parallel**
13: $pTerm \leftarrow$ FindPivotTerm($terms$, θ)
14: $\theta \leftarrow$ ThresholdSharingStrategy(θ)
15: **end while**
16: $result \leftarrow$ Sort($topkDocs$) **in parallel**

4.4.4 Algoritmo Paralelo baseado no MaxScore

No processamento de consulta com o algoritmo MaxScore, as listas invertidas dos termos da consulta são separadas em dois conjuntos ao decorrer das avaliações, essenciais e não essenciais. Os documentos avaliados são buscados somente no conjunto das listas essenciais e, dessa forma, o algoritmo realiza saltos de documentos.

No início do algoritmo, todas as listas estão contidas no conjunto das listas essenciais. A diferenciação das listas inicia-se quando o crescimento do *threshold* ultrapassa o resultado da soma dos *maxscores* em ordem crescente ao *maxscore* de cada lista. As listas cujos valores dos *maxscore* contribuíram para a soma são excluídas do grupo das listas essenciais e são mantidas somente no grupo das não essenciais.

O Algoritmo 5 representa a versão paralela proposta do MaxScore. Na linha 1, o particionamento paralelo e a inicialização dos próximos documentos em cada lista são definidos. A lista *terms*, que contém as listas invertidas dos termos, é estabelecida em ordem crescente pelo próximo docID a ser processado pela função *Sort* na linha 2. A partir dessa ordenação, o algoritmo começa a iteragir sobre os documentos pelo laço de repetição da linha 3. O menor docID indicado pelo ponteiro *nextDoc* das listas essenciais é sempre avaliado. Isso é garantido na linha 4, onde o valor referente ao *nextDoc* é obtido da primeira lista e avaliado completamente em paralelo. A pontuação obtida juntamente com docID é passado para função *ManageTopkDocs* que gerencia em paralelo as operações necessárias na lista dos top-k documentos.

Todos os ponteiros do *nextDoc* cujos os valores são menores que o docID avaliado são atualizados sequencialmente, embora essas operações sejam realizadas em paralelo na linha 7. No final de cada iteração do laço da linha 3, o *threshold* é atualizado e/ou compartilhado de acordo com a política de propagação de *threshold* escolhida pela função *ThresholdSharingStrategy* na linha 9. Posteriormente, as listas essenciais são analisadas e atualizadas a partir do valor alcançado pelo *threshold*.

A condição de parada do algoritmo, laço da linha 3, é alcançada quando o conjunto das listas essenciais estiver vazio ou quando o próximo menor documento a ser avaliado está fora da faixa de documentos determinada pela função *RangeDocs* ((d_{min}, d_{max})). Todos os documentos selecionados para a lista dos top-k documentos são ordenados em paralelo, linha 12, antes do término do algoritmo.

Algorithm 5 Processamento de Consulta em Paralelo baseado no MaxScore - ParallelMatchProcessing.

Input: [1] os termos da consulta $< t_1, ..., t_n >$; [2] o número de documentos recuperados k; [3] um conjunto de $Core[1..b]$ processadores, onde $Core[1..b] \in P$; [4] o número j de cada processador $\varepsilon_j \in Core$, onde $1 \leq j \leq b$;
Output: Lista dos top-k documentos $result[1..k]$;

1: Each processor $\varepsilon_j \in Core[1..b]$ **in parallel** processes, where $j \leq n$:
   $terms[j].term \leftarrow t_j$
   $terms[j].(d_{min}, d_{max}) \leftarrow$ RangeDocs($< t_1, ..., t_n >$, b, $|P|$) {Obtém as faixas de docIDs (d_{min}, d_{max}) nos *postings* dos $< t_1, ..., t_n >$.}
   $terms[j].nextDoc \leftarrow$ NextDoc(t_j, $terms[j].d_{min}$)
2: $essentialTerms \leftarrow$ Sort($terms$) **in parallel**
3: **while** $essentialTerms \neq \emptyset$ and $essentialTerms[0].nextDoc \leq essentialTerms[0].d_{max}$ **do**
4: Each processor $\varepsilon_j \in Core[1..b]$ **in parallel** processes, where $j \leq n$:
      if $essentialTerms[0].nextDoc == terms[j].nextDoc$ **then**
        $score \leftarrow$ FullScore($terms[j]$)
      end if
5: $\theta \leftarrow$ ManageTopkDocs($topkDocs$, $score$, $essentialTerms[0].nextDoc$) **in parallel**
6: **for** $l \leftarrow 1; l > n; l++$ **do**
7: Each processor $\varepsilon_j \in Core[1..b]$ **in parallel** processes, where $j \leq n$:
        $terms[l].nextDoc \leftarrow$ NextDoc(t_l, $essentialTerms[0].nextDoc + 1$), where $terms[l].nextDoc \leq essentialTerms[0].nextDoc$
8: **end for**
9: $\theta \leftarrow$ ThresholdSharingStrategy(θ)
10: UpdateEssentialTerms($essentialTerms$, $terms$, θ) **in parallel**
11: **end while**
12: $result \leftarrow$ Sort($topkDocs$) **in parallel**

4.5 Detalhes de Implementação dos Algoritmos

A paralelização aqui proposta para o processo de identificar documentos relevantes à consulta foi apresentada sobre a abstração da arquitetura GPU descrita na Seção 4.1, onde os detalhes da arquitetura do hardware e software foram abstraídos. Nesta seção, são apresentadas as técnicas e estratégias de programação utilizadas que permitiram tornar possíveis as implementações dos algoritmos.

4.5.1 Abundância de *Threads* e Reutilização de Dados

A arquitetura CUDA, apresentada no Capítulo 3, é composta por Hardware e Software. O hardware constitui-se de um número limitado de Multiprocessadores, onde cada um contém um conjunto pequeno de processadores síncronos, cada qual com acesso a alguns diferentes tipos de memória. No nível de software, há uma virtualização dos processadores através da disponibilidade massiva de *threads*. Essa disponibilidade permite que o software implementado na arquitetura CUDA seja escalável frente a diferentes hardwares.

A partir dessa disponibilidade massiva de *threads*, definimos a quantidade de partições das listas invertidas em função do número de *threads* por bloco, como é esquematizado na Figura 4.6. Isso nos permite experimentar os algoritmos com diferentes quantidade de *threads* e, consequentemente, diferentes quantidade de partições por consulta.

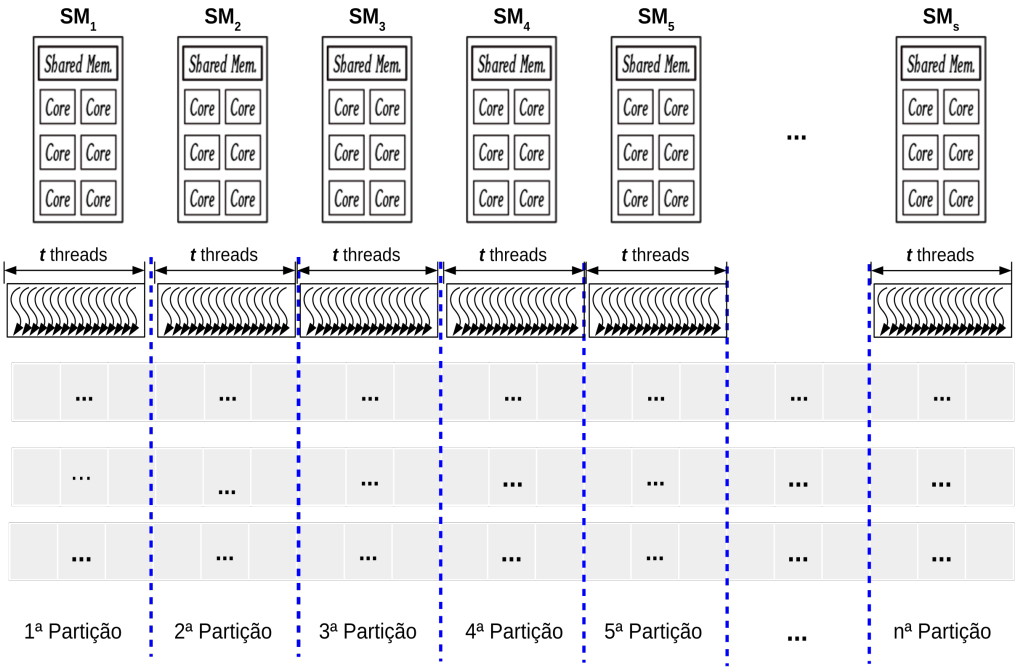

Figura 4.6: Particionamento das listas invertidas com *threads*.

A arquitetura oferece acessos a diferentes tipos de memória às *threads*. Uma memória que se destaca é a memória compartilhada, em que cada estado da memória é mantido por um único bloco de *threads*. Isso faz com que o tempo de vida de um contexto da memória compartilhada seja o mesmo de um bloco de *threads*. Então, para obter desempenho nos algoritmos propostos de forma que se reutilize os resultados das partições processadas, flexibilizamos nas implementações o número de partições por bloco de *threads*, sendo este número variável e determinado por experimentação.

A Figura 4.7 demonstra essa reutilização de dados. Sempre ao finalizar o processamento de uma partição, o contexto de um bloco de *threads* na memória compartilhada já apresenta resultados daquela partição. Ao reutilizar esses dados, os valores dos *thresholds* são altos já no ínicio dos próximos processamentos. Assim, as avaliações podem ser mais eficientes nas próximas partições. Contudo, existe neste processo uma serialização do processamento das partições, por isso há a necessidade de experimentar o número de partições por blocos de *threads* de acordo com o modelo da arquitetura utilizada combinado com as características das consultas.

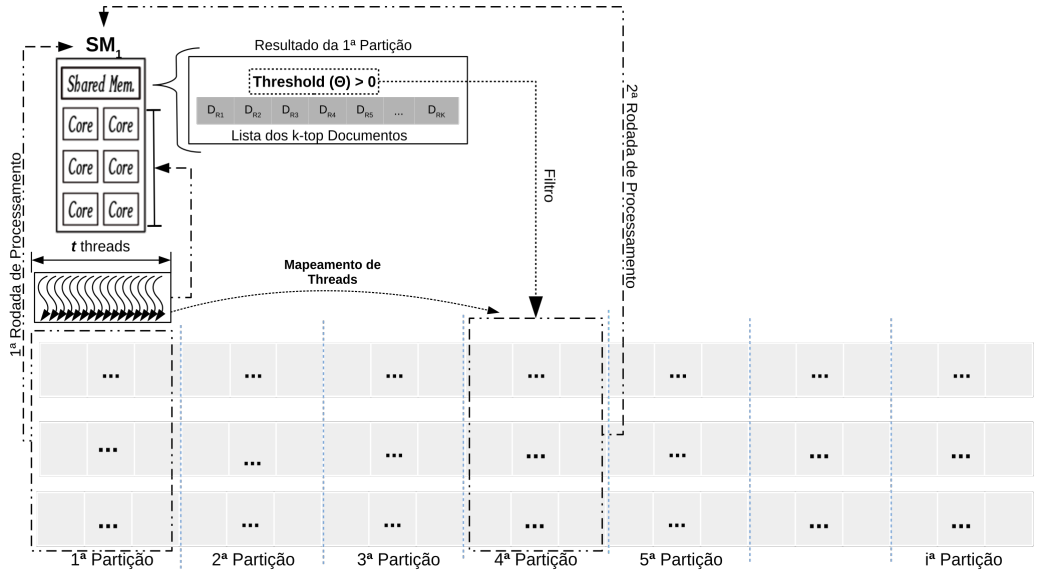

Figura 4.7: Reutilização dos resultados do processamento das partições.

4.5.2 Manutenção das Partições na Memória Compartilhada

A memória compartilhada é um dos componentes chaves da GPU. Ela é uma memória de baixa latência do tipo *on-chip* dos SMs, que possui uma pequena capacidade de armazenamento e oferece alta largura de banda quando comparada com a memória global. Geralmente, a memória compartilhada é usada como cache de dados da memória global e para comunicação

intra blocos (CHENG; GROSSMAN; MCKERCHER, 2014). Um bom uso da memória compartilhada pela aplicação pode levar a grandes ganhos de desempenho.

As estratégias de particionamento foram propostas de maneira que cada partição tenha um tamanho $\theta(\frac{N_m}{|P|})$ documentos, onde N_m é o tamanho da maior lista invertida dentre os termos. Os algoritmos utilizados para o processamento de consulta são da categoria DAAT que analisam todas as ocorrências de um documento juntas. Para que um documento tenha avaliação completa, todas as ocorrências precisam estar em uma mesma partição e, caso se faça a utilização da memória compartilhada, precisam estar no contexto de um mesmo SM.

Com o intuito de que as implementações façam uso eficente da memória compartilhada de forma que não exista limites no tamanho das partições, principalmente das partições da estratégia heterogênea, foi necessário fazer adequações das partições geradas. Assim, com o tamanho limitado da memória compartilhada, as partições não conseguiriam conter todos os dados das partições.

As adequações efetivadas dependem do número de *threads* por bloco de *threads* e do número de documentos que se deseja armazenar na memória compartilhada. Este último é determinado experimentalmente, pois depende de aspectos da arquitetura, como largura de banda e quantidade de bytes de armazenamento, e depende, também, das características da consulta, por exemplo, tamanho das listas invertidas. Além disso, o modelo de similaridade aplicado também pode influenciar, pois mais dados juntos aos documentos podem ser necessários e, portanto, mais bytes de armazenamentos são requisitados.

Determinado um número m de documentos residentes na memória compartilhada, divisões sucessivas são aplicadas nas partições de forma que as ocorrências de um mesmo documento fiquem residentes na memória compartilhada em um dado instante. Esse processo é feito da seguinte forma. Primeiro, em uma $i-$ésima divisão, obtém-se o menor docID ($docID_{limite}$) das posições $i * m$ dentro da partição. Posteriormente, carrega todos os docIDs não avaliados iguais ou menores que o $docID_{limite}$ para a memória compartilhada. Por último, o processo de avaliação é realizada nos documentos contidos na memória compartilhada. Ao finalizar esse processamento, um novo processo de divisão é inicializado, atualizando o $docID_{limite}$. Esse processo é repetido até que todos os documentos da partição sejam avaliados. Dessa forma, as localizações das ocorrências de um mesmo documento são garantidas na memoria compartilhada em um mesmo instante.

Para exemplificar este processo, as partições do SM_1 e SM_4 explicados anteriormente na Figura 4.4 são usados como exemplo na Figura 4.8. Neste exemplo, o número de documentos residentes na memória foi definido como 5 documentos. Assim, ambas partições ultrapassa-

ram esse limite de documentos e, dessa forma, não se adequam totalmente dentro da memória compartilhada em uma única passada. A Figura 4.8 demonstra esse processo, onde a parte (a) representa a primeira divisão, enquanto a parte (b), a segunda. Os documentos a serem considerados nas divisões estão visíveis na figura. Observa-se que neste caso foi necessário duas etapas de processamento para cada partição. Além dessa sequenciação, uma desvantagem, semelhante à geração das partições heterogêneas, é o desbalanceamento de dados em cada SM. Podem existir, então, vários documentos em uma subpartição, enquanto em outras, poucos.

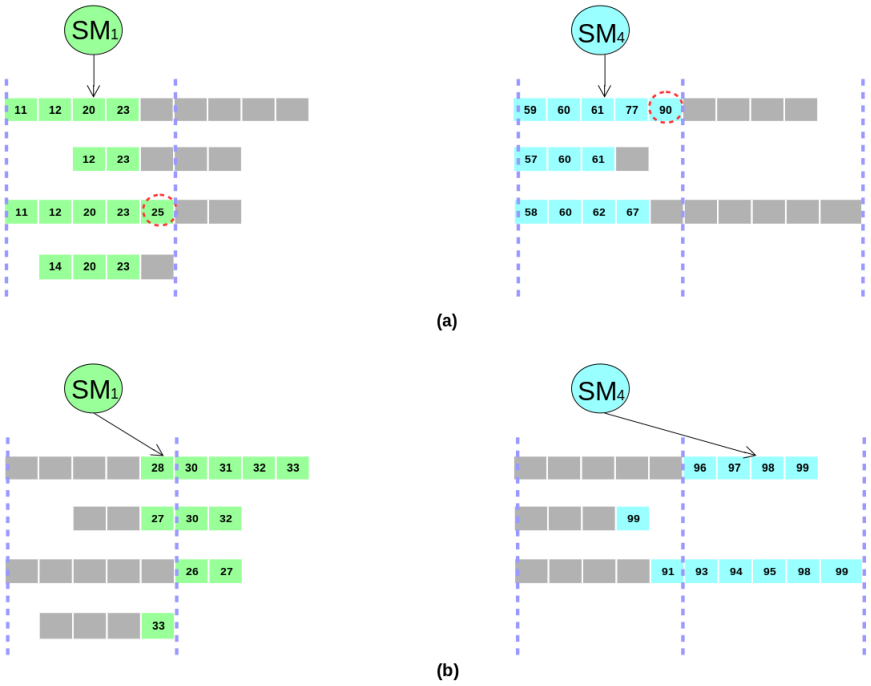

Figura 4.8: Divisão das Partições para se adequar na memória compartilhada.

4.5.3 Implementação da Lista dos Documentos mais Relevantes

A estrutura de dados comumente utilizada nos algoritmos sequenciais DAAT para manter os documentos mais relevantes é a estrutura *heap* mínimo (*heap-min*). Essa estrutura oferece tempo de acesso constante ao elemento de valor mínimo e operações de manutenção da ordem $\log(n)$, onde n é a quantidade de elementos mantidos pela estrutura. Por causa dessas vantagens, este trabalho escolheu o *heap-min* para manter a lista dos top-k documentos das propostas. Essa implementação do *heap-min* é local a cada *SM*.

O *heap-min* é uma estrutura de natureza sequencial, por isso é oferecido um paralelismo com poucas *threads* operando sobre ela. A solução aqui oferecida opera com o número máximo de 32 *threads* em paralelo sem sincronização explícita. Essa quantidade de *threads* permite trabalhar com *heaps-min* de 32 de altura. As 32 *threads* escolhidas fazem parte de um mesma

warp, cujas *threads* possuem a característica de serem síncronas e trabalham nos SPs de um mesmo multiprocessador da GPU. Essas características da arquitetura retiram a necessidade de barreira de sincronização (*syncthreads*) no processo da manutenção do *heap-min*.

A implementação paralela do *heap-min* baseia-se em três casos possíveis. No início do processamento dos documentos, a lista dos documentos mais relevantes encontra-se vazia. Então, a primeira inserção de um elemento é trivial, sendo preenchida a primeira posição em um único passo.

O segundo caso acontece quando o *heap-min* não está complementamente cheio. A manutenção do *heap* nesse caso ocorre a partir de dois passos. Primeiro, as *threads* são posicionadas nas posições dos nós pais da próxima posição a ser preenchida até o nó raíz, fazendo um caminho de manutenção no *heap*. Esse posicionamento é feito por meio dos identificadores das *threads*, sendo que a *thread* com o identificador *i* corresponde a um nó pai de altura *i*. Posteriormente, todas as *threads* em paralelo verificam se mantêm o elemento ou duplicam o valor contido no nó pai de sua posição ou inserem o novo elemento na sua posição mapeada. Dessa forma, a propriedade do *heap-min* é mantida. A Figura 4.9 demonstra esse processo.

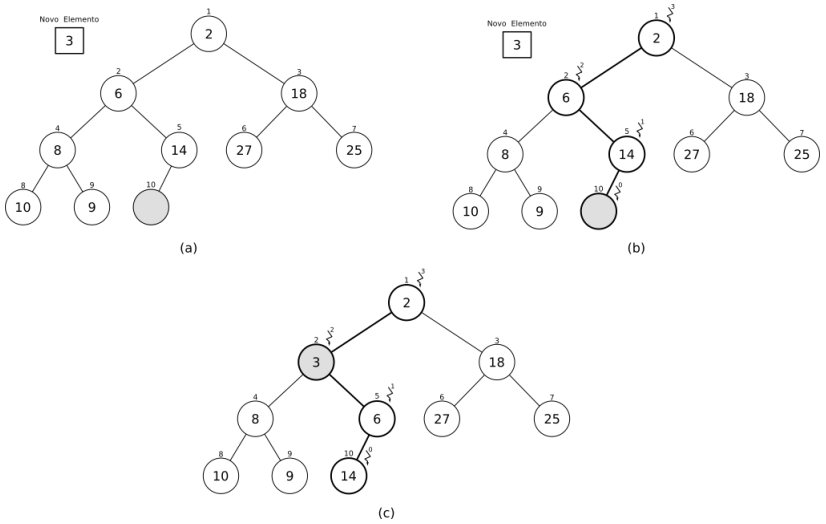

Figura 4.9: Inserção de um elemento em um *heap-min* não completamente preenchido.

A inserção de um elemento em um *heap-min* completamente preenchido é o terceiro caso. A manutenção das propriedades do heap é mantida em três etapas. Na primeira etapa, o menor elemento do *heap* contido no nó raiz é substituído pelo novo elemento a ser inserido. Logo após a inserção, procuram-se os nós filhos a serem alterados, ou seja, os *sub-heaps* que não terão a propriedade de mínimo mantida. As *threads* percorrem o *heap* a partir do nó raiz e descem ao nó filho cujo valor é o menor entre os dois nós filhos. Essa etapa é realizada pelas *threads* no pior caso em ordem de $\theta(\log(n))$, onde n é o tamanho do *heap*. A *thread* com identificador *i* irá

participar do processo se o *sub-heap* com nó raiz com altura de i não mantiver as propriedades da estrutura do *heap-min*. Após as *threads* estarem posicionadas em cada *sub-heap*, ocorrem as alterações paralelas nos nós onde a propriedade de mínimo foi desfeita. Essas alterações são realizadas em paralelo em ordem de tempo constante $\theta(1)$. Esse processo de inserção em um *heap-min* completo é representado pela Figura 4.10.

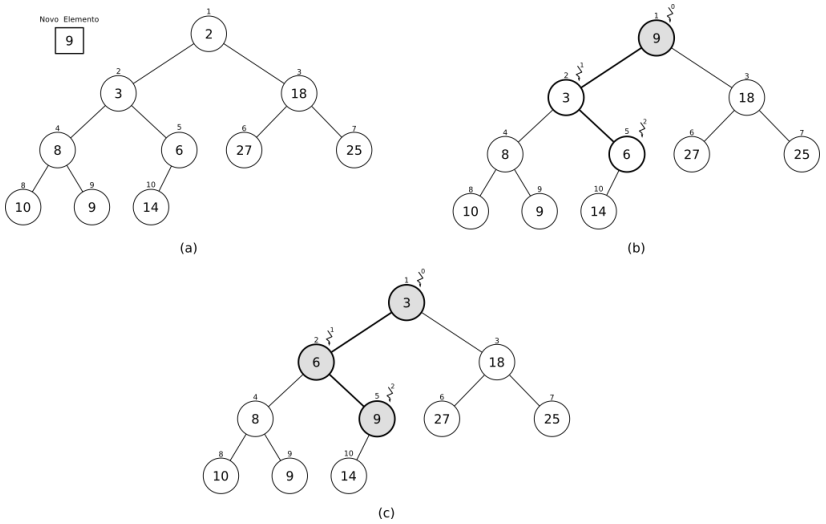

Figura 4.10: Inserção de um elemento em um *heap-min* completamente preenchido.

4.5.4 Ajustes nas Partições Homogêneas

A estratégia de particionamento homogênea produz partições de tamanho iguais, de modo que ocorrências de um mesmo documento podem estar localizadas em partições diferentes. Isso faz com que haja influência na acurácia dos resultados, pois as avaliações em determinados documentos ocorrem de maneira parcial. A Figura 4.11 explica esse comportamento. Nela, as ocorrências do docID 32, por exemplo, estão localizadas nas partições dos SM_2 e SM_3, impossibilitando uma avaliação completa desse documento por um desses SMs.

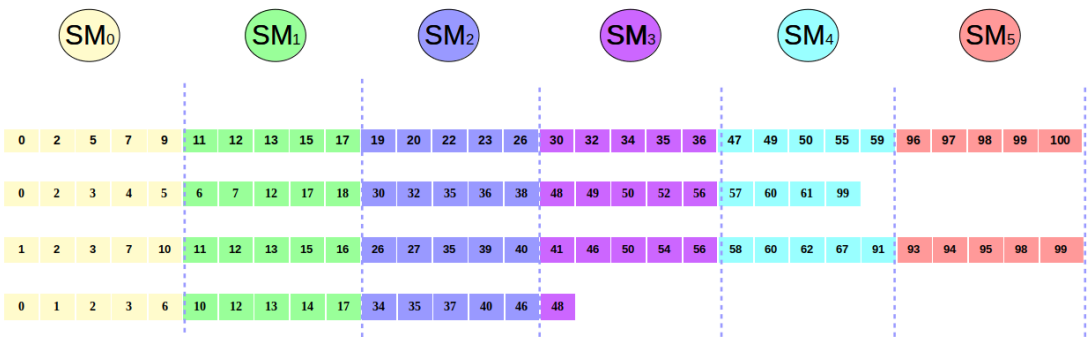

Figura 4.11: Estratégia de particionamento homogênea.

Para minimizar as avaliações parciais, propomos um ajuste nas posições das partições em

cada lista de *postings* da consulta. O ajuste propõe alinhar as faixas das partições sem alterar o tamanho delas. Para isso, obtém-se o maior docID, $docID_{maior}$, das primeiras posições de cada faixa da partição, além de realizar uma busca em cada lista de *postings* procurando o primeiro elemento maior ou igual ao $docID_{maior}$.

A Figura 4.12 mostra a aplicação desse processo nas partições demonstradas na Figura 4.11. As faixas apenas foram ajustadas para que o valor do docID inicial fosse menor que o $docID_{maior}$ a uma distância de 5 unidades. Na Figura 4.12, observa-se que algumas ocorrências foram descartadas (documentos de cinza), enquanto outras foram avaliadas completamente. Além disso, alguns documentos foram avaliados por dois processadores.

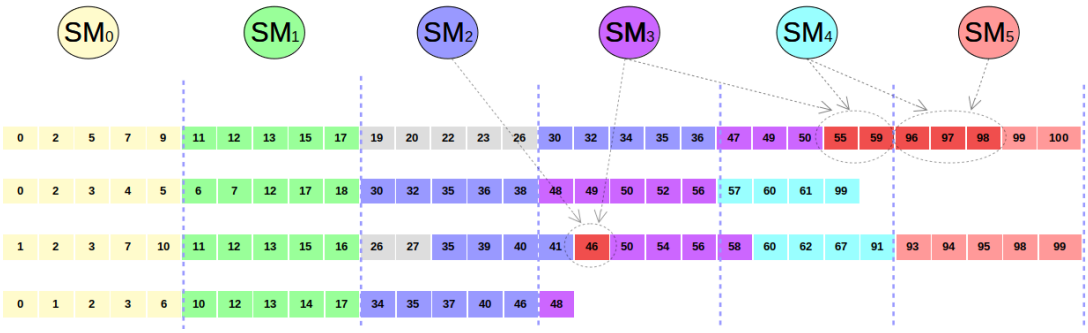

Figura 4.12: Estratégia de particionamento homogêneo ajustado.

4.5.5 Ocupação dos Processadores

No contexto da arquitetura da GPU, a ocupação refere-se a uma proporção que mede o número de *threads* que serão executadas em cada SM com o número máximo de *threads* que potencialmente poderiam estar sendo executadas em cada SM.

O número máximo de *threads* em um SM é uma constante que depende unicamente da capacidade do modelo do hardware. Ademais, o número de *threads* que serão executadas em cada SM ao lançar um *kernel* é uma função que depende dos seguintes recursos a serem utilizados pela aplicação: (1) *threads* por bloco, (2) registradores por bloco, (3) registradores por *thread* e (4) memória compartilhada por bloco. Para melhorar a ocupação de uma aplicação, é necessário redimensionar as configurações dos blocos de *threads* ou reajustar o uso de recursos. A arquitetura CUDA permite ajustar via software os usos de recursos através das configurações de execução e parâmetros de compilação. Todos esses métodos são conduzidos através de experimentos.

O aumento da ocupação de uma aplicação não leva necessariamente a um ganho de desempenho da mesma aplicação CUDA. Às vezes, é preciso combinar mais registradores por

thread com paralelismo no nível de instrução para obter ganhos de desempenho (VOLKOV, 2010). Por isso, há uma necessidade de analisar as características do algoritmo juntamente com as características da arquitetura.

Na paralelização proposta, o paralelismo dos processadores de granularidade grossa (SM) é desempenhado pelos blocos de *threads* nas implementações. Quanto mais blocos lançados em um único *kernel*, o processamento de consulta passa a ter mais paralelismo de granularidade grossa. Em relação ao paralelismo de granularidade fina, os algoritmos paralelos de poda dinâmica detalhados pelos Algoritmos 4 e 5 mostram que esse paralelismo é limitado ou por número de termos da consulta ou por número de documentos retornados (valor de k). Por isso, as implementações utilizam quantidades pequenas de *threads* por bloco e números altos de registradores por *threads*. Essa configuração permite que seja lançado mais blocos para abranger todas as listas invertidas da consulta no processamento. Dessa forma, a quantidade de blocos pode superar a quantidade de SMs disponíveis nas arquiteturas da GPU.

A Figura 4.13 mostra o excesso de paralelismo de granularidade grossa gerado pela configuração das *threads* no algoritmo de processamento de uma consulta na arquitetura da GPU, onde um SM pode processar simultaneamente diversas instâncias do processamento. As instâncias explanadas na Figura 4.13 utilizam o algoritmo paralelo de processamento de consulta baseado no WAND (Algoritmo 4) como exemplo.

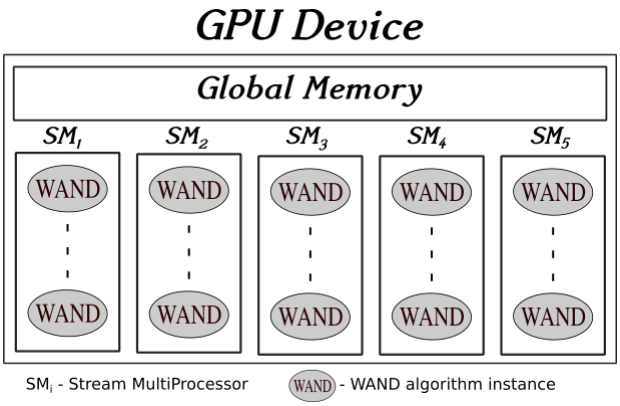

Figura 4.13: Processamento paralelo do algoritmo WAND.

4.6 Considerações Finais do Capítulo

Neste capítulo, foram apresentadas as estratégias de paralelização de algoritmos para o processamento de consultas em uma abstração da arquitetura da GPU. Essas estratégias adequaram-se às abordagens de algoritmos DAAT e aos algoritmos de poda dinâmica. Com o intuito de maximizar o desempenho dessas estratégias, as principais técnicas de programação da API

CUDA necessárias para concretizar as implementações foram detalhadas.

Capítulo 5

AVALIAÇÃO DE DESEMPENHO DOS ALGORITMOS DE PROCESSAMENTO DE CONSULTAS

Este capítulo tem por objetivo avaliar o desempenho das estratégias de processamento de um consulta apresentados no Capítulo 4.

5.1 Ambiente de Experimentos

Para os experimentos de avaliação de desempenho, foi escolhido um conjunto de 50,2 milhões de documentos do TREC ClueWeb09 (categoria B), que é parte do corpus ClueWeb09. Foram indexados todos os documentos usando a plataforma Terrier IR 4.2 (TERRIER, 2014) (OUNIS et al., 2005), resultando em um índice de tamanho 29 GB, em que o termo mais frequente produziu uma lista invertida com 2,7 milhões de documentos. Os experimentos foram guiados em duas direções, consultas sintéticas e log de consultas, buscando os top-k documentos mais relevantes de uma consulta, onde k foi configurado com o valor igual a 128.

Os algoritmos sequenciais do WAND e MaxScore foram executados em um único núcleo do processador Intel(R) Core(TM) i7-5820K de 3.30GHz e 64GB de memória RAM e implementados em JAVA com a versão 1.8.0_171 do jdk e inseridos dentro da plataforma Terrier IR 4.2 para serem executados. As propostas de paralelização foram implementadas em C/CUDA com sdk 9.1 e executadas em uma única NVIDIA GPU Titan Xp com 3840 CUDA cores distribuídos em 15 Multiprocessadores (SM) com 12 GB de memória global. As especificações completa dos hardwares, CPU e GPU, podem ser visualizadas na Tabela 5.1.

O uso de consultas sintéticas permite determinar o impacto do comprimento das listas invertidas sobre o desempenho dos algoritmos paralelos em processamento de consultas isoladas.

CPU:	
Nome	Intel i7-5820K
Ano de lançamento	2014
Frequência (GHz)	3,30
Cache (MB)	15
# de núcleos	6
# de *threads*	12
Threads/núcleo	6
Largura de banda máxima (GB/s)	68
Tipo da memória	DDR4
Capacidade da memória (GB)	64
GPU Pascal:	
Nome	NVIDIA TITAN Xp
GPU (codenome)	Pascal GP102
Ano de lançamento	2017
CUDA Cores	3840
# de Multiprocessadores (SM)	30
Capacidade da memória compartilhada (KB)	64
Frequência (GHz)	1,582
Velocidade da memória (Gbps)	11,4
Capacidade da memória (GB)	12
Interface da memória	GDDR5X
Largura da Interface de Memória	384-bit
Largura de banda máxima (GB/s)	547,7

Tabela 5.1: Especificações do hardware.

Comparações foram realizadas com consultas de dois termos, portanto compostas por duas listas de *postings* de diferentes comprimentos, curto, médio, longo e extralongo. Os termos foram selecionados aleatoriamente, de modo que o número total de docIDs de cada consulta varia em torno dos valores ilustrados na Tabela 5.2. Para cada tamanho das listas invertidas, 20 (vinte) consultas foram geradas e todas foram experimentadas 10 vezes. O maior e o menor valor foram removidos dos resultados finais. Assim, os resultados são a média das execuções das 18 consultas restantes.

# de termos	**Tamanho da Consulta**	**# médio de docIDs por lista invertida**
2	Curto	89.615
2	Médio	476.771
2	Longo	1.032.795
2	Extra	5.494.285

Tabela 5.2: Características das consultas sintéticas.

Neste primeiro experimento do processamento de consultas, foi avaliado o desempenho das propostas com as estratégias de particionamento homogêneo e heterogêneo executando consultas disjuntivas (OR) e consultas conjuntivas (AND) para achar os top-k (k=128) documentos mais relevantes. Consultas compostas por 2 termos foram escolhidas por serem mais representativas para maioria das buscas em máquinas de busca WEB. As versões paralelas dos algoritmos WAND e MaxScore foram implementadas e testadas. Para cada algoritmo, as três abordagens

do *threshold* foram realizadas e avaliadas: local, *safe-R shared* e *safe-WR shared*. Todos os algoritmos usaram a implementação global do *upper bound* e do *maxscore*.

Foram escolhidos 32 *threads* por bloco de *threads* dado que essa configuração apresentou os melhores resultados em relação a outros casos. Os algoritmos paralelos foram experimentados com os seguintes parâmetros:

- *Tamanho da Partição*: número de docIDs de cada lista de *postings* dos termos da consulta contidos em uma partição. Duas quantidades foram usadas: 32 docIDs e 64 docIDs;
- *Número de Partições*: quantidade de partições que um bloco de *threads* irá processar. Foram realizados com os seguintes casos: 1, 5, 10, 100, 150 e 200. No caso de uma partição por bloco, as estruturas de dados (*heap* e *threshold*) são recriadas no início de cada partição e não são reutilizadas (o que não gera reuso dos dados); por outro lado, mais blocos de *threads* são necessários para o processamento, o que dá mais oportunidade ao escalonador para melhorar a ocupação dos processadores. Nos casos de ter mais partições por bloco, essas estruturas são reutilizadas, o que tende a ser mais eficiente. Porém, haverá um número menor de blocos, o que tende a reduzir a ocupação dos processadores. Portanto, os valores deste parâmetro foram variados para investigar um possível *tradeoff*.

5.2 Avaliação do Algoritmo Paralelo WAND

5.2.1 Consultas OR

Os tempos de execução em milissegundos da implementação sequencial do algoritmo WAND para consultas OR são mostrados na Tabela 5.3. Esses tempos foram utilizados como referência para o cálculo do *speedup*. Todos os experimentos buscam os 128 documentos mais relevantes ($k = 128$). O algoritmo gastou um tempo de $\sim 14,01$ ms para avaliar 89.615 documentos, enquanto para avaliar consultas com 5,49 milhões de documentos gastou $\sim 202,79$ ms. Com isso, pode observado que o algoritmo sequencial é eficiente, pois com o aumento de $61\times$ do tamanho da entrada, o algoritmo aumentou o tempo de execução em $14\times$.

# de termos	Tamanho da Consulta	# médio de docIDs	Tempo médio (ms)
2	Curto	89.615	14,01
2	Médio	476.771	42,86
2	Longo	1.032.795	45,52
2	Extra	5.494.285	202,79

Tabela 5.3: Tempo médio de execução (ms) do algoritmo sequencial WAND para consultas OR.

A Tabela 5.4 apresenta os tempos médios de execução e os *speedups* da proposta paralela com algoritmo WAND para consultas OR. Inicialmente, as execuções foram testadas com 1 e 10 partições por bloco de *threads*. Os melhores *speedups* de cada caso foram destacados.

Algor.	**Thres-hold**	**Tamanho da Part.**	**# de Part.**	Curto		Médio		Longo		Extra	
				Tempo	**Speedup**	**Tempo**	**Speedup**	**Tempo**	**Speedup**	**Tempo**	**Speedup**
Hom.	Local	32	1	0,54	25,84	3,28	13,05	11,05	4,12	509,39	0,40
Hom.	Local	32	10	1,73	8,08	2,22	**19,34**	4,62	9,85	57,14	3,55
Hom.	Local	64	1	0,51	**27,58**	2,53	16,91	7,34	6,21	271,54	0,75
Hom.	Local	64	10	3,02	4,63	3,41	12,55	4,30	**10,58**	35,01	**5,79**
Hom.	Sh-R	32	1	0,57	**24,55**	3,42	12,54	11,31	4,02	508,60	0,40
Hom.	Sh-R	32	10	1,75	8,03	2,05	**20,86**	3,50	12,99	49,32	4,11
Hom.	Sh-R	64	1	0,54	25,83	2,62	16,34	7,46	6,10	270,41	0,75
Hom.	Sh-R	64	10	3,19	4,39	3,14	13,63	3,11	**14,63**	27,12	**7,48**
Hom.	Sh-WR	32	1	0,55	**25,27**	3,33	12,85	11,15	4,08	509,39	0,40
Hom.	Sh-WR	32	10	1,74	8,04	2,30	**18,60**	3,89	11,70	50,08	4,05
Hom.	Sh-WR	64	1	0,56	25,05	2,58	16,58	7,41	6,15	269,57	0,75
Hom.	Sh-WR	64	10	3,22	4,35	3,20	13,38	3,62	**12,56**	27,70	**7,32**
Het.	Local	32	1	1,76	**7,94**	3,93	10,90	10,07	4,52	225,70	0,90
Het.	Local	32	10	4,11	3,41	4,23	10,14	7,72	5,90	36,38	5,57
Het.	Local	64	1	2,50	5,61	3,68	**11,64**	7,60	**5,99**	124,15	1,63
Het.	Local	64	10	5,46	2,57	5,86	7,32	11,75	3,87	31,98	**6,34**
Het.	Sh-R	32	1	2,51	**5,58**	3,79	11,31	7,20	6,32	218,14	0,93
Het.	Sh-R	32	10	4,19	3,34	3,87	11,08	4,56	9,99	24,78	8,19
Het.	Sh-R	64	1	2,74	5,11	3,78	**11,34**	5,10	8,93	118,70	1,71
Het.	Sh-R	64	10	6,02	2,33	4,96	8,64	4,36	**10,44**	19,45	**10,43**
Het.	Sh-WR	32	1	2,53	5,53	4,10	10,44	6,78	6,71	222,14	0,91
Het.	Sh-WR	32	10	3,35	4,18	4,26	10,05	4,22	**10,78**	25,41	7,98
Het.	Sh-WR	64	1	2,01	**6,98**	3,56	**12,04**	6,16	7,39	121,90	1,66
Het.	Sh-WR	64	10	5,07	2,76	4,08	10,51	4,37	10,41	20,20	**10,04**

Tabela 5.4: Desempenho do algoritmo paralelo WAND para consultas OR com 2 termos. As colunas de 'Tempo' trazem o tempo de execução em milissegundos para cada teste.

Como esperado, o algoritmo homogêneo alcançou um melhor balanceamento de carga com as partições de mesmo tamanho e, assim, melhores *speedups*. A Figura 5.1 ilustra o gráfico da diferença de *speedups* dos algoritmos homogêneo e heterogêneo. Conquanto o homogêneo tenha apresentado um melhor desempenho frente ao heterogêneo, não há garantia de que cada partição homogênea tenha o mesmo intervalo de docIDs em suas listas de *postings*. Como consequência, esse algoritmo pode perder documentos relevantes durante o processo de busca e produzir resultados menos precisos, influenciando negativamente na acurácia, como mostrado na Tabela 5.5.

Porém, nas consultas extralongas, o algoritmo heterogêneo superou os *speedups* do algoritmo homogêneo. A obtenção de pontuações completas pode ter influenciado na ocorrência desse fato, pois o alto valor do *threshold* pode ter sido mais eficaz do que o aumento do processamento nas partições heterogêneas, ao permitir saltar mais documentos.

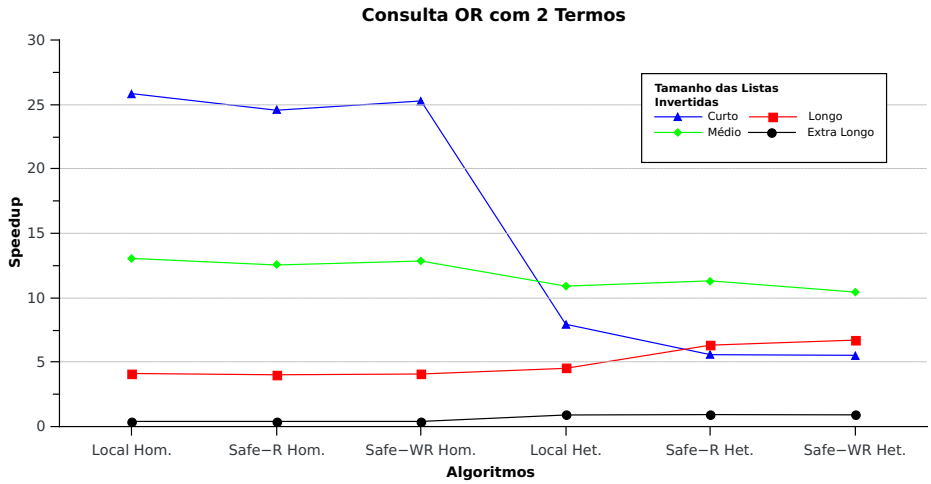

Figura 5.1: Desempenhos dos algoritmos paralelos homogêneo e heterogêneo com a versão paralela do WAND e as políticas de *threshold* para consultas de diferentes tamanhos e número de partições igual a 1.

Os top-k documentos retornados pelo algoritmo homogêneo combinam somente com 90,5% dos documentos retornados pelo algoritmo sequencial WAND para consultas curtas e os piores resultados são produzidos pelas consultas longas. Assim, o algoritmo paralelo homogêneo fornece uma solução aproximada. Como o algoritmo paralelo heterogêneo retorna exatamente a mesma lista dos top-k documentos retornados pelo algoritmo sequencial, seu *recall* é 1.0 e, por essa razão, não foi adicionado na Tabela 5.5.

Algor.	**Threshold**	Curto		Médio		Longo		Extralongo	
		Recall	**Des. Padrão**	**Recall**	**Des. Padrão**	**Recall**	**Des. Padrão**	**Recall**	**Des. Padrão**
Hom.	Local	0,891	0,030	0,897	0,011	0,833	0,021	0,905	0,019
Hom.	Sh-R	0,890	0,030	0,897	0,011	0,833	0,021	0,905	0,019
Hom.	Sh-WR	0,891	0,030	0,897	0,011	0,833	0,021	0,905	0,019

Tabela 5.5: *Recall* para o algoritmo paralelo homogêneo com a versão paralela do WAND para consultas OR.

As estratégias de compartilhamento do *threshold* foram mais efetivas nos ganhos computacionais dos algoritmos nos casos em que o número de partição por bloco de *threads* é maior que 1 e para consultas longas e extralongas. As Figuras 5.2 e 5.3 demonstram os gráficos de *speedup* das estratégias de compartilhamento com o número de partição igual a 10, utilizando os algoritmos homogêneo e heterogêneo, respectivamente. Nesses gráficos, pode ser observado que o compartilhamento não garantiu um ganho de desempenho para as consultas curtas e médias. A leve queda do *speedup* para essas consultas pode ter sido causada pela baixa sobrecarga de processamento das partições. Como há pouco trabalho aguardando o processamento nessas consultas por causa da quantidade de documentos, a utilização do *threshold* é realizada

em uma pequena porção das partições. Enquanto isso, o acesso à memória global da GPU para o compartilhamento do *threshold* global continua constante para todos os processadores.

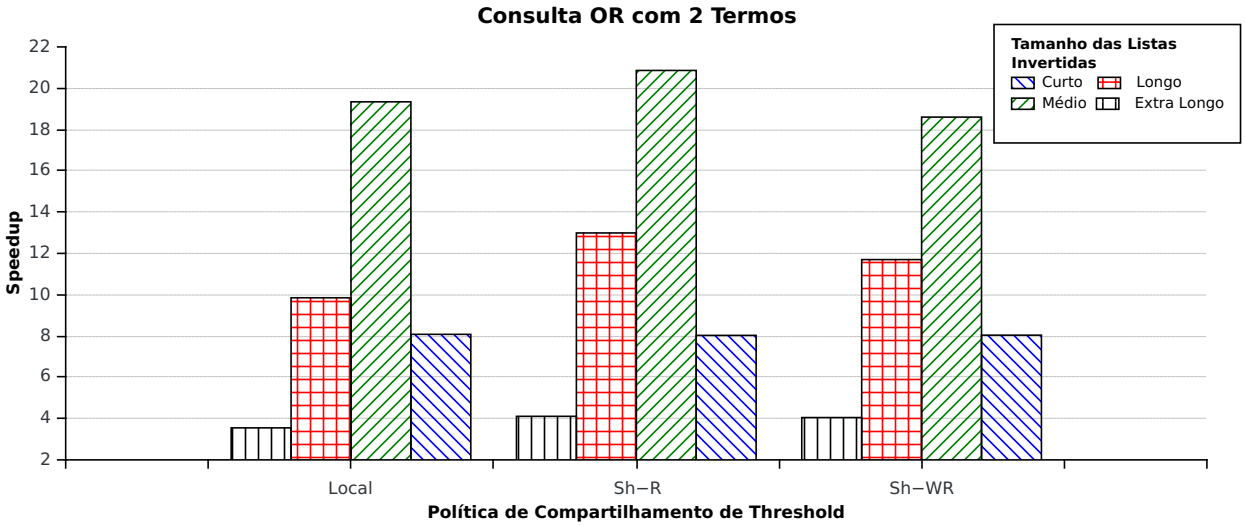

Figura 5.2: Desempenho do algoritmo paralelo homogêneo com a versão paralela do WAND e as políticas de *threshold* para consultas OR de diferentes tamanhos e número de partições igual a 10.

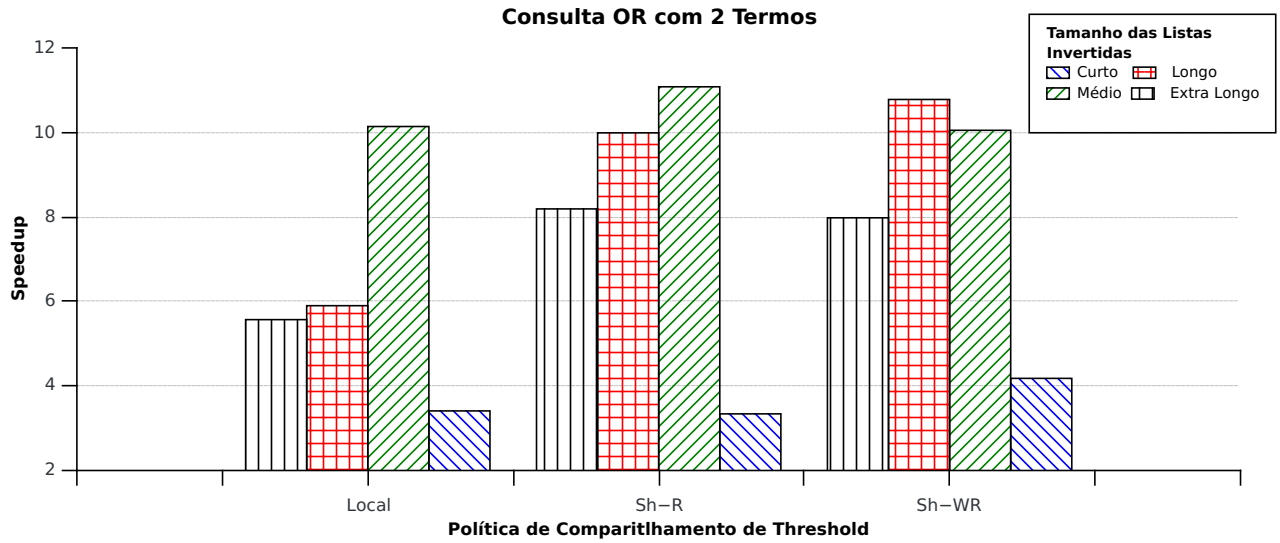

Figura 5.3: Desempenho do algoritmo paralelo heterogêneo com a versão paralela do WAND e as políticas de *threshold* para consultas OR de diferentes tamanhos e número de partições igual a 10.

Nos resultados dos primeiros experimentos, Tabela 5.4, os melhores *speedups* nas consultas longas e extralongas são encontrados nos casos das partições de tamanho 64 docIDs e/ou com o número de partições por bloco de *threads* igual a 10. Nestes casos, em que há um aumento de trabalho, o reaproveitamento dos dados leva a um melhor *speedup*. Visto isso, um segundo

conjunto de experimentos foi testado de modo que o número de partições por bloco de *threads* foi elevado até o valor de 200 partições.

As Figuras 5.4 e 5.5 mostram os gráficos de desempenho para os algoritmos homogêneo e heterogêneo, respectivamente. Como o tamanho das partições igual a 32 docIDs apresentou em média os melhores tempos de execução para ambos algoritmos, esses gráficos foram baseados nos desempenhos desse caso. Todos os algoritmos obtiveram ganho de desempenho para todos os tamanhos das listas invertidas, com exceção das consultas curtas, até o número de partições igual a 10. Após esse valor, somente as consultas extralongas (5,49 milhões docIDs) obtiveram um ganho de *speedup*. Observa-se que neste caso, inicialmente, os desempenhos ficaram próximo de 1, enquanto com o número de partições igual a 100, chegou a ultrapassar o valor de 10$\times$ de *speedup*. Isso mostra o potencial de escalabilidade das soluções frente ao crescimento das listas invertidas para o processamento de consultas OR, principalmente para as implementações utilizando as políticas de compartilhamento.

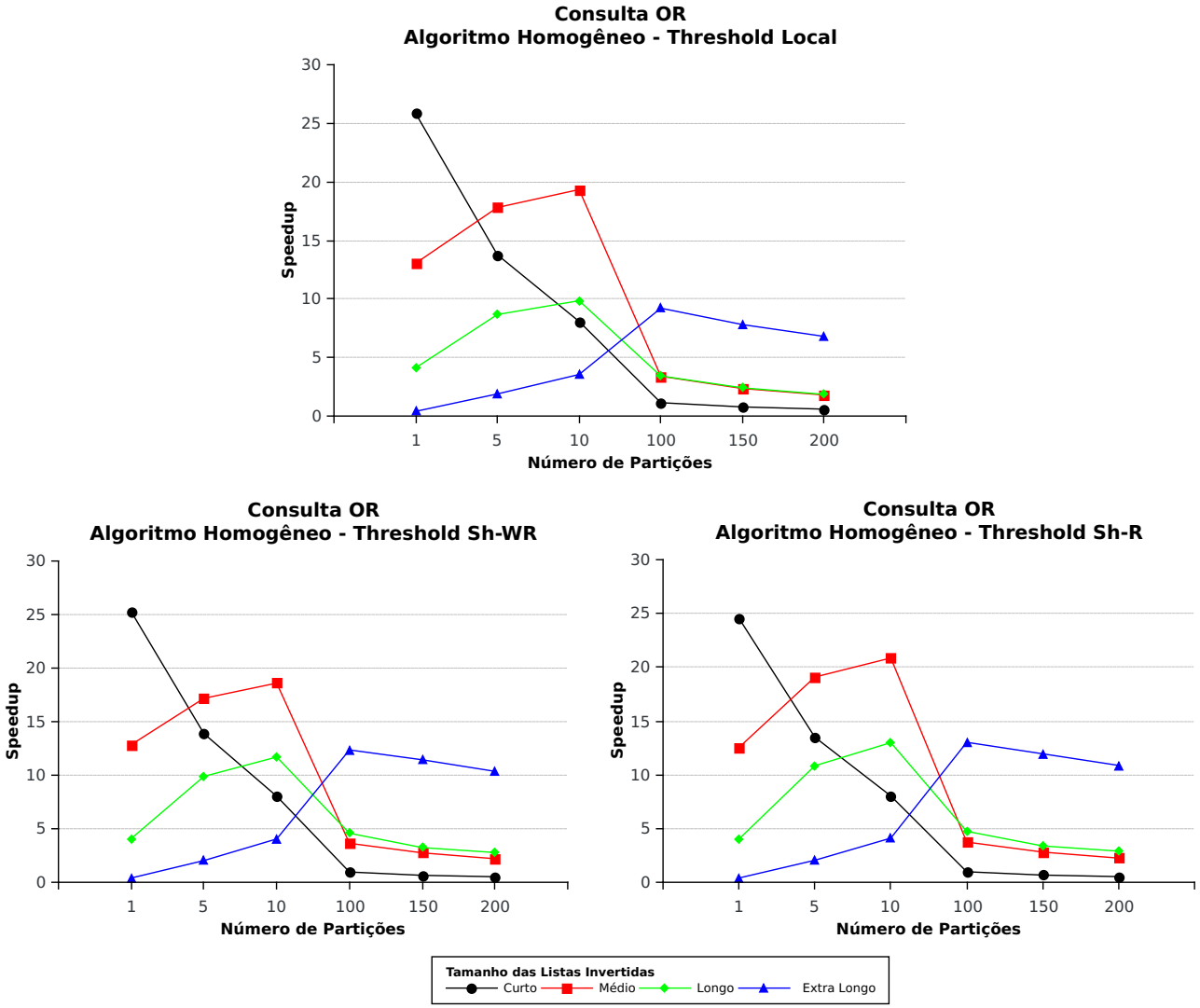

Figura 5.4: Desempenho do algoritmo paralelo homogêneo com a versão paralela do WAND e as políticas de *threshold* para consultas OR variando o número de partições.

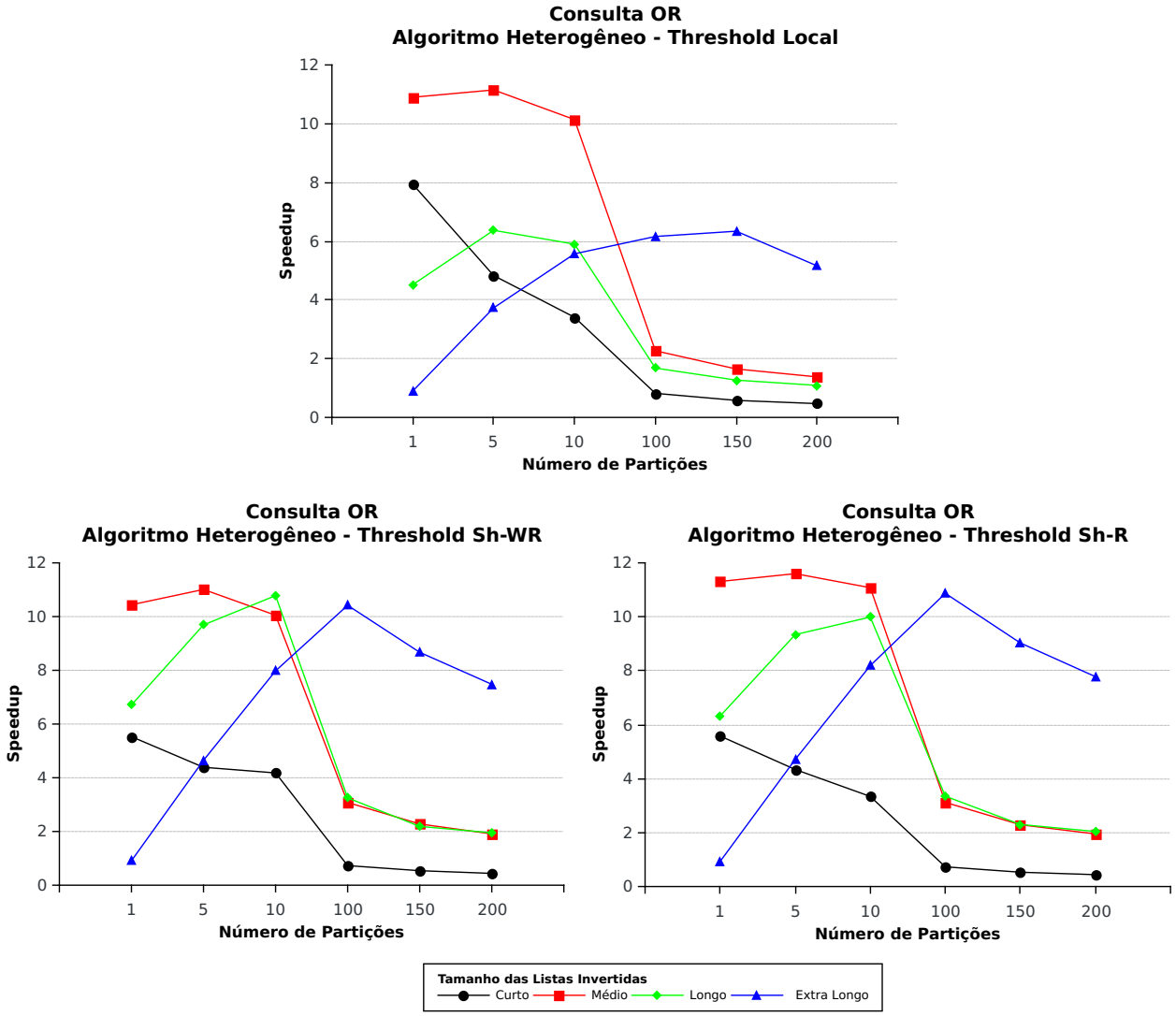

Figura 5.5: Desempenho do algoritmo paralelo heterogêneo com a versão paralela do WAND e as políticas de *threshold* para consultas OR variando o número de partições.

5.2.2 Consultas AND

A Tabela 5.6 mostra os tempos médios de execução do algoritmo sequencial WAND para as consultas AND. A mesma quantidade de documentos retornados nos experimentos OR ($k = 128$) foi realizado com as consultas AND. O algoritmo sequencial levou em média 13,83 ms para processar consultas curtas AND e 202,79 ms para consultas extralongas AND.

# de termos	Tamanho da Consulta	# médio de docIDs	Tempo (ms)
2	Curto	89.615	13,83
2	Médio	476.771	42,43
2	Longo	1.032.795	45,44
2	Extra	5.494.285	200,77

Tabela 5.6: Tempo médio de execução (milissegundos) do algoritmo sequencial WAND para consultas AND.

Os tempos apresentados pelos algoritmos paralelos, homogêneo e heterogêneo, para consultas AND são mostrados na Tabela 5.7, onde o tamanho da partição foi variado entre 32 e 64; e o número de partições por bloco de *threads* foi variado entre 1 e 10. O desempenho obtido pelo algoritmo homogêneo foi de 40× de *speedup* para consultas curtas, enquanto o heterogêneo alcançou 18× para as mesmas consultas. O ganho de *speedup* foi maior para consultas AND comparado com os obtidos pelo algoritmo para consultas OR, pois o algoritmo paralelo respondeu às consultas de forma mais eficiente do que o algoritmo sequencial. Por exemplo, para consultas curtas o algoritmo paralelo teve uma melhora de $\sim 35\%$.

Algor.	Thres-hold	Tamanho da Part.	# de Part.	Curto		Médio		Longo		Extra	
				Tempo	Speedup	Tempo	Speedup	Tempo	Speedup	Tempo	Speedup
Hom.	Local	32	1	0,34	**40,73**	2,39	17,74	9,11	4,99	503,78	0,40
Hom.	Local	32	10	1,59	8,70	2,05	20,69	3,79	**11,99**	53,67	3,74
Hom.	Local	64	1	0,46	30,23	1,80	**23,61**	5,54	8,20	264,82	0,76
Hom.	Local	64	10	2,79	4,95	3,27	12,99	4,08	11,14	31,18	**6,44**
Hom.	Sh-R	32	1	0,36	**38,64**	2,44	17,37	9,20	4,94	502,65	0,40
Hom.	Sh-R	32	10	1,74	7,97	1,96	21,61	3,12	**14,57**	49,34	4,07
Hom.	Sh-R	64	1	0,49	28,31	1,88	**22,61**	5,65	8,04	259,13	0,77
Hom.	Sh-R	64	10	3,17	4,36	3,18	13,35	3,22	14,12	27,32	**7,35**
Hom.	Sh-WR	32	1	0,35	**39,59**	2,41	17,63	9,16	4,96	499,57	0,40
Hom.	Sh-WR	32	10	1,76	7,87	2,06	20,63	3,28	**13,86**	50,53	3,97
Hom.	Sh-WR	64	1	0,49	28,05	1,84	**23,10**	5,58	8,14	264,19	0,76
Hom.	Sh-WR	64	10	3,23	4,29	3,25	13,04	3,43	13,25	27,81	**7,22**
Het.	Local	32	1	0,77	**18,04**	2,90	14,62	8,23	5,52	217,83	0,92
Het.	Local	32	10	2,46	5,62	4,35	9,75	6,88	6,61	33,91	5,92
Het.	Local	64	1	1,17	11,86	2,71	**15,68**	6,23	**7,30**	119,35	1,68
Het.	Local	64	10	4,09	3,38	6,12	6,94	8,90	5,11	30,30	**6,63**
Het.	Sh-R	32	1	0,85	**16,23**	2,90	14,64	8,43	5,39	217,66	0,92
Het.	Sh-R	32	10	2,83	4,89	4,07	10,43	6,70	**6,78**	25,00	8,03
Het.	Sh-R	64	1	1,29	10,69	2,68	**15,86**	7,11	6,39	116,10	1,73
Het.	Sh-R	64	10	4,77	2,90	5,57	7,62	8,56	5,31	19,89	**10,09**
Het.	Sh-WR	32	1	0,85	**16,18**	2,97	14,27	8,72	5,21	216,24	0,93
Het.	Sh-WR	32	10	2,82	4,91	4,13	10,27	5,68	**7,99**	25,95	7,74
Het.	Sh-WR	64	1	1,25	11,02	2,81	**15,09**	7,55	6,02	117,87	1,70
Het.	Sh-WR	64	10	4,69	2,95	5,70	7,45	8,67	5,24	20,19	**9,95**

Tabela 5.7: Resultado do Desempenho para consultas AND com 2 termos. As colunas de 'Tempo' trazem o tempo de execução em milissegundos para cada teste.

O valor inicial do *threshold* no processamento de consultas AND é a soma dos *upper bounds* dos termos da consulta. Então, no início, cada processador já tem um valor de *threshold* alto

suficiente para ignorar documentos com probabilidade baixa de estar entre os top-k documentos mais relevantes. Nas consultas curtas, embora o alto valor do *threshold* juntamente com o balanceamento de carga fizeram com que o algoritmo homogêneo alcançasse o dobro de *speedup* frente à sua versão heterogênea, o algoritmo manteve o *recall* em 90%, um resultado próximo ao da consulta OR. A Tabela 5.8 detalha a acurácia do algoritmo homogêneo nas consultas AND.

Algor.	**Threshold**	Curto		Médio		Longo		Extra Longo	
		Recall	**Des. Padrão**	**Recall**	**Des. Padrão**	**Recall**	**Des. Padrão**	**Recall**	**Des. Padrão**
Hom.	Local	0,911	0,012	0,907	0,011	0,821	0,016	0,812	0,015
Hom.	Sh-R	0,891	0,013	0,887	0,021	0,821	0,016	0,812	0,015
Hom.	Sh-WR	0,911	0,012	0,857	0,019	0,821	0,016	0,812	0,015

Tabela 5.8: *Recall* para o algoritmo paralelo homogêneo com a versão paralela do WAND para consultas AND.

Os desempenhos dos algoritmos com as diferentes políticas de propagação com uma partição (tamanho da partição igual a 32) por bloco de *threads* são detalhados no gráfico mostrado na Figura 5.6. Como neste caso não há reproveitamento de dados e pouca propagação de *threshold*, os algoritmos apresentaram baixo *speedup* frente ao aumento de documentos nas consultas longas e extralongas.

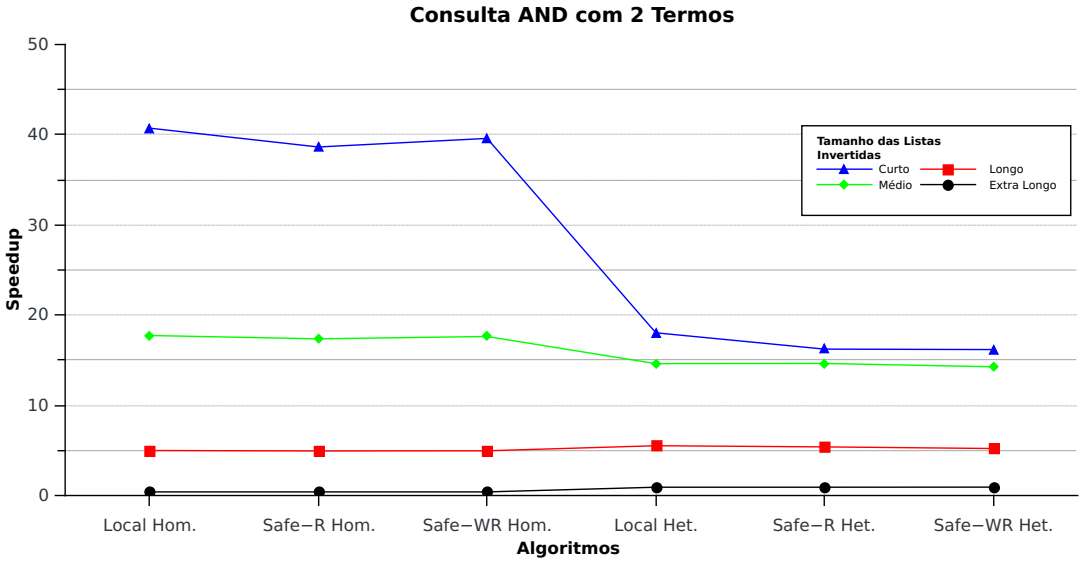

Figura 5.6: Desempenhos dos algoritmos paralelos homogêneo e heterogêneo com a versão paralela do WAND e as políticas de *threshold* para consultas AND de diferentes tamanhos e número de partições igual a 1.

O gráfico da Figura 5.7 mostra o comportamento do algoritmo homogêneo com as políticas de propagação e 10 partições por bloco de *threads*, sendo que o tamanho das partições é igual

a 32, para consultas AND. Com o valor inicial alto do *threshold*, verifica-se que o algoritmo comportou-se de forma similar nas diferentes políticas de propagação. O mesmo se pode verificar com o algoritmo heterogêneo na Figura 5.8. Logo, pode-se afirmar que as implementações das políticas de compartilhamento de *threshold* têm baixo impacto no desempenho da proposta paralela utilizando a versão WAND para consultas AND, com ligeiro ganho em favor da política *Sh-R*.

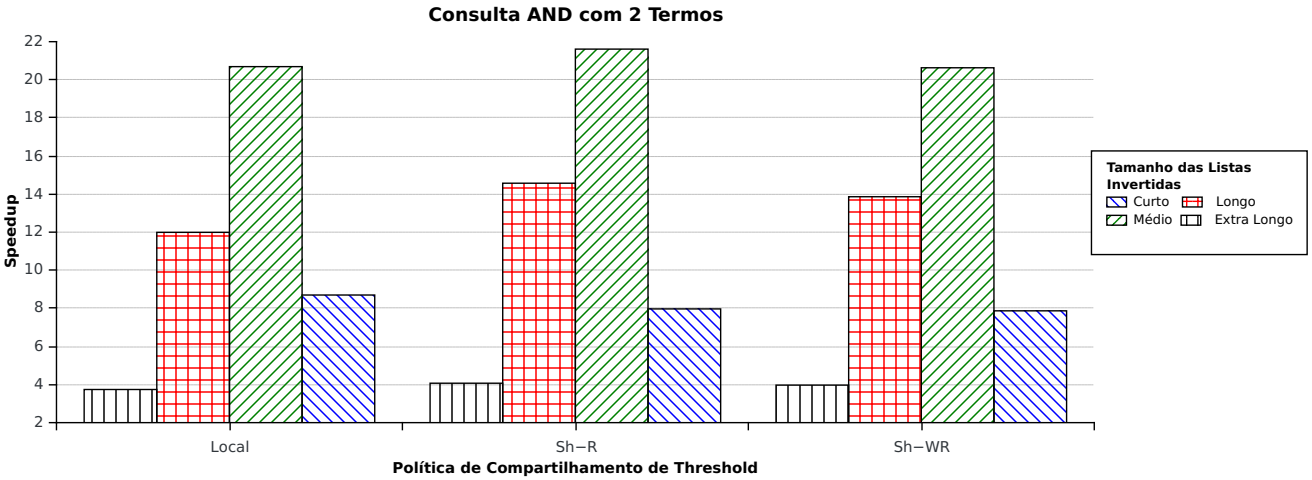

Figura 5.7: Desempenho do algoritmo paralelo homogêneo com a versão paralela no WAND e as políticas de *threshold* para consultas AND de diferentes tamanhos e número de partições igual a 10.

Um fator que se destaca nos dois algoritmos, homogêneo e heterogêneo, nos casos do número baixo de partições ($\sim$ 10) por bloco de *threads* é o desempenho deles sobre consultas extralongas, tanto para consultas OR quanto para AND. O ganho computacional é maior nesse tipo de consulta para os algoritmos heterogêneos, o que não acontece para o restante das consultas. Esse fator pode ser observado nos gráficos apresentados nas Figuras 5.2, 5.3, 5.7 e 5.8. A classificação completa dos documentos na estratégia heterogênea é um possível fator que esteja influenciando a ocorrência desse fato, pois ao classificar todas as ocorrências dos documentos o valor do *threshold* tende a crescer mais rapidamente do que em uma classificação parcial dos documentos.

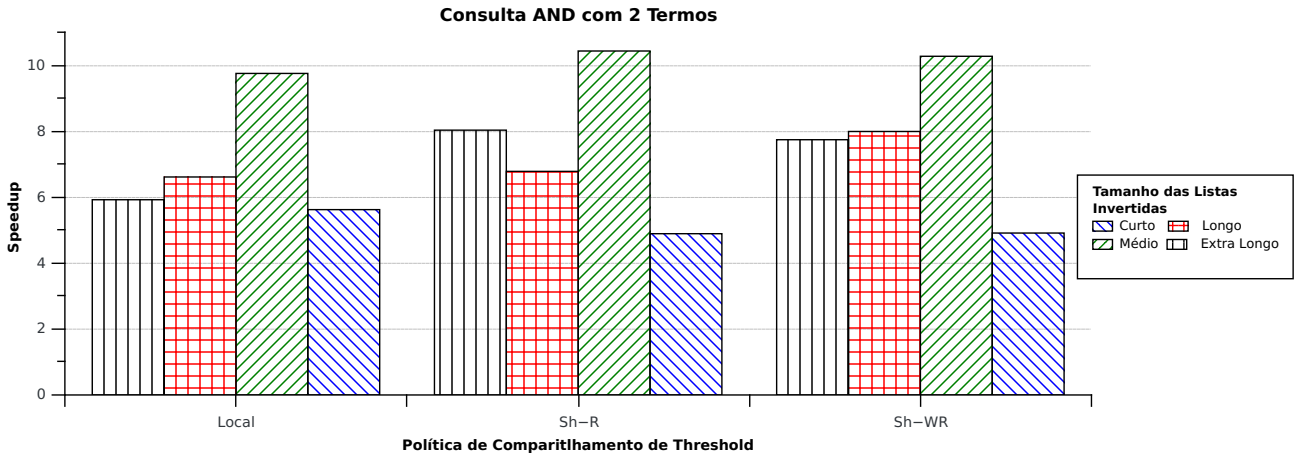

Figura 5.8: Desempenho do algoritmo paralelo heterogêneo com a versão paralela do WAND e as políticas de *threshold* para consultas AND de diferentes tamanhos e número de partições igual a 10.

Na segunda parte dos experimentos, os algoritmos apresentaram um aumento significativo de desempenho nas consultas extralongas para todas as políticas de compartilhamento de *threshold*, processando 100 partições por bloco de *threads*. Por outro lado, no restante das consultas, os algoritmos apresentaram quedas no *speedup* na mesma quantidade de partições, como pode ser visualizado nos gráficos das Figuras 5.9 e 5.10.

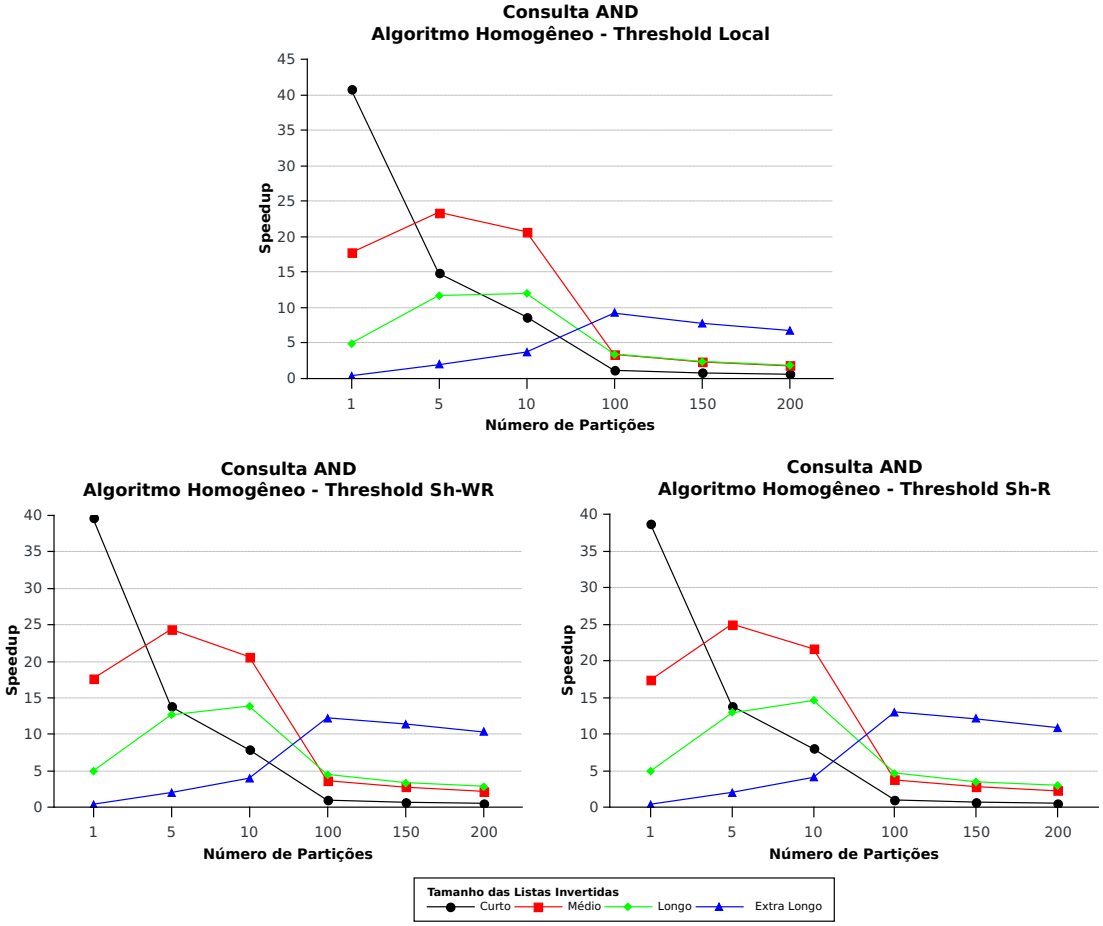

Figura 5.9: Desempenho do algoritmo paralelo homogêneo com a versão paralela do WAND e as políticas de *threshold* para consultas AND variando o número de partições.

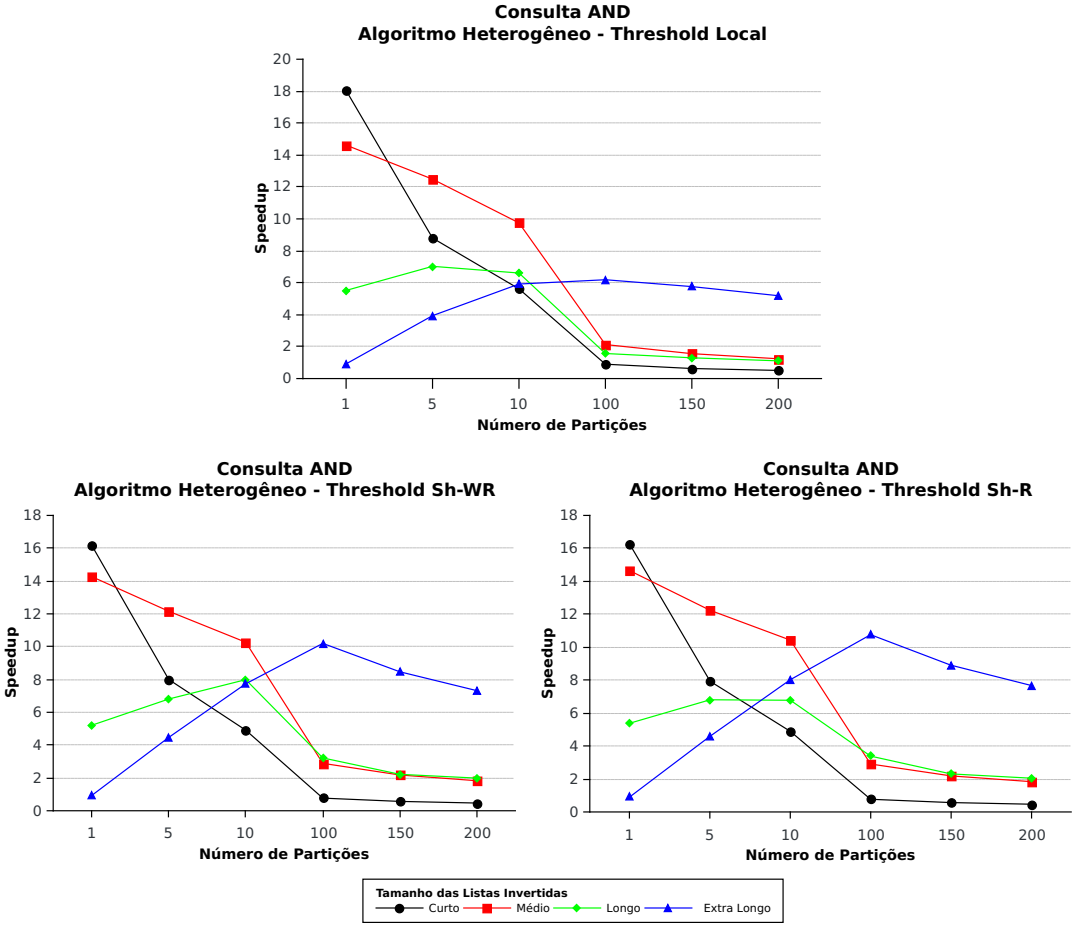

Figura 5.10: Desempenho do algoritmo paralelo heterogêneo com a versão paralela do WAND e as políticas de *threshold* para consultas AND variando o número de partições.

5.3 Avaliação do Algoritmo Paralelo MaxScore

Nesta seção, avaliamos o desempenho das estratégias propostas aplicadas à versão paralela do algoritmo MaxScore descrito no Capítulo 4.

5.3.1 Consultas OR

Os tempos de processamento (ms) de consultas OR do algoritmo sequencial MaxScore são mostrados na Tabela 5.9. O algoritmo levou 6,26 ms para processar $\sim 89,61$ mil documentos de uma consulta, enquanto para processar $\sim 5,49$ milhões de documentos gastou 334,31 ms.

# de termos	**Tamanho da Consulta**	**# médio de docIDs**	**Tempo (ms)**
2	Curto	89.615	6,26
2	Médio	476.771	33,33
2	Longo	1.032.795	69,18
2	Extra	5.494.285	334,31

Tabela 5.9: Tempo médio de execução (ms) do algoritmo sequencial MaxScore para consultas OR.

A Tabela 5.10 mostra os tempos de execução para consultas OR do algoritmo paralelo Maxscore, onde a quantidade de partições por blocos de *threads* foi variada entre 1 e 10. Os melhores resultados de cada caso foram destacados em negrito. O maior *speedup* obtido nas consultas curtas foi de $13,86\times$ pelo algoritmo homogêneo, enquanto nas consultas extralongas o maior *speedup* alcançado foi de $15,34\times$ pelo algoritmo heterogêneo.

Algor.	**Thres-hold**	**Tamanho da Part.**	**# de Part.**	Curto		Médio		Longo		Extra	
				Tempo	**Speedup**	**Tempo**	**Speedup**	**Tempo**	**Speedup**	**Tempo**	**Speedup**
Hom.	Local	32	1	0,47	13,19	3,04	10,95	10,55	6,56	504,82	0,66
Hom.	Local	32	10	1,54	4,07	2,00	**16,68**	4,34	15,93	55,73	6,00
Hom.	Local	64	1	0,45	**13,86**	2,29	14,54	6,89	10,04	266,48	1,25
Hom.	Local	64	10	2,84	2,21	3,22	10,35	4,11	**16,85**	34,32	**9,74**
Hom.	Sh-R	32	1	0,51	12,34	3,15	10,57	10,77	6,42	499,76	0,67
Hom.	Sh-R	32	10	1,72	3,64	2,06	**16,18**	3,90	17,75	49,05	6,82
Hom.	Sh-R	64	1	0,48	**12,96**	2,40	13,87	7,12	9,72	266,75	1,25
Hom.	Sh-R	64	10	3,24	1,94	3,14	10,63	3,54	**19,56**	27,98	**11,95**
Hom.	Sh-WR	32	1	0,49	12,71	3,11	10,73	10,67	6,48	504,09	0,66
Hom.	Sh-WR	32	10	1,74	3,60	2,28	**14,61**	4,36	15,87	50,34	6,64
Hom.	Sh-WR	64	1	0,47	**13,41**	2,36	14,14	7,02	9,85	271,61	1,23
Hom.	Sh-WR	64	10	3,29	1,90	3,23	10,31	4,03	**17,15**	28,61	**11,69**
Het.	Local	32	1	2,23	**2,81**	3,98	8,37	10,96	6,31	220,79	1,51
Het.	Local	32	10	3,71	1,69	4,50	7,41	8,47	**8,17**	35,43	9,44
Het.	Local	64	1	2,46	2,54	3,78	**8,81**	9,72	7,12	121,62	2,75
Het.	Local	64	10	5,46	1,15	5,97	5,59	13,79	5,02	32,11	**10,41**
Het.	Sh-R	32	1	2,54	**2,47**	3,88	8,58	9,62	7,19	217,52	1,54
Het.	Sh-R	32	10	4,27	1,47	5,02	6,63	6,54	10,58	28,14	11,88
Het.	Sh-R	64	1	2,71	2,31	3,58	**9,32**	7,82	8,85	116,84	2,86
Het.	Sh-R	64	10	6,15	1,02	5,53	6,03	6,41	**10,80**	21,80	**15,34**
Het.	Sh-WR	32	1	2,47	**2,54**	3,95	8,44	8,95	7,73	220,20	1,52
Het.	Sh-WR	32	10	4,40	1,42	5,43	6,14	6,02	**11,49**	28,73	11,63
Het.	Sh-WR	64	1	2,57	2,44	3,74	**8,91**	6,87	10,08	118,87	2,81
Het.	Sh-WR	64	10	6,13	1,02	7,49	4,45	6,97	9,93	22,71	**14,72**

Tabela 5.10: Desempenho do algoritmo paralelo MaxScore para consultas OR com 2 termos. As colunas de 'Tempo' trazem o tempo de execução em milissegundos para cada teste.

Nos experimentos com 1 partição por bloco de *threads*, os algoritmos homogêneo e heterogêneos obtiveram resultados próximos nas consultas médias, longas e extralongas, como pode ser observado na Figura 5.11. Nestes casos, o desbalanceamento de carga do algoritmo heterogêneo com MaxScore não degradou os *speedups*.

O *recall* do algoritmo homogêneo com MaxScore para consultas OR é apresentado na Ta-

bela 5.11. O algoritmo apresentou um *recall* crescente de acordo com o aumento dos tamanhos da consulta. Para consultas extralongas, o algoritmo alcançou um *recall* de $\sim 85,5\%$, enquanto para consultas curtas obteve o menor *recall*, $\sim 74\%$.

Algor.	**Threshold**	Curto		Médio		Longo		Extralongo	
		Recall	**Des. Padrão**	**Recall**	**Des. Padrão**	**Recall**	**Des. Padrão**	**Recall**	**Des. Padrão**
Hom.	Local	0,749	0,020	0,781	0,021	0,815	0,031	0,855	0,033
Hom.	Sh-R	0,749	0,020	0,781	0,021	0,815	0,031	0,855	0,033
Hom.	Sh-WR	0,743	0,029	0,777	0,029	0,812	0,032	0,809	0,035

Tabela 5.11: *Recall* para o algoritmo paralelo homogêneo com a versão paralela MaxScore para consultas OR.

As implementações das políticas de compartilhamento do *threshold* não mostraram impacto nos *speedups* quando comparadas com as implementações locais dos mesmos algoritmos no caso de uma partição por bloco de *threads*, casos em que não há reaproveitamento de dados pelos processadores. Por outro lado, nas situações em que há mais partições por bloco, as políticas apresentaram destaque na eficiência dos algoritmos nas consultas longas e extralongas. A Figura 5.12 retrata esse ganho de desempenho com 10 partições por bloco de *threads* para cada tipo de consulta. Nas consultas extralongas, o algoritmo heterogêneo da política *Sh-R* obteve 40% a mais de *speedup* comparado com o *speedup* obtido nas mesmas condições com a política local. Também nessa situação, o algoritmo heterogêneo para consultas extralongas obteve um *speedup* maior do que sua versão homogênea, o mesmo ocorrido com a implementação paralela do WAND.

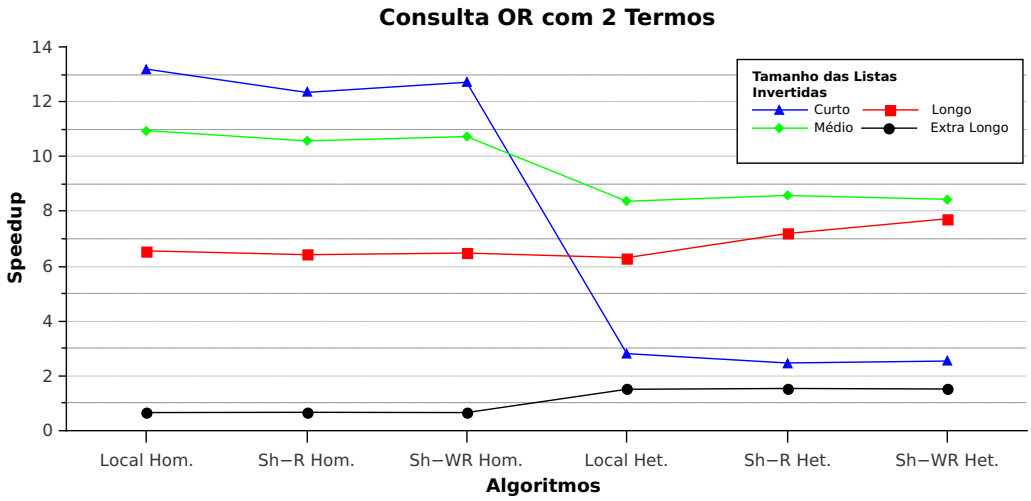

Figura 5.11: Desempenho dos algoritmos paralelos com a versão paralela do MaxScore e 1 partição por bloco de *threads*.

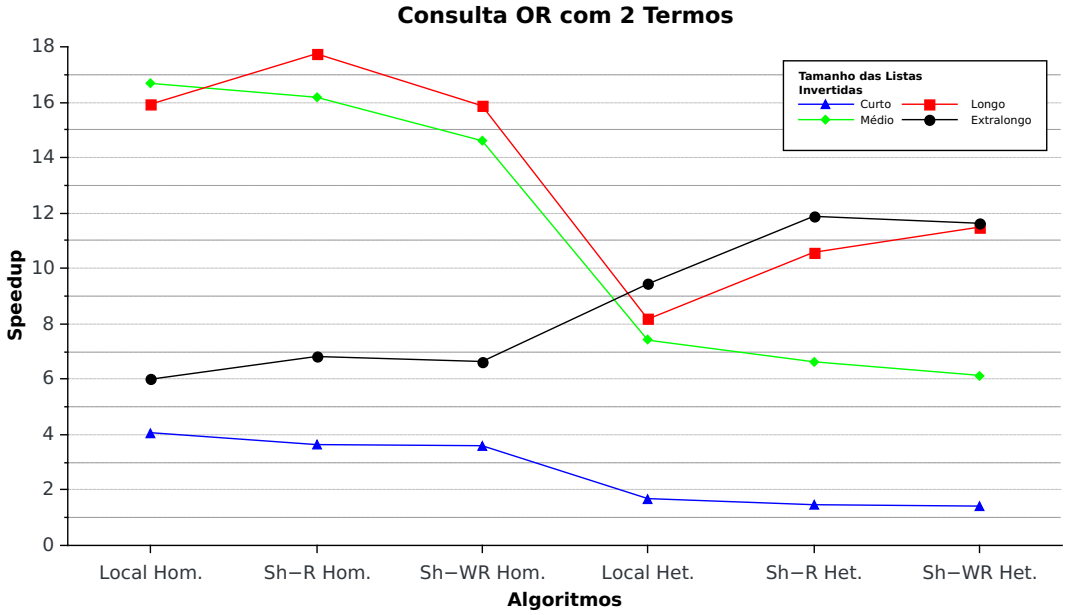

Figura 5.12: Desempenho dos algoritmos paralelos com a versão paralela do MaxScore e 10 partições por bloco de *threads*.

Os gráficos apresentados nas Figuras 5.13 e 5.14 mostram o desempenho das estratégias quando aplicadas ao algoritmo paralelo MaxScore, homogêneo e heterogêneo respectivamente, com o aumento de trabalho por bloco de *threads* para consultas OR. O processamento de consultas curtas é efetivo com uma única partição por bloco de *threads*, enquanto para médias e longas o número de 10 partições por bloco apresentou mais efetividade. Ao ampliar em $18,7\%$ o número de documentos das consultas longas, consultas extralongas, alcança-se um desempenho superior com 100 partições por bloco.

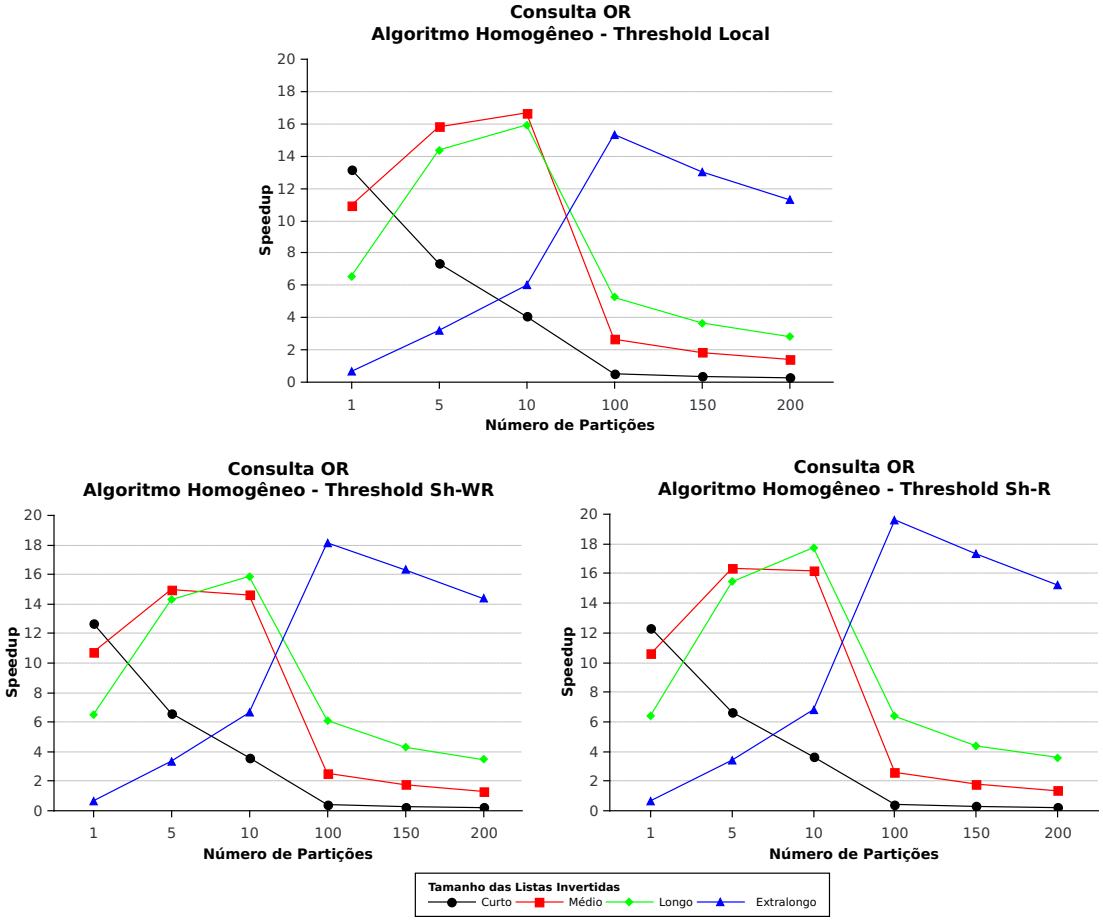

Figura 5.13: Desempenho do algoritmo paralelo homogêneo com a versão paralela do MaxScore e as políticas de *threshold* para consultas OR variando o número de partições.

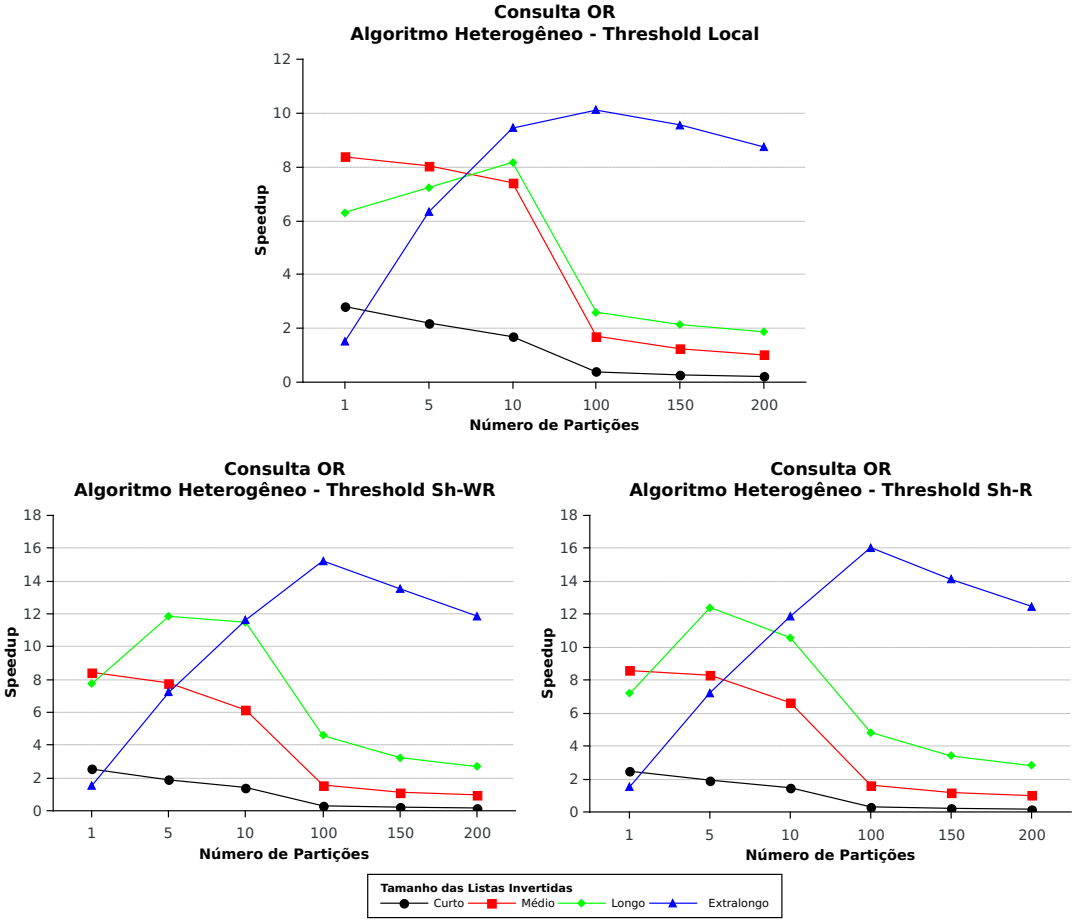

Figura 5.14: Desempenho do algoritmo paralelo heterogêneo com a versão paralela do MaxScore e as políticas de *threshold* para consultas OR variando o número de partições.

As políticas de propagação *Sh-R* e *Sh-WR* no algoritmo heterogêneo tiveram um desempenho superior nas consultas extralongas OR quando comparadas com a implementação local. Na quantidade de 100 partições por bloco de *threads*, as políticas de propagação chegaram a representar $\sim 60\%$ a mais de *speedup* em relação à implementação básica local. A implementação da política *Sh-R* apresentou um melhor *speedup* para as quantidades superiores a 10, como pode ser observado na Figura 5.15.

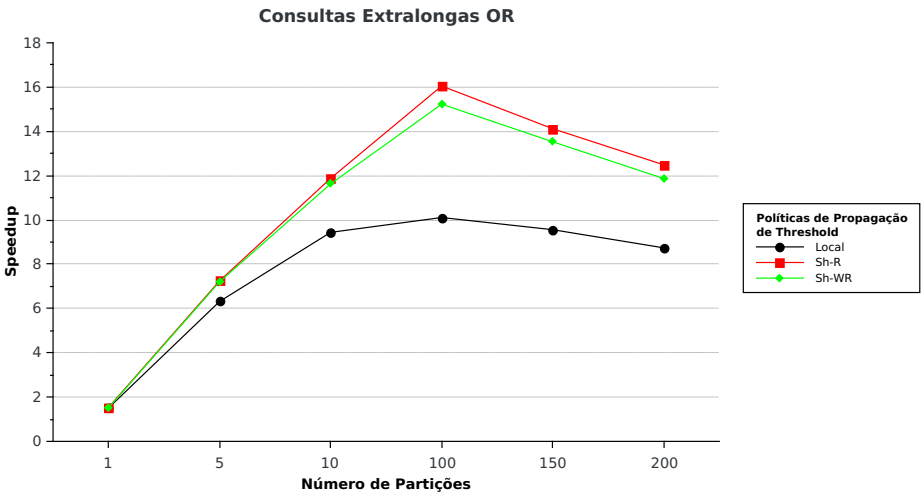

Figura 5.15: Desempenho das políticas de propagação de *threshold* no algoritmo paralelo heterogêneo com a versão paralela do MaxScore para consultas extralongas OR.

5.3.2 Consultas AND

O algoritmo sequencial MaxScore conseguiu os tempos de execução (ms) contidos na Tabela 5.12 para consultas AND. Os tempos alcançados nas consultas AND ficaram próximos das consultas OR para todas as quantidades de documentos por consulta. Para as curtas, o algoritmo levou 6,31 ms e 331,06 ms para as extralongas.

# de termos	Tamanho da Consulta	# médio de docIDs	Tempo (ms)
2	Curto	89.615	6,31
2	Médio	476.771	33,43
2	Longo	1.032.795	70,08
2	Extra	5.494.285	331,06

Tabela 5.12: Tempo médio de execução (ms) do algoritmo sequencial MaxScore para consultas AND.

O algoritmo paralelo teve um comportamento melhor para consultas AND, quando comparado com os resultados obtidos para consultas OR. Por exemplo, o heterogêneo alcançou um *speedup* de $\sim 3,38\times$ para consultas curtas e um de $\sim 14,68\times$ para consultas longas. Por causa de o algoritmo sequencial ter mantido o tempo de execução para consultas AND, os *speedups* médios do algoritmo paralelo com MaxScore para consultas AND aumentaram, como apresentado na Tabela 5.13. Portanto, o algoritmo paralelo demonstrou-se mais eficiente para consultas do tipo AND.

O algoritmo paralelo com MaxScore apresentou os *recalls* detalhados na Tabela 5.14 para

Algor.	**Thres-hold**	**Tamanho da Part.**	**# da Part.**	Curto		Médio		Longo		Extra	
				Tempo	**Speedup**	**Tempo**	**Speedup**	**Tempo**	**Speedup**	**Tempo**	**Speedup**
Hom.	Local	32	1	0,34	**18,43**	2,38	14,06	9,17	7,64	495,99	0,67
Hom.	Local	32	10	1,46	4,30	1,88	17,79	3,49	**20,10**	51,16	6,47
Hom.	Local	64	1	0,42	15,09	1,74	**19,22**	5,56	12,60	264,12	1,25
Hom.	Local	64	10	2,67	2,36	3,11	10,76	3,88	18,07	29,66	**11,16**
Hom.	Sh-R	32	1	0,36	**17,30**	2,43	13,74	9,28	7,55	493,43	0,67
Hom.	Sh-R	32	10	1,65	3,83	1,95	17,17	3,27	**21,40**	49,23	6,72
Hom.	Sh-R	64	1	0,45	14,06	1,82	**18,32**	5,69	12,31	263,73	1,26
Hom.	Sh-R	64	10	3,07	2,05	3,20	10,44	3,47	20,17	28,10	**11,78**
Hom.	Sh-WR	32	1	0,35	**17,90**	2,40	13,91	9,21	7,61	497,32	0,67
Hom.	Sh-WR	32	10	1,67	3,78	1,99	16,80	3,30	**21,24**	49,52	6,69
Hom.	Sh-WR	64	1	0,43	14,60	1,78	**18,75**	5,63	12,45	265,75	1,25
Hom.	Sh-WR	64	10	3,13	2,02	3,22	10,39	3,60	19,45	27,62	**11,98**
Het.	Local	32	1	1,87	**3,38**	3,11	10,75	8,71	8,04	215,78	1,53
Het.	Local	32	10	2,91	2,17	4,44	7,53	9,16	7,65	33,66	9,84
Het.	Local	64	1	2,05	3,07	2,90	**11,53**	7,84	**8,94**	116,46	2,84
Het.	Local	64	10	4,44	1,42	6,64	5,04	11,97	5,85	31,21	**10,61**
Het.	Sh-R	32	1	2,16	**2,91**	3,29	10,15	9,04	7,75	213,17	1,55
Het.	Sh-R	32	10	3,36	1,88	4,61	7,25	8,15	8,60	27,96	11,84
Het.	Sh-R	64	1	2,28	2,76	3,04	**10,99**	7,90	**8,88**	114,31	2,90
Het.	Sh-R	64	10	4,98	1,27	6,64	5,04	9,15	7,66	22,19	**14,92**
Het.	Sh-WR	32	1	2,04	**3,09**	3,23	10,34	9,07	7,73	214,88	1,54
Het.	Sh-WR	32	10	3,28	1,93	4,68	7,15	6,47	**10,82**	28,38	11,66
Het.	Sh-WR	64	1	2,16	2,92	3,06	**10,92**	8,08	8,67	117,21	2,82
Het.	Sh-WR	64	10	4,97	1,27	7,04	4,75	7,84	8,94	22,55	**14,68**

Tabela 5.13: Desempenho do algoritmo paralelo MaxScore para consultas AND com 2 termos. As colunas de 'Tempo' trazem o tempo de execução em milissegundos para cada teste.

consultas AND. O algoritmo manteve seu recall em $\sim 78\%$ com o aumento do tamanho das listas invertidas. O melhor caso foi apresentado para consultas extralongas, em que alcançou um *recall* de $\sim 78\%$.

Algor.	**Threshold**	Curto		Médio		Longo		Extralongo	
		Recall	**Des. Padrão**	**Recall**	**Des. Padrão**	**Recall**	**Des. Padrão**	**Recall**	**Des. Padrão**
Hom.	Local	0,777	0,034	0,783	0,032	0,781	0,031	0,789	0,031
Hom.	Sh-R	0,777	0,034	0,783	0,032	0,781	0,031	0,789	0,031
Hom.	Sh-WR	0,732	0,038	0,781	0,039	0,780	0,039	0,789	0,032

Tabela 5.14: *Recall* para o algoritmo paralelo homogêneo com a versão paralela do MaxScore para consultas AND.

Para uma única partição por bloco de *threads*, os ganhos computacionais dos algoritmos paralelos com MaxScore se mantiveram no tamanho longo e extralongo das listas invertidas, enquanto uma queda de *speedup* ocorreu para os tamanhos curto e médio de homogêneo para heterogêneo. A Figura 5.16 constata isso por meio da ilustração do gráfico de desempenho dos algoritmos homogêneo e heterogêneo com uma única partição por bloco de *threads*. Para essa quantidade de partições por bloco, as estratégias de propagação do *threshold* não se destacaram com ganho de *speedup* comparado com a implementação local do *threshold*.

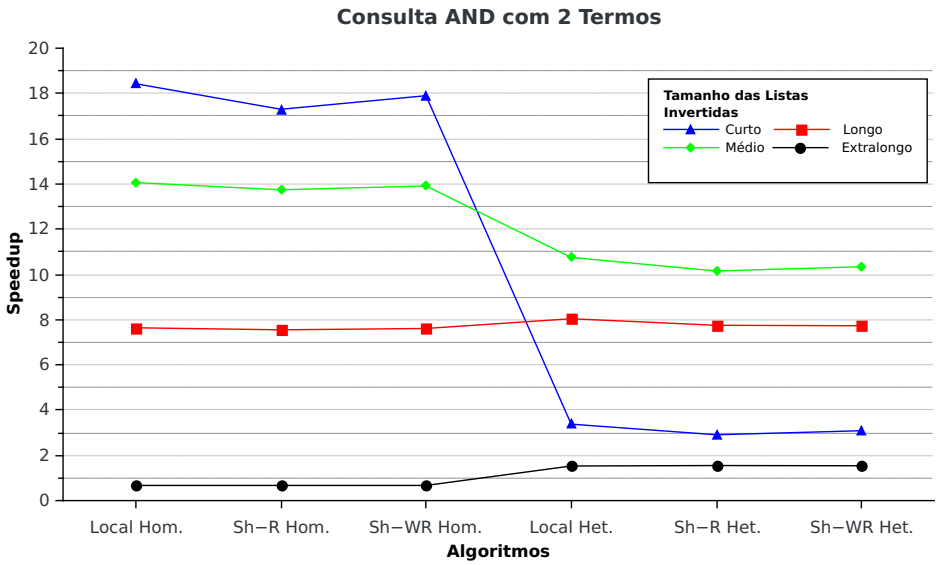

Figura 5.16: Desempenho dos algoritmos paralelos com a versão paralela do MaxScore e 1 partição por bloco de *threads* para consultas AND.

O algoritmo homogêneo com 10 partições por bloco de *threads* alcançou nas consultas longas um *speedup* três vezes maior do que com uma partição por bloco. Já para o mesmo tipo de consulta, o algoritmo heterogêneo manteve seu *speedup*. Um comportamento semelhante foi verificado para consultas médias, como demonstrado na Figura 5.17. Com o aumento do número de partições por bloco, os algoritmos tiveram um aproveitamento melhor para as consultas longas e extralongas, considerando que o *speedup* chegou a alcançar um desempenho de $\sim 12\times$ para o algoritmo heterogêneo.

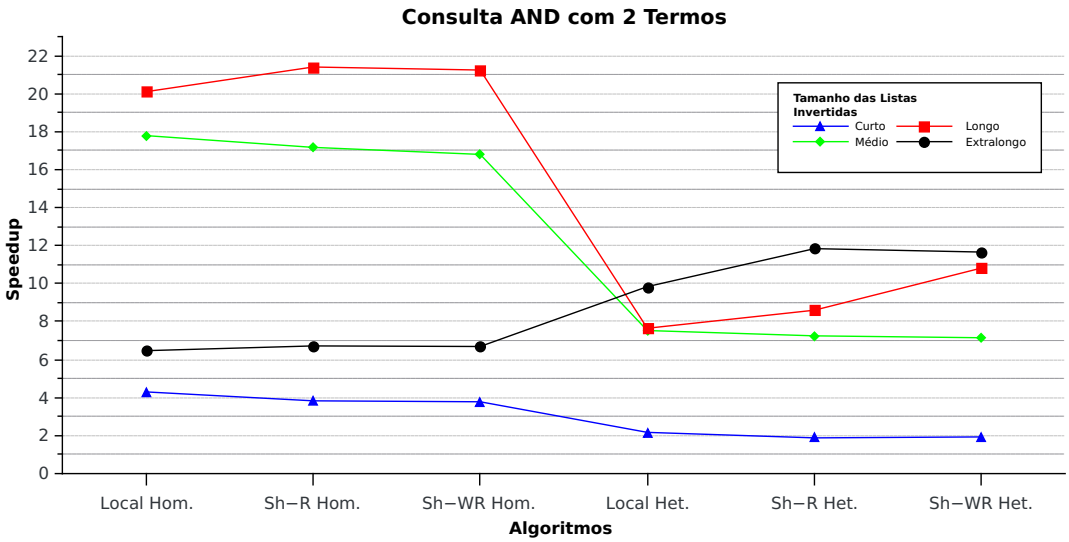

Figura 5.17: Desempenho dos algoritmos paralelos com a versão paralela do MaxScore e 10 partição por bloco de *thread* para consultas AND.

Os gráficos das Figuras 5.18 e 5.19 mostram o desempenho dos algoritmos homogêneo e heterogêneo, respectivamente, com as diferentes estratégias de propagação do *threshold* com o aumento da quantidade de processadores. O melhor cenário para as consultas curtas AND é uma partição por bloco de *threads* em todos os algoritmos. Para as consultas médias, os casos de 5 partições por bloco apresentam os melhores *speedup* para o algoritmo homogêneo, enquanto uma partição por bloco oferece o melhor *speedup* para o heterogêneo. Com 10 partições por bloco, os algoritmos alcançam o melhor *speedup* para as estratégias de propagação. As Figuras 5.18 e 5.19 mostram que o ganho computacional nas consultas extralongas é proporcional ao aumento do número de partições por bloco com o limite de 10 partições.

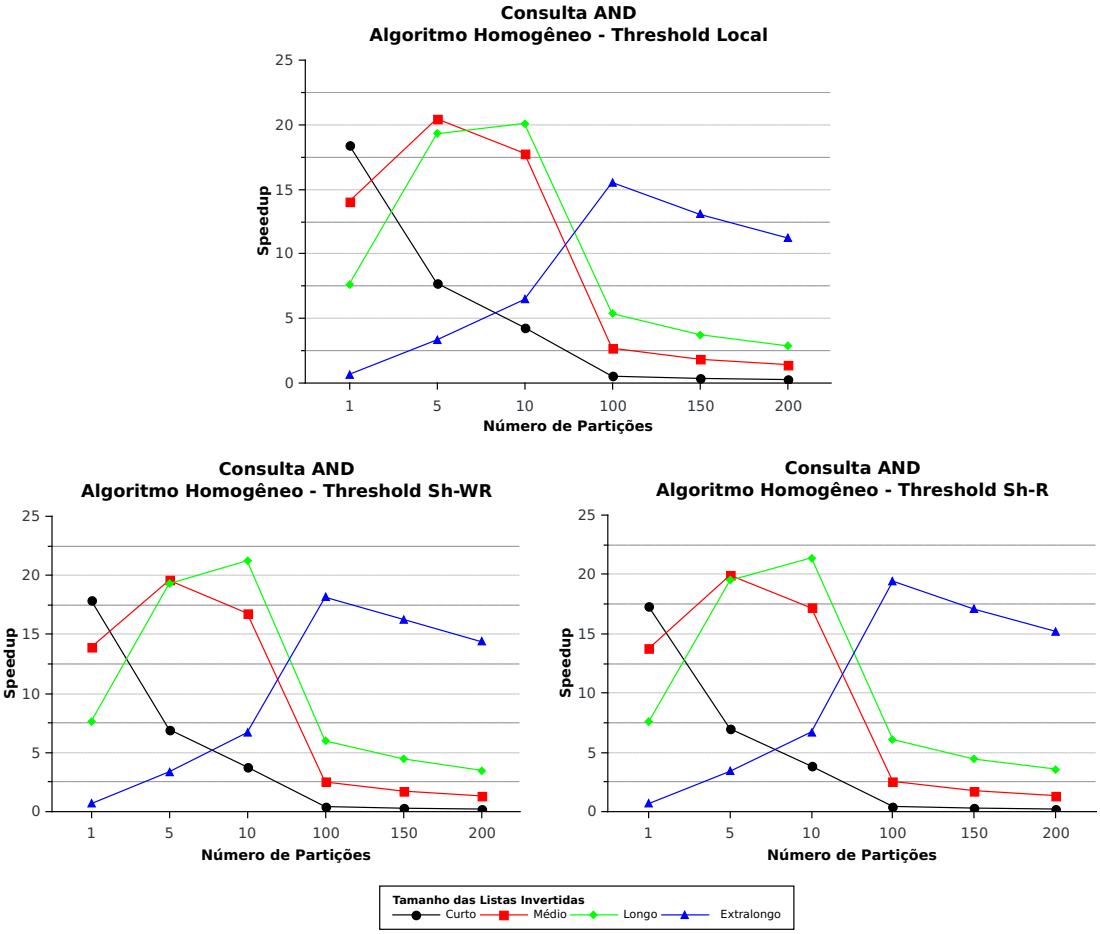

Figura 5.18: Desempenho do algoritmo paralelo homogêneo com a versão paralela do MaxScore e as políticas de *threshold* para consultas AND variando o número de partições.

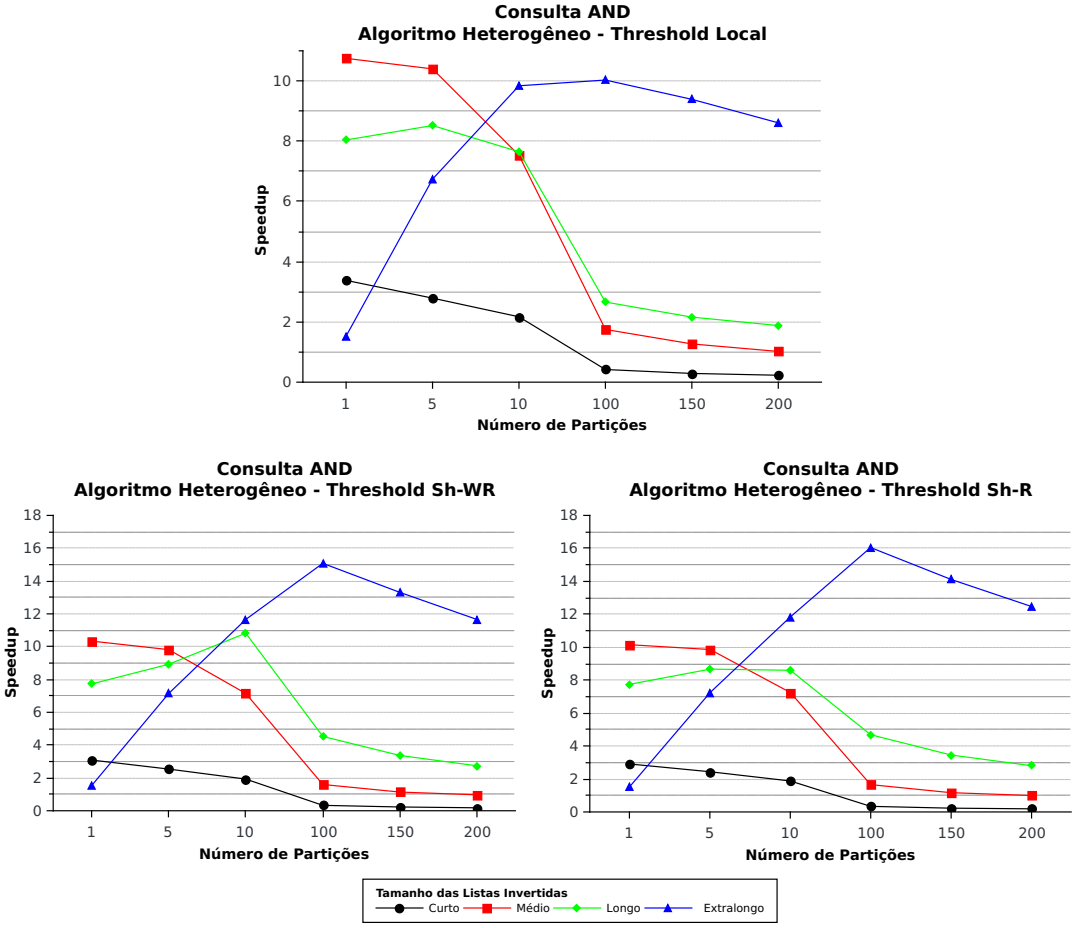

Figura 5.19: Desempenho do algoritmo paralelo heterogêneo com a versão paralela do MaxScore e as políticas de *threshold* para consultas AND variando o número de partições.

As estratégias de propagação mostraram efetividade nos algoritmos heterogêneos para consultas extralongas. As estratégias *Sh-R* e *Sh-WR* chegaram a alcançar um *speedup* de $\sim 60\%$ a mais em relação à implementação local. O melhor desempenho foi apresentado pela estratégia *Sh-R* para consultas extralongas com 100 partições por bloco de *threads*. A Figura 5.20 demonstra os ganhos computacionais do algoritmo heterogêneo com as diferentes implementações de propagação sobre o aumento do número de partições por bloco de *threads* para consultas extralongas.

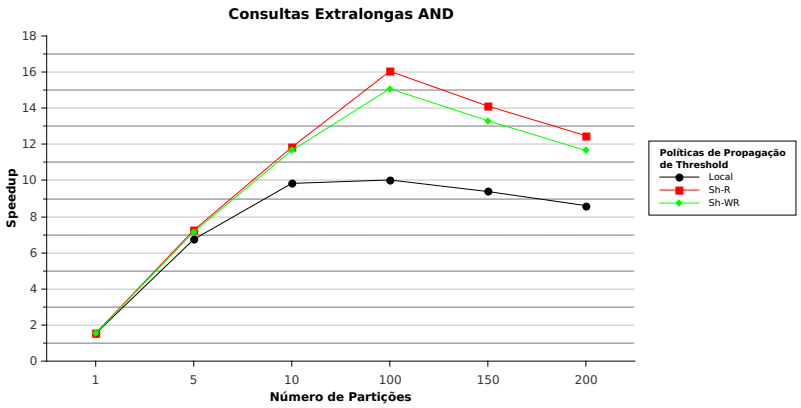

Figura 5.20: Desempenho das políticas de propagação de *threshold* no algoritmo paralelo heterogêneo com a versão paralela do MaxScore para consultas extralongas AND.

5.4 Considerações Finais do Capítulo

As avaliações das propostas aqui apresentadas mostraram que as melhores configurações de parâmetros (tamanho das partições e número de blocos de thread) variam bastante com as características das consultas.

A proposta de paralelização para o processamento de consulta utilizando a versão do WAND e MaxScore apresentou eficiência nas consultas curtas, médias e longas, tanto para consultas OR quanto para AND, processando de 1 a 10 partições por bloco de *threads*. Nas consultas extralongas, foi necessário aumentar o número de partições por bloco de *threads* para se alcançar um ganho computacional entre $\sim 10\times$ e $\sim 12\times$. Com valor acima de 100 partições por bloco, as implementações apresentaram quedas de desempenho e, mesmo assim, houve ganho de *speedup* em todos os tamanhos das consultas, com exceção das curtas.

Como esperado, a estratégia homogênea de particionamento alcançou melhores resultados do que a estratégia heterogênea, apesar de apresentar uma estratégia aproximada. A estratégia homogênea com alguns ajustes alcançou um *recall* com valor a partir de 70%. Assim, essa estratégia pode ser considerada uma opção quando o desempenho se faz necessário, sendo que quantidades pequenas de perdas de qualidade nos resultados sejam admissíveis.

As implementações das políticas de compartilhamento do *threshold* mostraram eficiência nas consultas com o tamanho acima das curtas e com reaproveitamento de dados frente à implementação local sem influenciar na acurácia dos resultados. Essa eficiência ficou mais evidente nas consultas extralongas, pois a propagação de *threshold* alcança mais processadores no início do processamento das partições, fazendo com que haja saltos já no início de cada partição. Isso acontece devido à alta sobrecarga dos processadores. Dessa forma, a proposta

mostrou escalável frente ao aumento de tamanho das listas invertidas dos termos das consultas.

Capítulo 6

PROPOSTA DE PARALELIZAÇÃO DO PROCESSAMENTO DE LOTE DE CONSULTAS EM GPUS

Um lote de consultas é um conjunto de consultas que podem ser coletadas de um fluxo de consultas recebidas de diversos usuários por uma máquina de busca durante um período de tempo, armazenadas em buffer, processadas e devolvidas simultaneamente para os usuários.

Os algoritmos de processamento de consultas podem ser executados em poucos milissegundos para processar uma consulta. A qualidade da experiência dos usuários de consultas na Web não é degradada se a latência for mantida abaixo dos limites razoáveis de até um ou dois segundos. Assim, várias consultas podem ser agrupadas em lotes para execução, a fim de aumentar a vazão de consultas, mesmo que isso aconteça ao custo de pequeno aumento na latência (GAIOSO et al., 2018) (HUANG et al., 2017).

Neste capítulo, são apresentadas e analisadas duas propostas de paralelização para o processamento de lotes de consulta. Para mostrar a eficiência dessas paralelização na prática, as propostas são implementadas e avaliadas com consultas de registros reais.

6.1 Estratégia Assíncrona de Processamento de Lote de Consultas

A primeira proposta de paralelização do processo de identificação de documentos para cada consulta de um lote de consultas consiste em distribuir o processamento de cada consulta para os processadores de granularidade grossa disponíveis na GPU. Essa proposta é detalhada no Algoritmo 6. O algoritmo itera sequencialmente sobre as consultas do lote para cada consulta e

os termos são enviados aos processadores de granularidade grossa, linha 2. Em paralelo, esses processadores buscam as listas invertidas dos termos recebidos e identificam os k documentos mais relevantes de suas partições. A barreira de sincronização localizada na linha 3 garante que a identificação de documentos seja realizada antes do início do processo de combinação de resultados.

Essa primeira solução para o lote de consultas, Algoritmo 6, é inspirado no modelo BSP (VALIANT, 1989), sendo constituída por duas superetapas para cada consulta do lote. Uma é o processamento das listas invertidas e a segunda superetapa é a combinação dos resultados alcançados.

Após a identificação de documentos de uma consulta na linha 2 pelos processadores de granularidade grossa disponíveis na GPU, os processos de combinações são realizadas em paralelo na GPU por meio de $\theta(\log(|P|))$ chamadas da função *kernel ParallelMerge*. Entre essas chamadas, a barreira localizada na linha 6 permite que uma chamada ocorra somente no término da antecessora. No final de todo processo de combinação, os k documentos mais relevantes à consulta são adicionados na estrutura de dados global *result* que conterá todos os resultados das consultas do lote. A partir desse ponto, outra consulta do lote é inicializada no processamento.

Algorithm 6 Estratégia Assíncrona de Processamento de Lote de Consultas.

Input: [1] as consultas do batch $< q_1, ..., q_m >$;[2] o número de documentos a serem identificados k; [3] um conjunto de P processadores; [4] o número i de cada processador $p_i \in P$, onde $1 \leq i \leq P$;
Output: Lista dos top-k documentos $result[1..k]$;

1: **for** query $< t_1, ..., t_n > \in q_l$ and $q_l \in < q_1, ..., q_m >$, where $l \leq m$ **do**
2: Each processor $p_i \in P$ **in parallel**
$parcial\_result[l][i] \leftarrow$ ParallelMatchProcessing($< t_1, ..., t_n >, k$) {Cada p_i processa $\theta(\frac{N_m}{P})$ docs e returna os top-k docs.}
3: sync_barrier()
4: **for** $j \leftarrow |P|; j > 0; j \leftarrow \frac{j}{2}$ **do**
5: Each processor p_i **in parallel**, where $i < j$:
ParallelMerge($parcial\_result[l][1..(j * k)]$) {Cada p_i realiza uma junção de duas listas ordenadas dos top-k docs.}
6: sync_barrier()
7: **end for**
8: $result[l] \leftarrow parcial\_result[l][1..k]$
9: **end for**

6.2 Estratégia Síncrona de Processamento de Lote de Consultas

A segunda proposta de paralelização do processamento de lote de consultas processa todas as consultas simultaneamente na GPU. Cada consulta é mapeada para um processador de granularidade grossa, em que o processamento ocorre de forma paralela nos processadores de granularidade fina. O Algoritmo 7 detalha esse processo. Cada processador de granularidade grossa executa uma consulta distinta do lote, pecorrendo as listas invertidas dos termos da consulta totalmente, sem um particionamento global. Isso é realizado pela chamada da função

ParallelMatchProcessing na linha 1, em que a identificação dos documentos mais relevantes à consulta é concluída. Neste formato, o Algoritmo 7 exclui a necessidade dos resultados parciais, como foi necessário pelo Algoritmo 6. Dessa forma, o processo de combinação dos resultados parciais não é mais necessário.

Uma barreira localizada na linha 2 garante que os resultados de todas as consultas do lote serão retornados somente quando todos os processadores finalizarem o processamento das consultas. Assim, o tempo de execução do algoritmo será dominado pelo tempo de processamento da consulta que contiver o maior número de documentos a serem avaliados.

Algorithm 7 Estratégia Síncrona de Processamento de Lote de Consultas.

Input: [1] as consultas do batch $< q_1, ..., q_m >$; [2] o número de documentos a serem identificados k; [3] um conjunto de P processadores; [4] o número i de cada processador $p_i \in P$, onde $1 \leq i \leq P$;
Output: Lista dos top-k documentos $result[1..k]$;
1: Each processor $p_i \in P$ **in parallel**
$result[i] \leftarrow$ ParallelMatchProcessing($< t_1, ..., t_n > \in q_i, k$) {Cada p_i processa as listas de *postings* da consulta q_i e retorna os top-k docs.}
2: sync_barrier()

6.3 Detalhes de Implementação dos Algoritmos

Para concretizar as implementações das propostas de paralelização, as técnicas apresentadas na Seção 4.5 do Capítulo 4 foram também utilizadas nestas soluções. Juntamente a elas, outras estratégias de programação oferecidas pela arquitetura CUDA foram necessárias para tornar as implementações eficientes.

6.3.1 O Uso de CUDA *Streams*

A estratégia assíncrona de processamento de lote executa uma consulta individual de maneira similar à proposta detalhada pelo Algoritmo 3. Uma única consulta é mapeada para os blocos de *threads* que são distribuídos para os SMs. A identificação dos documentos mais relevantes é executada em paralelo por meio de uma chamada à função *ParallelMatchProcessing*. A etapa de combinação dos resultados da consulta também é operada em paralelo por meio de chamadas à função *ParallelMerge*. Para seguir com o processamento das consultas do lote de maneira paralela, sem obrigatoriamente esperar a finalização da execução da consulta anterior, foi necessário explorar a técnica de CUDA *streams*.

A técnica de CUDA *Streams* é uma forma de trabalhar com concorrência de tarefas na GPU. Um CUDA *stream* é a uma sequência de operações CUDA, operações de processamento e/ou de memória, que são executadas na GPU em ordem de emissão, ou seja, a execução das tarefas

ocorre no estilo de fila. Essa restrição de operação é garantida pela API CUDA em tempo de execução. Portanto, a consumação das tarefas atribuídas a um CUDA *stream* é assegurada, de tal forma que uma tarefa é consumida somente quando todas posteriores a ela forem finalizadas.

Enquanto a ordem das operações de um CUDA *stream* é mantida, operações de diferentes CUDA *streams* não são restritas quanto ao processamento, ou seja, em um mesmo contexto de execução da GPU pode haver o processamento simultâneo de vários CUDA *streams*. Isso leva a considerar que a realização dos CUDA *streams* é assíncrona.

Esses conceitos foram utilizados na implementação da proposta assíncrona. Os processos de identificar os documentos e de combinação de resultados de cada consulta foram encapsulados em CUDA *streams* distintos. Ao proceder dessa maneira, a quantidade de CUDA *streams* criadas na solução fica a mesma quantidade de consultas contidas no lote. As barreiras de sincronização das linhas 3 e 6 foram substituídas pela ordem de execução das operações dentro dos CUDA *streams*. Assim, vários envios simultâneos de consultas para a GPU foram alcançados, de forma que não houvesse espera nestes envios. A Figura 6.1 ilustra o mapeamento do processamento das consultas para os CUDA *streams* e a distribuição de trabalho que é realizada pela API CUDA.

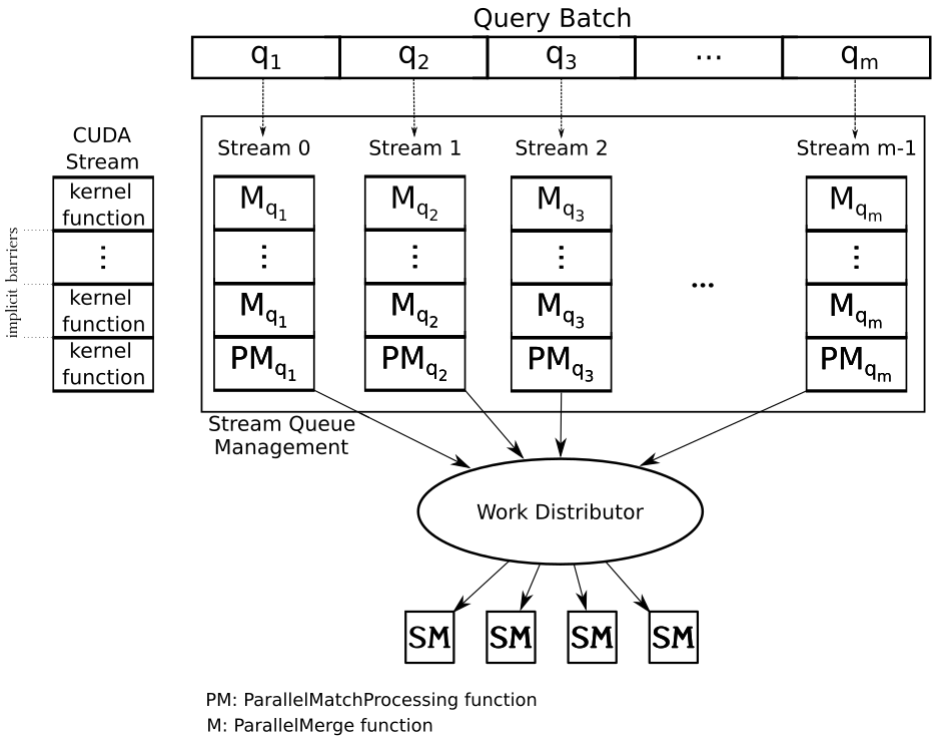

Figura 6.1: O processamento assíncrona do lote de consulta distribuído em CUDA *Streams*.

O hardware da GPU impõe um limite máximo de tarefas sendo executadas de forma concorrentes. Esse limite depende do modelo da arquitetura utilizada. A arquitetura Pascal, utilizada na experimentação deste trabalho, oferece um limite máximo de 32 tarefas (*streams*) concorren-

tes nos processadores. Analisando isto, a implementação da primeira proposta de paralelização representada pelo Algoritmo 6 tem o limite máximo de 32 consultas sendo executadas simultaneamente.

6.3.2 Processamento de Consultas por Bloco de *Threads*

A segunda solução de paralelização emite o processamento de cada consulta para os processadores de granularidade grossa. Como detalhado na Seção 4.5.1 do Capítulo 3, a arquitetura CUDA transparece o número de Multiprocessadores contidos em uma GPU oferecendo um número alto de *threads* na execução das aplicações. Essas *threads* são organizadas em blocos de *threads* e estes são lançados em grades de *threads*. Os blocos são considerados uma virtualização dos SMs. Assim, para efetivar a implementação da solução síncrona, o processamento de consultas do lote é emitida para os blocos, sendo que cada qual com uma consulta distinta. Essa condução direciona o processo de todo lote a ser aplicado por uma única grade de *threads*.

As *threads* de uma grade executam o mesmo *kernel*. A finalização da execução de um *kernel* é sucedida quando o processamento de todas as *threads* termina. Desse modo, a implementação da segunda solução torna-se síncrona em relação ao processamento de cada consulta do lote. Logo, o desempenho dessa estratégia irá depender do desempenho da execução da consulta que leve mais tempo de processamento.

6.4 Considerações Finais do Capítulo

Este capítulo apresentou duas estratégias de paralelização de processamento de lote de consultas, sendo que a primeira processa cada consulta de forma assíncrona e a segunda, síncrona.

A proposta da estratégia assíncrona conduz o processamento de cada consulta a ser realizado por vários processadores. No entanto, isso faz com que as ocupações das consultas na GPU influenciem no desempenho dessa solução, pois a quantidade média de consultas processadas em paralelo depende do tamanho médio das listas invertidas. Assim, quanto maiores forem os tamanhos das listas invertidas dos termos da consulta, maior será a ocupação média dos processadores por consulta e, consequentemente, menor será a quantidade média de consultas executadas em paralelo.

Na estratégia síncrona, o lote de consultas é proposto a ser processado totalmente em paralelo, de forma que cada consulta seja executada em um único processador e os resultados são

retornados em conjunto. Assim, o desempenho dessa solução será dependente da consulta que leva mais tempo para ser processada.

As implementações de ambas estratégias fizeram o uso das propostas de paralelização de consultas descritas no Capítulo 4 para realizar o processamento individual de cada consulta. Além disso, se fez necessário a utilização das técnicas de programação de paralelismo de tarefas da arquitetura CUDA GPU.

Capítulo 7

AVALIAÇÃO DAS PROPOSTAS DE PARALELIZAÇÃO PARA O PROCESSAMENTO DE LOTE DE CONSULTAS

As estratégias de paralelização de processamento de lote de consultas foram propostas no Capítulo 6. Neste capítulo, avaliações de desempenho dessas estratégias, Algoritmo 6 e Algoritmo 7, são realizadas com as estratégias de particionamento executando consultas disjuntivas (OR) para achar os top-k (k=128) documentos mais relevantes. As consultas são obtidas a partir do registro de consultas (*log de consultas*).

O registro de consultas, obtido de requisições reais em ambiente Web pelo *Yahoo! logs* (MENDOZA et al., 2016), é aplicado somente nas soluções de processamento de lote de consultas. Desse registro, foram empregadas 2000 consultas, divididas em lotes para serem processadas juntas. Cada lote contém 500 consultas as quais foram executadas 10 vezes e uma média foi obtida dessas execuções.

As versões paralelas dos algoritmos WAND e MaxScore foram implementadas e testadas dentro das estratégias de processamento de lote. Para cada algoritmo na estratégia assíncrona, as três abordagens do *threshold* foram realizadas e avaliadas: local, *Safe-Read Shared (Sh-R)* e *Safe-Write-Read Shared (Sh-WR)*. Todos os algoritmos usaram a implementação global do *upper bound* e *maxscore*.

Os algoritmos originais, WAND e MaxScore, foram testados sobre o conjunto de consultas retiradas do registro para resolver o problema de processamento de lote de consulta com o intuito de analisar o desempenho das propostas paralelas. A Tabela 7.1 mostra os tempos médios de execução obtidos por esses algoritmos.

Algoritmo Processamento	Tempo (s)
WAND	17,106
MaxScore	22,763

Tabela 7.1: Tempo médio de execução (s) do lote de consultas pelos algoritmos sequenciais.

7.1 Avaliação da Estratégia Assíncrona de Lote de Consultas

Os experimentos da estratégia assíncrona foram executados em duas partes. A primeira parte foi realizada com 1 e 10 partições por bloco de *threads*. A investigação do comportamento do algoritmo frente à redução da ocupação da GPU por consulta e ao excesso de trabalho por processador foi desempenhado na segunda parte dos experimentos. Para isso, o número de partições por bloco de *threads* foi variado entre 1 e 250 partições.

7.1.1 Processamento em Lotes com Algoritmo WAND

Os tempos de execução adquiridos na primeira parte dos experimentos com particionamento homogêneo e heterogêneo com a versão paralela do WAND são detalhados na Tabela 7.2. Os maiores *speedups* de cada política de propagação de *threshold* foram destacados. Os tempos são mostrados em segundos. A extensão das partições sobre as listas invertidas foi variada em 32, 64 e 128 docIDs e a quantidade de partições por bloco foi de 1 e 10 partições.

O aumento da extensão das partições quanto o aumento do números de partições por bloco fazem com que a quantidade de recursos necessários para processar uma única consulta seja reduzida e, ao fazer isso, possibilitam aumentar o número de processamento simultâneo de consultas distintas. Desse modo, a estratégia assíncrona alcançou *speedups* maiores, e os melhores resultados alcançados foram nos extremos dos dois casos, 10 partições por bloco e 128 docIDs de extensão. Essa decorrência sugere que o algoritmo beneficia-se com a redução da ocupação gerada por cada consulta.

As diferentes políticas de propagação de *threshold* impactaram nos tempos de execução somente quando houve o aumento de partições por bloco de threads, onde se constatou uma melhora de $\sim 42\%$ no *speedup* comparado com a política local.

Algor.	**Thres-**	**# de**	**32 docIDs**		**64 docIDs**		**128 docIDs**	
	hold	**Part.**	**Tempo**	**Speedup**	**Tempo**	**Speedup**	**Tempo**	**Speedup**
Hom.	Local	1	52,17	0,33	26,10	0,66	13,51	1,27
Hom.	Local	10	5,58	3,07	3,05	5,60	1,77	**9,65**
Hom.	Sh-R	1	51,51	0,33	25,97	0,66	13,16	1,30
Hom.	Sh-R	10	5,07	3,37	2,63	6,51	1,33	**12,86**
Hom.	Sh-WR	1	51,51	0,33	25,82	0,66	12,74	1,34
Hom.	Sh-WR	10	5,06	3,38	2,68	6,39	1,41	**12,10**
Het.	Local	1	8,59	1,99	4,90	3,49	2,83	6,05
Het.	Local	10	1,81	9,43	1,42	12,01	1,16	**14,70**
Het.	Sh-R	1	7,54	2,27	4,07	4,20	2,49	6,86
Het.	Sh-R	10	1,37	12,49	1,02	16,74	0,82	**20,76**
Het.	Sh-WR	1	7,51	2,28	4,06	4,21	2,34	7,32
Het.	Sh-WR	10	1,37	12,45	1,02	16,73	0,85	**20,24**

Tabela 7.2: Resultados de desempenho do algoritmo assíncrono de processamento de lote de consultas com WAND paralelo. As colunas de 'Tempo' trazem o tempo de execução em segundos para cada teste.

A segunda parte do experimento elevou a quantidade de partições por blocos de *threads* entre 1 a 250 partições. Os gráficos dos tempos de execução em segundos do algoritmo homogêneo baseado no WAND para o lote de consultas são apresentados em escala logarítmica na Figura 7.1. O tamanho das partições (extensão das partições sobre as listas invertidas) foi variado entre 32, 64 e 128 docIDs. Nessa segunda parte do experimento, o algoritmo homogêneo obteve os menores tempos em todos os casos com até 100 partições por bloco e com as partições de tamanho de 128 docIDs. A partir dessa quantidade de partições por bloco, os tempos de execução tenderam a estabilizar entre os três tamanhos de partições experimentados. Isso pode ser justificado com a perda de desempenho devido ao excesso de trabalho por blocos de *threads* e, ao mesmo tempo, o aumento de desempenho com o crescimento do número de saltos de documentos devido à propagação de *threshold*.

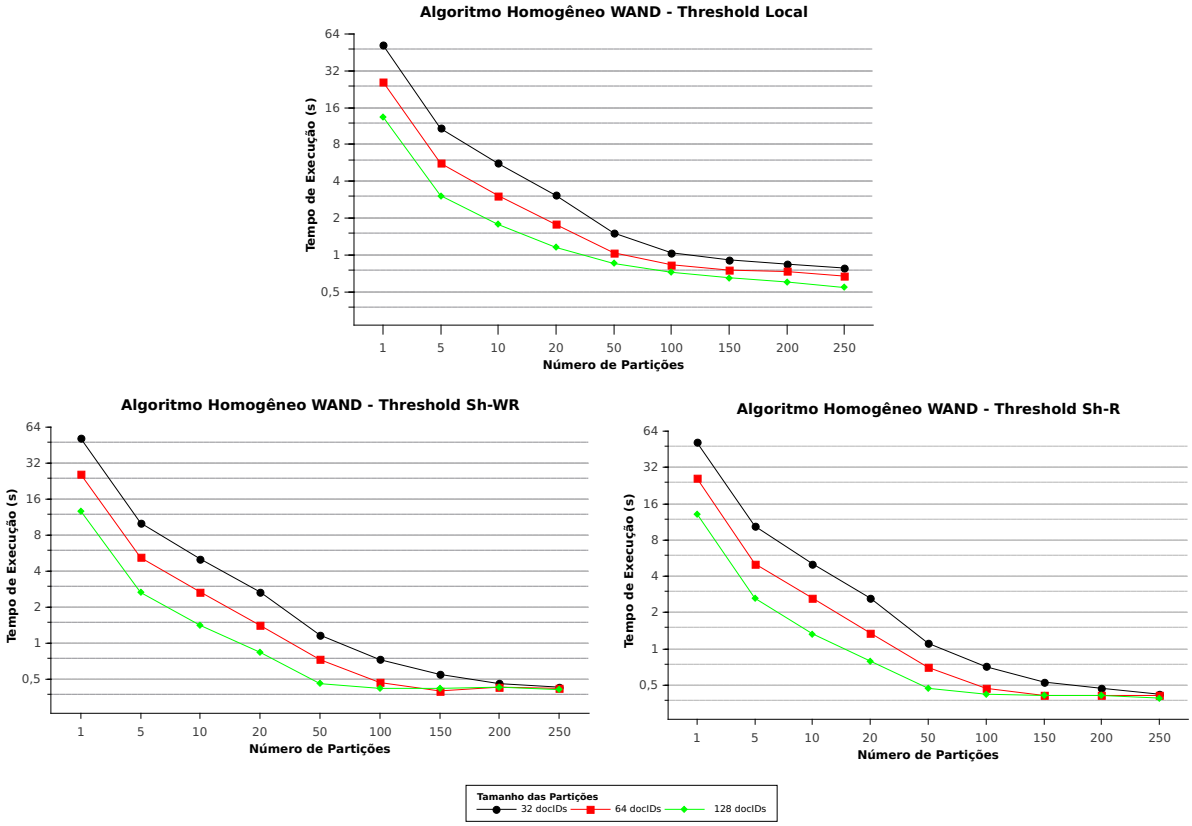

Figura 7.1: Tempos de execução (s) em escala logarítmica do algoritmo paralelo homogêneo com a versão paralela do WAND e as políticas de *threshold* para o lote de consultas.

As políticas de propagação do *threshold* no algoritmo homogêneo com a versão paralela WAND se mostraram mais eficientes do que o caso local para todos os contextos onde há reaproveitamento de dados, ou seja, o número de partições por bloco maiores do que 1 partição. A Figura 7.2 ilustra o gráfico de ganho computacional das estratégias de propagação de *threshold* com o tamanho de 128 docIDs das partições. O melhor *speedup* do algoritmo homogêneo com WAND foi obtido com 250 partições por bloco de *threads* em todas as políticas de compartilhamento. Na quantidade de 100 partições por bloco, as estratégias apresentaram um ganho de $\sim 78\%$ de *speedup* em relação à política local.

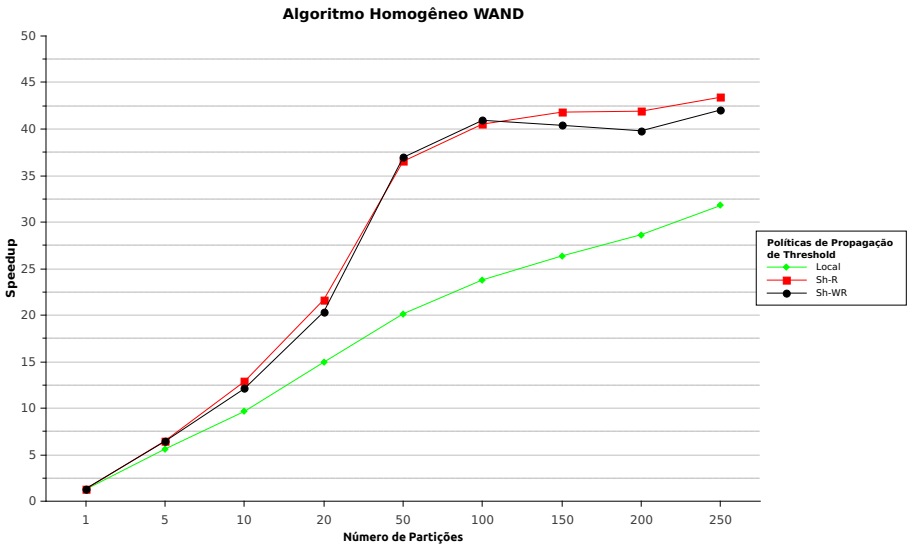

Figura 7.2: Desempenho do algoritmo paralelo homogêneo com a versão paralela do WAND e as políticas de *threshold* e partições com tamanho de 128 docIDs para o lote de consultas.

A Figura 7.3 ilustra os gráficos da segunda parte do experimentos para a estratégia heterogênea com a versão paralela do WAND. Essa implementação atingiu os melhores resultados entre os números de 100 a 250 partições por bloco com o tamanho das partições de 128 docIDs em cada lista de *postings*. A partir de 100 partições por bloco, os melhores tempos foram obtidos com as menores quantidades de documentos por partição, 32 e 64.

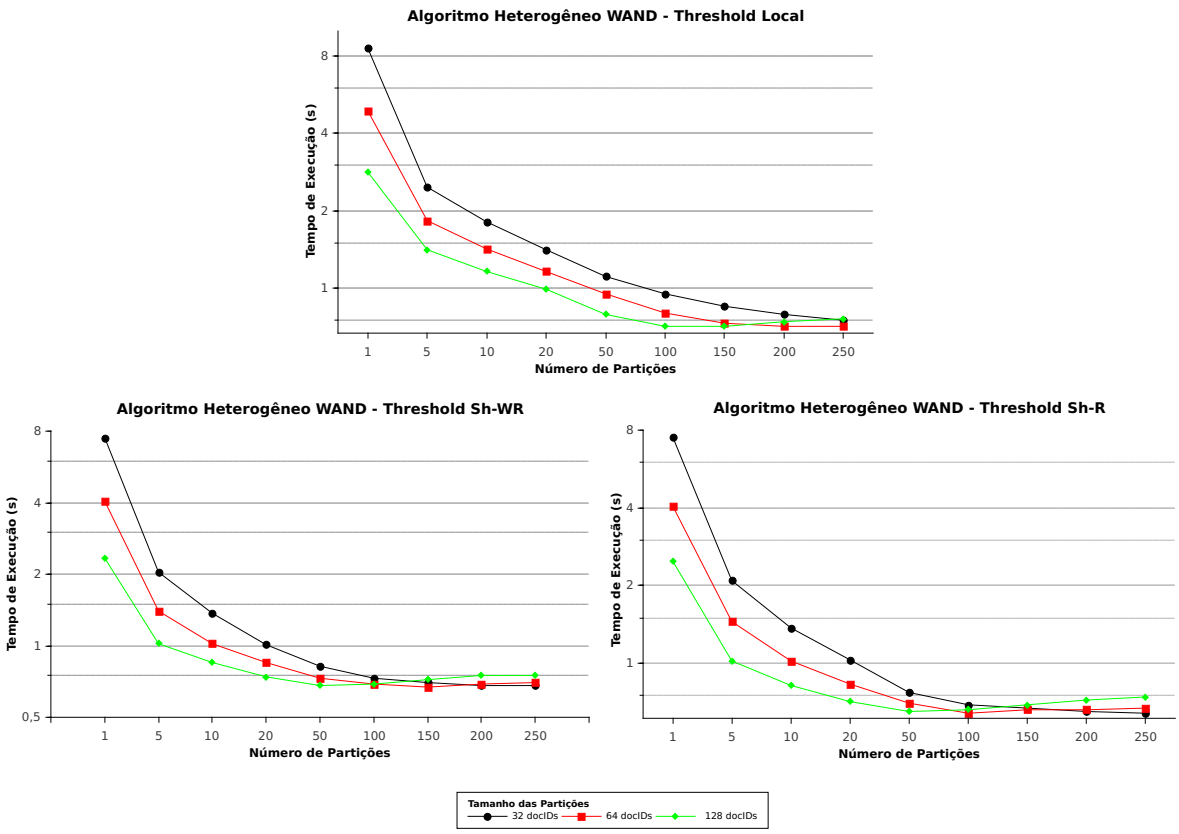

Figura 7.3: Tempos de execução (s) em escala logarítmica do algoritmo paralelo heterogêneo com a versão paralela do WAND e as políticas de *threshold* para o lote de consultas.

O compartilhamento do *threshold* no algoritmo heterogêneo levou a ganhos de desempenho em todos os casos com reaproveitamento de dados por consulta (números de partições por bloco maiores que 1), atingindo um *speedup* de $\sim 25\times$, enquanto a política local obteve $\sim 22\times$ de *speedup*. O melhor *speedup* de cada política de propagação foi alcançado no caso de 250 partições por bloco de *threads*. A Figura 7.4 ilustra o gráfico do ganho computacional do algoritmo heterogêneo do WAND com a extensão mínima das partições de ~ 32 docIDs em cada lista invertida dos termos da consulta.

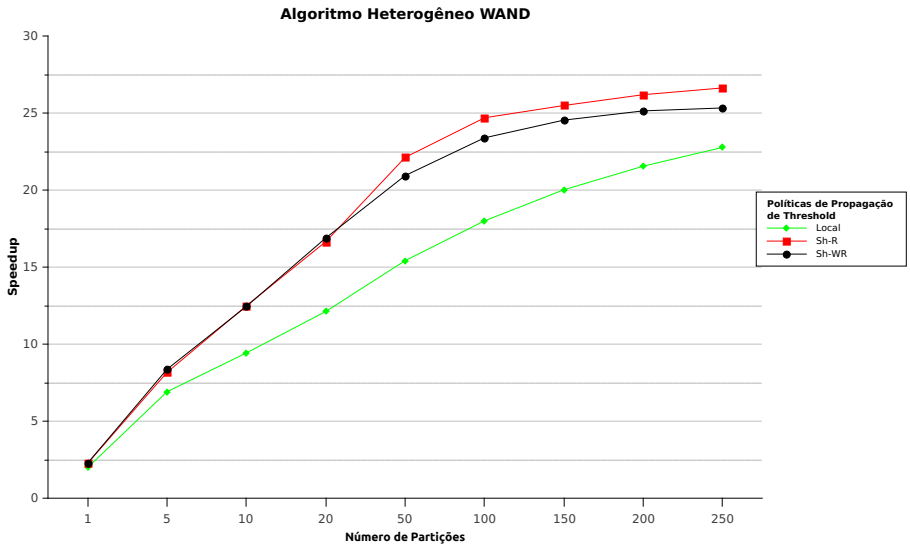

Figura 7.4: Desempenho do algoritmo paralelo heterogêneo com a versão paralela do WAND, as políticas de *threshold* e partições com tamanho de 32 docIDs para o lote de consultas.

7.1.2 Processamento em Lote com Algoritmo MaxScore

Os tempos de execução do estratégia assíncrona de processamento de lote de consultas com a versão paralela do MaxScore são detalhados na Tabela 7.3. Nesse primeiro experimento, o número de partições por bloco de *threads* foi variado entre 1 e 10 partições e a extensão das partições foi alterado entre os valores 32, 64 e 128 documentos.

As políticas de propagação do *threshold* na estratégia assíncrona, Algoritmo 6, com a versão paralela do MaxScore não mostraram diferenças significativas nos *speedups*. No algoritmo heterogêneo, a estratégia local apresentou melhor desempenho frente às outras.

Algor.	**Thres-**	**# de**	**32 docIDs**		**64 docIDs**		**128 docIDs**	
	hold	**Part.**	**Tempo**	**Speedup**	**Tempo**	**Speedup**	**Tempo**	**Speedup**
Hom.	Local	1	51,28	0,44	25,77	0,88	13,14	1,73
Hom.	Local	10	5,41	4,21	2,92	7,80	1,64	**13,88**
Hom.	Sh-R	1	51,42	0,44	25,75	0,88	13,08	1,74
Hom.	Sh-R	10	5,20	4,38	2,78	8,19	1,51	**15,09**
Hom.	Sh-WR	1	51,38	0,44	25,80	0,88	12,87	1,77
Hom.	Sh-WR	10	5,21	4,37	2,79	8,15	1,52	**14,98**
Het.	Local	1	9,71	2,34	6,23	3,66	4,65	4,89
Het.	Local	10	3,74	6,09	3,39	6,72	3,18	**7,16**
Het.	Sh-R	1	9,53	2,39	6,25	3,64	4,92	4,62
Het.	Sh-R	10	3,94	5,77	3,67	6,21	3,41	**6,67**
Het.	Sh-WR	1	10,00	2,28	6,58	3,46	4,83	4,71
Het.	Sh-WR	10	4,04	5,64	3,65	6,23	3,46	**6,57**

Tabela 7.3: Resultados de desempenho do primeiro algoritmo de processamento de lote de consultas com a versão paralela do MaxScore. As colunas de 'Tempo' trazem o tempo de execução em segundos para cada teste.

A Figura 7.5 ilustra os gráficos em escala logarítmica dos resultados da segunda parte dos experimentos com a estratégia homogênea da versão paralela do MaxScore. Essa estratégia atingiu os melhores tempos de execução, inferiores a um segundo, a partir de 100 partições por bloco e nas extensões de 32 e 64 docIDs. Os experimentos com a extensão das partições igual a 128 docIDs apresentaram os menores tempos até com 50 partições por bloco de *threads*.

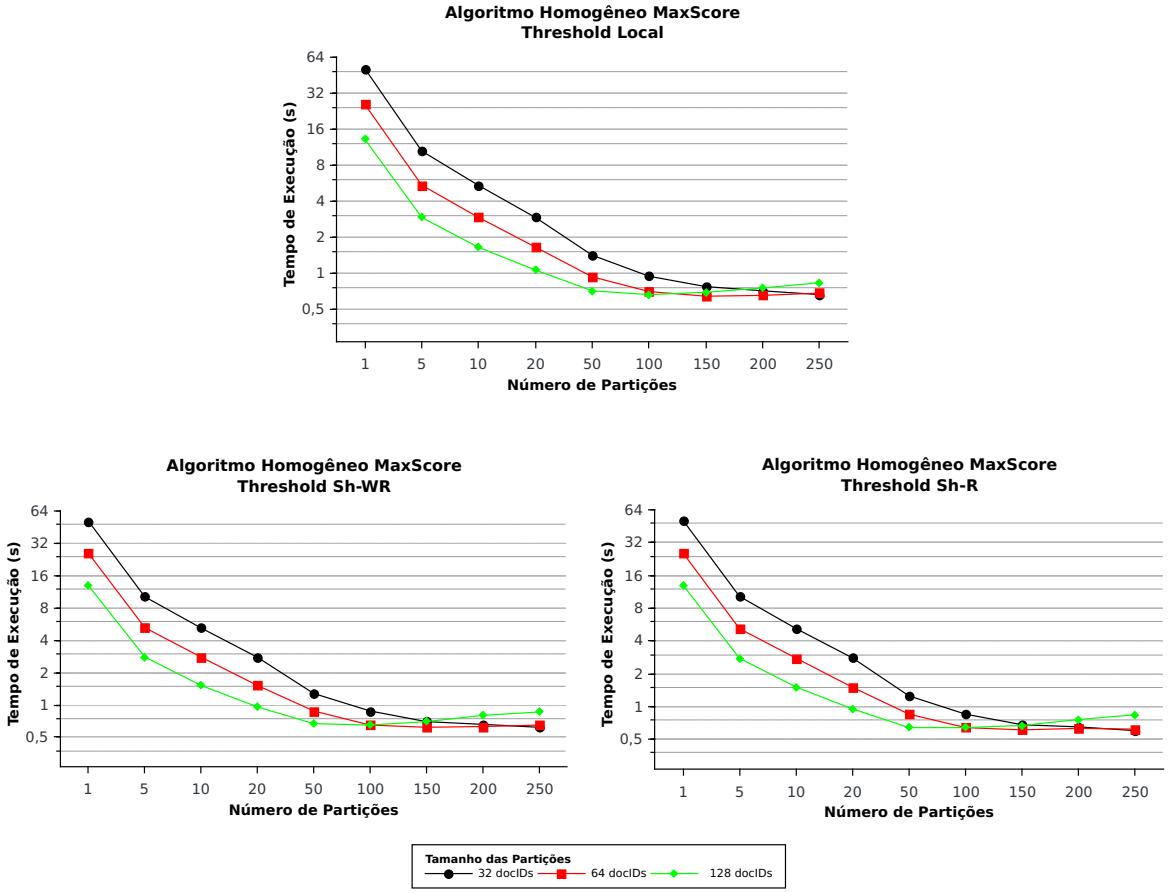

Figura 7.5: Tempos de execução (s) em escala logarítmica do algoritmo paralelo homogêneo com a versão paralela do MaxScore e as políticas de *threshold* para o lote de consultas.

Os ganhos computacionais da estratégia homogêneo com Maxscore com a extensão das partições de 32 docIDs são ilustrados na Figura 7.6. As políticas de propagação não apresentaram diferenças significativas entre elas até 50 partições por bloco, porém a partir de 100, as políticas $Sh-R$ e $Sh-WR$ alcançaram superioridade sobre a estratégia local.

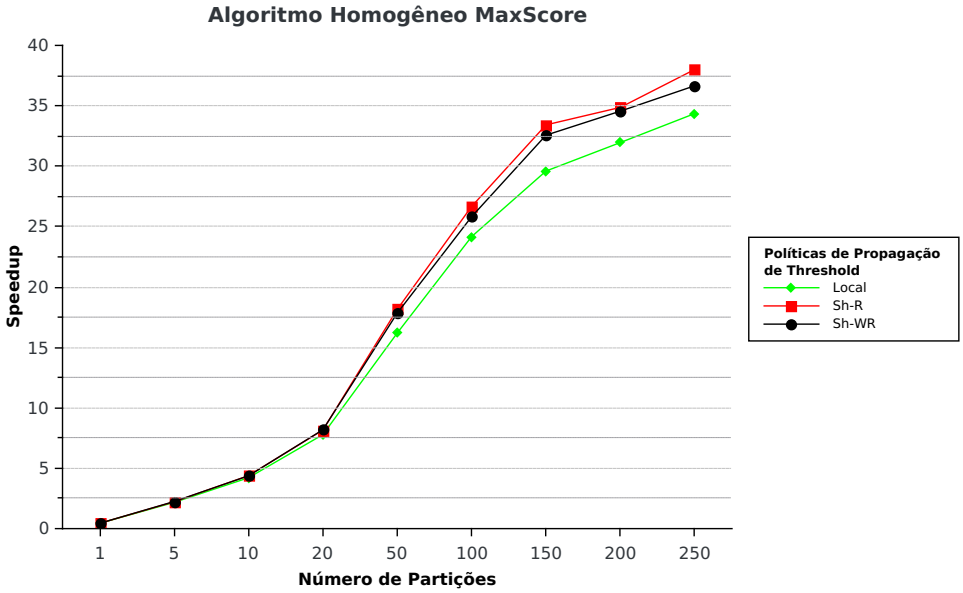

Figura 7.6: Desempenho do algoritmo paralelo homogêneo com a versão paralela do MaxScore, as políticas de *threshold* e partições com tamanho de 32 docIDs para o lote de consultas.

Seguindo na segunda parte dos experimentos, os tempos de processamento da estratégia heterogêneo com a versão paralela do Maxscore ficaram entre 3s e 4s para todas as políticas de propagação de *threshold* nas quantidades maiores de 5 partições por bloco de *threads*. A Figura 7.7 mostra os gráficos de tempo de execução em segundos do algoritmo heterogêneo com Maxscore para as políticas de propagação do *threshold*.

O comportamento da estratégia heterogênea se manteve, embora tenha havido um aumento da quantidade de trabalho e, consequentemente, alterações na ocupação da GPU por consulta. Isso pode ser observado pelo gráfico ilustrado na Figura 7.8, em que mostra o *speedup* alcançado pela estratégia com Maxscore com a extenção das partições de 32 docIDs. Com isso, mostrou-se que as políticas de compartilhamento não levaram a ganhos significativos de desempenho no processamento do lote de consultas.

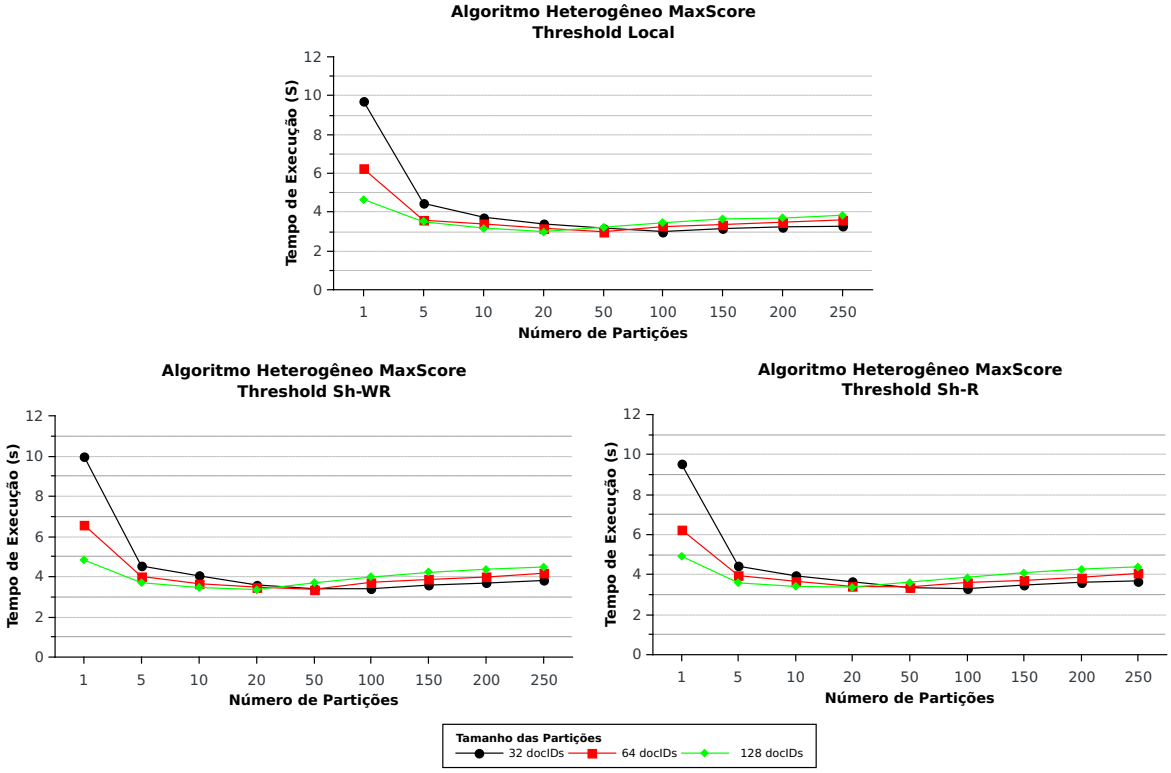

Figura 7.7: Tempos de execução (s) em escala logarítmica (eixo y) do algoritmo paralelo heterogêneo com a versão paralela do MaxScore e as políticas de *threshold* para o lote de consultas.

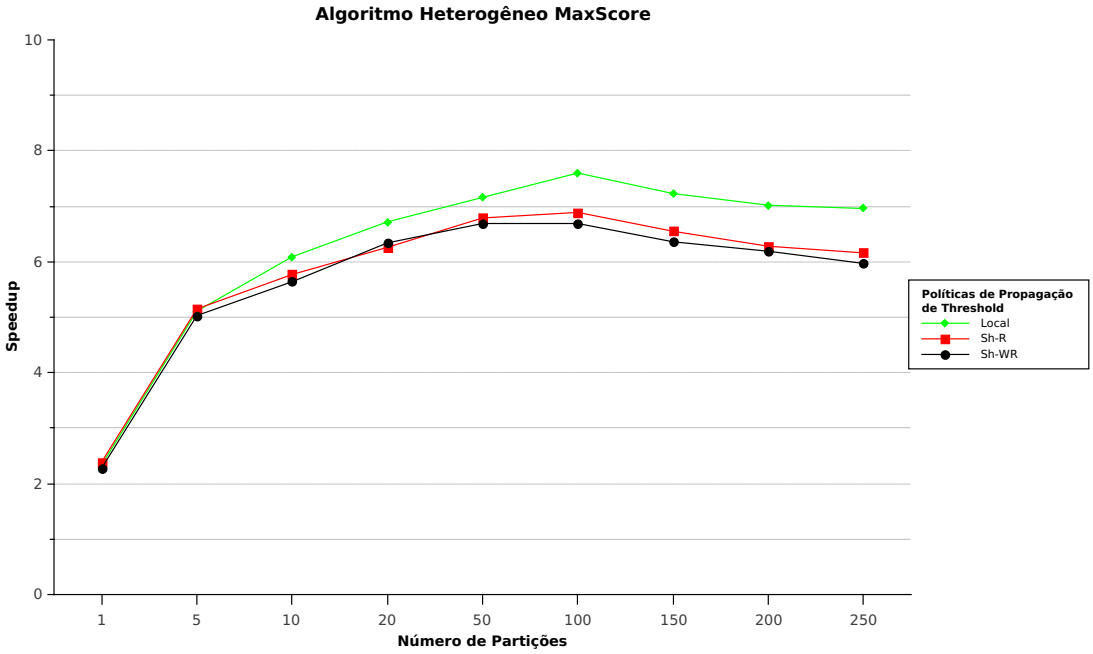

Figura 7.8: Desempenho do algoritmo paralelo heterogêneo com a versão paralela do MaxScore e as políticas de *threshold* e partições com tamanho de 32 docIDs para o lote de consultas.

7.2 Avaliação da Estratégia Síncrona de Lote de Consultas

A segunda proposta para o processamento de lote de consultas, detalhado no Algoritmo 7, propõe que cada processador manipule uma consulta do lote de forma independente dos outros processadores. Essa característica faz com que os experimentos sejam guiados somente pela quantidade de *threads* por processador, pois o número de partições por bloco de *threads* e as políticas de propagação do *threshold* não fazem efeito quando não há vários processadores executando uma única consulta.

A Tabela 7.4 mostra os tempos de execução e os *speedups* da proposta síncrona do lote de consultas com as versões paralelas do WAND e MaxScore. O número de *threads* por bloco de *threads* foi variado entre 32, 64 e 128 *threads*. O algoritmo com a versão paralela do WAND apresentou melhores resultados, alcançando um *speedup* de $\sim 21\times$, enquanto o algoritmo com a versão paralela do Maxscore apresentou $\sim 7,4\times$ de *speedup*. Dessa forma, a versão paralela do WAND apresentou ser mais promissora na segunda proposta de algoritmo de processamento de consulta.

Algoritmo	**32 *threads***		**64 *threads***		**128 *threads***	
	Tempo	**Speedup**	**Tempo**	**Speedup**	**Tempo**	**Speedup**
WAND Paralelo	0,809	21,131	0,799	21,403	0,813	21,021
MaxScore Paralelo	3,072	7,407	3,071	7,410	3,079	7,392

Tabela 7.4: Performance from Synchronous Query Batch Processing Strategy. As colunas de 'Tempo' trazem o tempo de execução em segundos para cada teste.

7.3 Considerações Finais do Capítulo

Este capítulo avaliou as propostas de paralelização do processamento de lote de consulta com as versões paralelas dos algoritmos WAND e MaxScore, as estratégias de particionamento e propagação do *threshold*. Ambas as propostas, tanto com o algoritmo MaxScore quanto o WAND, mostraram-se eficientes perante os algoritmos sequenciais.

A versão paralela do WAND e do MaxScore com a estratégia assíncrona e o particionamento homogêneo chegaram a alcançar um *speedup* $\sim 40\times$ no melhor caso que se apresentou com o número de partições por bloco maior que 100. Com isso, a estratégia assíncrona homogênea pode ser considerada uma boa opção quando o desempenho se faz necessário e perdas de qualidade sejam admissíveis.

A estratégia assíncrona atingiu um *speedup* $\sim 20\times$ com partições heterogêneas e com a

versão paralela do WAND, ao passo que a estratégia síncrona com a versão do WAND alcançou um *speedup* de $\sim 21\times$. Isso mostra que a versão paralela do algoritmo WAND nas duas estratégias se apresentou competitiva. Enquanto a versão paralela do algoritmo MaxScore com particionamento heterogêneo manteve o desempenho em $\sim 7\times$ em ambas estratégias, síncrona e assíncrona.

Na estratégia assíncrona de paralelização de lote de consultas, as políticas de propagação do *threshold*, *Sh-R* e *Sh-WR*, se mostraram mais eficientes do que a política local com a versão paralela do WAND. Enquanto com a versão paralela do Maxscore, já não alcançaram a mesma eficiência quando comparado com a versão do algoritmo sequencial Maxscore.

Os desempenhos alcançados por ambas estratégias de paralelização de lote de consultas, propostas no Capítulo 6, em conjunto com as propostas de processamento de consulta individual, detalhadas no Capítulo 4, mostraram que há vantagens de se processar lotes de consultas em paralelo na GPU.

Capítulo 8

TRABALHOS RELACIONADOS

Os algoritmos de classificação são executados sobre o índice invertido para retornar os k documentos mais relevantes para consultas de usuários. Esses algoritmos examinam as listas de *postings* associadas aos termos da consulta para determinar os documentos que são os mais relevantes à consulta. Atualmente, uma dificuldade importante na classificação de consultas é o grande tamanho das listas invertidas. Para evitar o processamento de todos os documentos nas listas de *postings* dos termos da consulta e reduzir a quantidade de cálculos, os sistemas de processamento de consulta usam algoritmos de poda dinâmica. Como já mencionado, este trabalho concentra-se nesses algoritmos, mais especificamente no algoritmo WAND (BRODER et al., 2003) e no algoritmo MaxScore (TURTLE; FLOOD, 1995).

Ding *et al.* (2009) propuseram a construção de um sistema de recuperação de informação de alto desempenho que utiliza GPUs, desenvolvendo algoritmos paralelos para diversas subtarefas, como compressão de índice invertido, intersecção de listas invertidas, processamento de consultas e outros. Os algoritmos de intersecção, chamados de *Parallel Merge Find*, se assemelham à estratégia de particionamento heterogênea proposta neste trabalho, no que se refere à obtenção de partições com as mesmas faixas de docIDs nas listas invertidas. No entanto, esses algoritmos foram baseados em algoritmos paralelos de *merge sorted* e as suas operações são executadas por *threads* que realizam uma busca independente por cada segmento. No sistema processamento de consulta de Ding *et al.* (2009), é utilizada a abordagem TAAT, ao invés dos algoritmos da categoria DAAT e dos algoritmos de poda dinâmica, como os algoritmos WAND e MaxScore. Os autores afirmam que embora a categoria DAAT trabalhe bem em arquiteturas sequenciais, a natureza sequencial dessa classe de algoritmos é inadequada para um processamento paralelo eficiente. Além disso, a estrutura de dados *heap* convencional não foi usada para o armazenamento dos top-k documentos devido à natureza sequencial apresentada por essa estrutura de dados e por estarem utilizando pontuações acumulativas em TAAT.

Wu *et al.* (2010) propuseram duas estratégias de intersecção de listas para a GPU, além de um algoritmo de escalonamento que determina se uma consulta irá ser executada na CPU ou na GPU. As consultas são agrupadas em lotes de acordo com o tamanho das listas para serem enviadas para GPU. Uma estratégia fixa vários blocos de *threads* por consulta na GPU para encontrar as intersecções entre as listas, enquanto uma segunda estratégia faz a intersecção com um bloco de *threads* por consulta na GPU. Ambos os processos geram uma lista intermediária com o mesmo tamanho das listas invertidas para indicar a presença dos docIDs que estão sendo buscados em cada bloco. Posteriormente, um algoritmo de *scan* é executado nesta lista intermediária para obter as posições dos docIDs que são buscados.

Os trabalhos Ding e Suel (2011) e Chakrabarti, Chaudhuri e Ganti (2011) de forma independente, ofereceram novos algoritmos sequenciais baseados nos algoritmos WAND e MaxScore, respectivamente, com um índice invertido aumentado, nomeado de *Block-Max Index*. Nestes trabalhos, utilizando o *Block-Max Index*, os algoritmos WAND e MaxScore foram denominados *Block-Max WAND (BMW)* e *Block-Max MaxScore (BMM)*, respectivamente. Essa estrutura *Block-Max Index* usa compactação nas listas invertidas, de maneira que os docIDs são compactados em blocos e, para cada bloco, um *upper bound* é armazenado. Isso permite que os algoritmos saltem os blocos de documentos com pontuações baixas para estarem na lista dos documentos mais relevantes à consulta. Dessa forma, esses algoritmos alcançam speedups significativos em relação aos algoritmos originais. Diversos trabalhos ofereceram técnicas sequenciais sobre esses algoritmos com o intuito de reduzir o tempo de latência do processamento de consultas (DIMOPOULOS; NEPOMNYACHIY; SUEL, 2013b; ROSSI et al., 2013; SHAN et al., 2012, 2012; MALLIA et al., 2017).

Rojas, Gil-Costa e Marin (2013) apresentaram uma proposta paralela do algoritmo BMW com duas etapas. O objetivo dessa proposta é reduzir o custo computacional e de comunicação do processamento de consulta em um ambiente distribuído, onde considera-se o estilo *master-slave*. A máquina *master* envia as consultas para as máquinas *slaves* para serem avaliadas e, assim, o sistema obtém os k documentos mais relevantes à consulta. Na primeira etapa do algoritmo, todos os nós enviam um conjunto de documentos para o *master*, sendo que esse número de documentos é menor que o número k (número de documentos que deseja-se retornar). Ao receber os documentos, o *master* detecta e descarta os nós cujos os documentos não são relevantes. Na segunda etapa, a máquina master requisita mais documentos para os nós restantes e, ao recebê-los, executa a junção dos resultados. O processamento em cada nó é realizado através da execução do algoritmo BMW que é efetivado em paralelo pela CPU, usando a abordagem de *heap-min* local, um por *thread*, e a de *heap-min* compartilhado.

O trabalho apresentado em (ROJAS; GIL-COSTA; MARIN, 2013) expôs um paralelismo de granularidade grossa para o processamento de consultas, em que as *threads* da CPU executam de forma sequencial o algoritmo BMW ao mesmo tempo. Por sua vez, as propostas oferecidas aqui nesta tese de doutorado são de granularidade grossa e fina, ou seja, a execução de uma instância dos algoritmos também é efetuada em paralelo, como, por exemplo, a manutenção da lista dos documentos mais relevantes, as ordenações e a movimentação dos ponteiros. As políticas de propagação de *threshold* aqui propostas podem ser facilmente utilizadas no algoritmo multi-threading proposto por Rojas, Gil-Costa e Marin (2013), substituindo a abordagem do *heap* compartilhado. Assim, o algoritmo pode obter um melhor desempenho ao evitar bloqueios adicionais para garantir acessos exclusivos e o valor do *threshold* pode ser atualizado rapidamente por ter o valor propagado entre todas as *threads*.

Bonacic *et al* (2015) apresentaram uma solução paralela para o processamento de lote de consultas em ambiente *multithreading* e estratégias de atualização de índice invertido enquanto processa as consultas. No processassamento de lote, é oferecido um escalonador de tarefas para definir dinamicamente o número de *threads* por consulta ao receber um lote. Essa predição é realizada por meio do número de termos das consultas e da quantidade de documentos nas listas invertidas desses termos. A partir dessas informações, as listas invertidas de todas as consultas são particionadas e adicionadas em uma fila de tarefas para evitar que as *threads* fiquem ociosas. Assim, essa solução do Bonacic *et al* (2015) foca somente no gerenciamento eficiente das consultas do lote e não detalha o paralelismo nos processamentos das consultas, como o particionamento das listas invertidas, gerenciamento da estrutura dos documentos identificados, compartilhamento de informações, a combinação das listas dos documentos mais relevantes e as barreiras de sincronizações. Em seus experimentos, utilizaram o algoritmo WAND (BRODER et al., 2003) para o processamento de consulta. Com isso, as políticas de propagação de *threshold* e o paralelismo de granularidade fina aqui oferecidos podem ser utilizados no projeto proposto no trabalho (BONACIC et al., 2015).

Em relação ao lote de consultas desse trabalho do (BONACIC et al., 2015), um escalonador que combina diversos atributos das consultas foi oferecido, de maneira que o particionamento de uma consulta é realizado de acordo com o tamanho da maior lista invertida. Como os atributos utilizados na proposta dos autores são somente das consultas, o escalonador proposto em (BONACIC et al., 2015) pode ser utilizado nas estratégias paralelas aqui propostas para o processamento de lote de consultas.

Trados (2015) avalia o uso de GPU no processamento de consultas nas máquinas de busca Web. A técnica de ranqueamento de documentos explorada é a de duas fases, sendo que o

trabalho foca somente na segunda fase do ranqueamento. A primeira fase, aplicação de modelos de classificação em cada documento das listas invertidas dos termos da consulta, é efetivada sequencialmente na CPU. Na segunda fase, aplicação de técnicas de aprendizado de máquina é realizada, propondo-se que seja processada na GPU. Nesta proposta, o trabalho propõe portar para a GPU as três principais etapas para classificar um documento a partir de aprendizado de máquina: extração de características, transformação de modelo de entrada e avaliação do modelo de classificação. Os experimentos alcançaram um speedup de 19× comparado com a classificação totalmente realizada na CPU.

Huang *et al.* (2017) propõem um algoritmo colaborativo CPU-GPU de processamento de lote de consulta, de modo que na CPU seja o processamento de uma consulta realizado sequencialmente. Os algoritmos WAND (BRODER et al., 2003) e o BMW (DING; SUEL, 2011) foram extendidos para formarem o algoritmo proposto. O algoritmo paralelo da GPU atribui a um bloco de *threads* a tarefa de gerar os resultados dos top-k documentos para uma única consulta do lote. As *threads* individuais dentro de um bloco geram os top-k documentos locais do subconjunto de docIDs correspondente ao bloco. Uma vez que todos os resultados top-k documentos locais são obtidos dentro de um bloco de *threads*, as *threads* mesclam os resultados top-k documentos locais de cada *thread* para calcular os *k* documentos finais do bloco de *threads*. Para completar essa operação, selecionaram o método sequencial *insert sort*. Uma vez que os resultados são completados em todas as consultas do lote, eles são transferidos para a CPU como um lote. As consultas são distribuídas entre CPU e GPU para evitar ociosidade dos processadores, de forma que somente as consultas curtas são enviadas para GPU. Os experimentos foram realizados com *k* igual a 1, 10, 20 documentos e o número de *threads* variou entre 32 a 512. Os melhores *speedups* apresentados ficaram em torno de $\sim 2\times$. O trabalho foi estendido para multi-GPUs. As propostas aqui expostas nesta tese de douturado diferem desse trabalho em vários aspectos. Primeiro, as estratégias do nosso trabalho exploram qualquer tipo de consulta, pois elas são constituídas por dois níveis de paralelismo e, no caso de lote de consultas, três níveis são alcançados. Os experimentos aqui realizados são levados ao valor de *k*, número de documentos retornados, igual a 128 documentos, considerando que esse valor é dominante para a maioria das máquinas de busca comerciais.

Liu, Wang, Swanson (2018) descrevem um sistema de recuperação de informação que combina dinamicamente o processamento da CPU e GPU para processar consultas individuais de acordo com as características das consultas e da memória da GPU e a sobrecarga do sistema. O sistema proposto utiliza no módulo de processamento de consulta algoritmos de intersecção de listas baseados em algoritmos paralelos de merge, especificamente no algoritmo paralelo para GPU proposto por Green, Mccoll e Bader (2012). Esses algoritmos de intersecção são mais

simples que os algoritmos de poda dinâmica por não trabalharem com aplicação de modelos de similaridades e, de forma mais direta, obtêm os documentos que estão presentes em todas as listas invertidas dos termos da consulta.

Capítulo 9

CONCLUSÃO

O uso da computação paralela no processo de identificar os documentos mais relevantes a uma dada consulta na arquitetura GPU foi investigado neste trabalho. Estratégias de paralelização foram propostas e analisadas para o processamento de uma única consulta e de lote de consultas. A análise e a experimentação foram guiadas nas duas direções existentes no que se referem à eficiência dos sistemas de processamento de consulta de uma máquina de busca, latência de resposta e vazão.

Duas estratégias de particionamento de documentos que se adequam bem à abordagem de algoritmos DAAT foram propostas. Na estratégia homogênea, as listas invertidas são particionadas entre os processadores de acordo com as posições dos identificadores de documento. Já na estratégia heterogênea, as partições foram obtidas por meio de geração de faixas de identificadores de documentos, assim as partições resultantes possuíam tamanhos diferentes. Versões paralelas de algoritmos de podas dinâmicas DAAT, WAND e MaxScore, foram propostas para realizar o processamento. Três políticas de propagação do *threshold*, denominadas de local, *Safe-R* e *Safe-WR*, foram essenciais para obtenção de desempenho dos algoritmos paralelos.

Os experimentos foram conduzidos por consultas sintéticas e registros de consultas de requisições reais. Nas consultas sintéticas, os experimentos examinaram a influência do tamanho das listas invertidas dos termos da consulta no tempo de resposta do processamento paralelo das consultas. As consultas dos registros foram utilizadas para experimentar a resposta dos algoritmos paralelos sobre um conjunto de consultas agrupadas. Os resultados mostraram que a estratégia homogênea permitiu melhores *speedups* por causa do balanceamento homogêneo de carga gerado entre os processadores. Além disso, os resultados apresentaram influência na acurácia dos resultados por esses algoritmos de modo que houve perdas de documentos relevantes durante o processo de busca. Por outro lado, os algoritmos heterogêneos retornaram os top-k documentos exatos e apresentaram desempenhos promissores. Uma vez que houve uma quan-

tidade suficiente de trabalho por bloco de *threads*, as políticas de propagação *Safe-R* e *Safe-WR* demonstraram um melhor desempenho frente à política local nos experimentos.

A investigação do uso da arquitetura da GPU no processo de identificar os documentos mais relevantes a uma dada consulta mostrou que a GPU é uma alternativa viável para se obter ganhos de desempenho nestes tipos aplicações de buscas textuais. Contudo, para isso, são necessárias estratégias de paralelismo que se adequam bem tanto à arquitetura quanto aos algoritmos de busca, como mostradas neste trabalho.

9.1 Limitações do Trabalho

As máquinas de busca atuais utilizam funções complexas para ordenar os documentos mais relevantes a uma dada consulta. Essas funções comumente dependem da combinação de centenas de características dos elementos do conjunto de documentos. Para implementar tais funções, as máquinas de busca dividem o processamento da consulta em duas fases. Na fase inicial, é usada uma função simples de ordenação para selecionar um conjunto de documentos candidatos. Na segunda fase, uma função de ordenação baseada em aprendizado de máquina é aplicada somente no conjunto de documentos candidatos obtidos na primeira fase, gerando uma nova ordenação final que é apresentada ao usuário. Considerando esse contexto, este presente trabalho focou somente na primeira fase desse processo de identificar documentos dada uma consulta.

As contribuições aqui apresentadas se limitaram em uma única máquina contendo uma GPU. A experimentação das implementações das estratégias paralelas foi conduzida de forma que o processamento de consulta realizou-se em paralelo somente na GPU e no índice invertido localizado na memória global da GPU.

Os algoritmos sequenciais fundamentais para se alcançar os objetivos deste trabalho foram os algoritmos de poda DAAT, em que as listas invertidas encontravam-se ordenadas de forma crescente aos identificadores de documentos e não compactadas. Essas soluções sequenciais de poda DAAT foram limitadas neste trabalho nos algoritmos WAND e MaxScore. As pontuações máximas dos termos (*upper bound*/*maxscore*) utilizadas nos algoritmos paralelos foram globais às listas invertidas e a lista de top-k documentos mais relevantes foram locais aos processadores de granularidade grossa.

9.2 Trabalhos Futuros

Nesta tese, paralelizações de processamento de consulta e de lote de consultas para GPUs foram desenvolvidas. A experimentação foi considerada em um único nó e se realizou somente em uma única GPU.

Para trabalhos futuros, pretendemos formalizar as soluções paralelas em um modelo paralelo para se obter previsões de custo de comunicação e de computação. Além disso, pretendemos estender as implementações para um ambiente paralelo com multi-GPUs e integrar o processamento paralelo de consultas na GPU com o processamento oferecido pela CPU, cujo o estado encontra-se ocioso nas partes paralelas dos algoritmos.

As estratégias aqui projetadas consideram que o índice invertido, ou parte dele, esteja inicialmente na memória global da GPU, assim não é levado em consideração o tempo de transferência de grandes porções de documentos, ou seja, das listas invertidas dos termos de uma dada consulta. Nessa direção, o trabalho pode ser levado a considerar esse tempo de transferência para cada consulta ou lote de consultas. Dessa forma, os impactos dessas transferências na eficiência do processamento de consulta e do lote de consultas podem ser analisados.

A investigação do processo de identificar os documentos mais relevantes a uma consulta em um ambiente paralelo considera a divisão de documentos a partir do qual se obteve as estratégias de particionamento das listas invertidas. Uma segunda frente de trabalho futuro relaciona-se à aplicação dessas estratégias em listas invertidas compactadas. Isso leva a um estudo de paralelização de algoritmos de poda que obtêm vantagens no desempenho ao considerar essa característica de compactação, por exemplo, o algoritmo *Block-Max WAND (BMW)* (DING; SUEL, 2011).

A paralelização dos algoritmos de poda gerou as políticas de propagação de *threshold*. Essas políticas podem ser levadas para um ambiente distribuído, pois elas podem reduzir o custo de comunicação entre os nós de processamento ao enviar somente o valor do *threshold* aos processadores, sem necessariamente produzir um *heap* global para aumentar a eficiência desses algoritmos. Já no contexto paralelo desses algoritmos, pode-se aplicar diferentes implementações do *upper bound*/*maxscore*, global e compartilhado, para realizar um estudo que verifica o impacto dessas implementações no desempenho.

No processamento de lote de consultas, escalonadores de consultas podem ser incluídos com as estratégias paralelas para realizar diferentes alinhamentos de consultas antes de enviar a uma dada GPU. Isso geraria uma análise de desempenho das estratégias frente a diferentes escalonadores de consulta. Além disso, esse estudo pode ser estendido para ambientes multi-GPUs e

ambientes CPU-GPU, sendo que este último considera a participação da CPU no processamento de consulta, de forma paralela ou não.

Um único lote de consultas foi investigado na experimentação das soluções aqui propostas. Uma extensão da análise do comportamento dos algoritmos de processamento de lote de consultas seria em um contexto de processamento de um conjunto de lotes de consultas armazenado em um buffer de memória. Essa extensão pode ser aplicada em um ambiente com uma única GPU, multi-GPUs e/ou CPU-GPU.

Todos os algoritmos foram analisados isoladamente no sistema de processamento das máquinas de busca. Considerando o processamento de consultas de duas fases utilizado por máquinas de busca complexas, em que a primeira fase é a aplicação de modelos de similaridade e a segunda é a execução de algoritmos de aprendizado de máquina nos resultados da primeira fase, uma direção de trabalho pode ser investigada na inclusão nesse contexto das estratégias paralelas propostas. Isso geraria novos experimentos e análises na segunda fase do processamento por ter a garantia de redução do tempo de processamento da primeira fase utilizando algoritmos paralelos na GPU.

REFERÊNCIAS

ALTINGOVDE, I. S.; OZCAN, R.; ULUSOY, Ö. Static index pruning in web search engines: Combining term and document popularities with query views. *ACM Transactions on Information Systems (TOIS)*, ACM, v. 30, n. 1, p. 2, 2012.

ALVES, C. E.; CÁCERES, E. N.; DEHNE, F. Parallel dynamic programming for solving the string editing problem on a cgm/bsp. In: ACM. *Proceedings of the fourteenth annual ACM symposium on Parallel algorithms and architectures*. [S.l.], 2002. p. 275–281.

AMATI, G.; JOOST, C.; RIJSBERGEN, V. Probabilistic models for information retrieval based on divergence from randomness. Citeseer, 2003.

ARAPAKIS, I.; BAI, X.; CAMBAZOGLU, B. B. Impact of response latency on user behavior in web search. In: ACM. *Proceedings of the 37th international ACM SIGIR conference on Research & development in information retrieval*. [S.l.], 2014. p. 103–112.

BAEZA-YATES, R.; RIBEIRO-NETO, B. et al. *Modern information retrieval*. [S.l.]: ACM press New York, 1999. v. 463.

BILLERBECK, B.; ZOBEL, J. Efficient query expansion with auxiliary data structures. *Information Systems*, Elsevier, v. 31, n. 7, p. 573–584, 2006.

BONACIC, C.; BUSTOS, D.; GIL-COSTA, V.; MARIN, M.; SEPULVEDA, V. Multithreaded processing in dynamic inverted indexes for web search engines. In: ACM. *Proceedings of the 2015 Workshop on Large-Scale and Distributed System for Information Retrieval*. [S.l.], 2015. p. 15–20.

BRODER, A. Z.; CARMEL, D.; HERSCOVICI, M.; SOFFER, A.; ZIEN, J. Efficient query evaluation using a two-level retrieval process. In: ACM. *Proceedings of the twelfth international conference on Information and knowledge management*. [S.l.], 2003. p. 426–434.

BUONO, D.; ARTICO, F.; CHECCONI, F.; CHOI, J. W.; QUE, X.; SCHNEIDENBACH, L. Data analytics with nvlink: An spmv case study. In: ACM. *Proceedings of the Computing Frontiers Conference*. [S.l.], 2017. p. 89–96.

BUSATO, F.; BOMBIERI, N. Bfs-4k: an efficient implementation of bfs for kepler gpu architectures. *IEEE Transactions on Parallel and Distributed Systems*, IEEE, v. 26, n. 7, p. 1826–1838, 2015.

BÜTTCHER, S.; CLARKE, C. L. A document-centric approach to static index pruning in text retrieval systems. In: ACM. *Proceedings of the 15th ACM international conference on Information and knowledge management*. [S.l.], 2006. p. 182–189.

BÜTTCHER, S.; CLARKE, C. L.; CORMACK, G. V. *Information retrieval: Implementing and evaluating search engines*. [S.l.]: Mit Press, 2016.

CAMBAZOGLU, B. B.; BAEZA-YATES, R. Scalability challenges in web search engines. *Synthesis Lectures on Information Concept, Retrieval, and Services*, Morgan & Claypool Publishers, v. 7, n. 6, p. 1–138, 2015.

CARMEL, D.; COHEN, D.; FAGIN, R.; FARCHI, E.; HERSCOVICI, M.; MAAREK, Y. S.; SOFFER, A. Static index pruning for information retrieval systems. In: ACM. *Proceedings of the 24th annual international ACM SIGIR conference on Research and development in information retrieval*. [S.l.], 2001. p. 43–50.

CHEETHAM, J.; DEHNE, F.; RAU-CHAPLIN, A.; STEGE, U.; TAILLON, P. J. Solving large fpt problems on coarse-grained parallel machines. *Journal of Computer and System Sciences*, Elsevier, v. 67, n. 4, p. 691–706, 2003.

CHEN, L.; HUO, X.; AGRAWAL, G. Accelerating mapreduce on a coupled cpu-gpu architecture. In: IEEE COMPUTER SOCIETY PRESS. *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*. [S.l.], 2012. p. 25.

CHENG, J.; GROSSMAN, M.; MCKERCHER, T. *Professional Cuda C Programming*. [S.l.]: John Wiley & Sons, 2014.

CORMEN, T. H.; LEISERSON, C. E.; RIVEST, R. L.; STEIN, C. *Introduction to algorithms*. [S.l.]: MIT press, 2009.

CROFT, W. B.; METZLER, D.; STROHMAN, T. *Search engines: Information retrieval in practice*. [S.l.]: Addison-Wesley Reading, 2010. v. 283.

CUTTING, D.; PEDERSEN, J. Optimization for dynamic inverted index maintenance. In: *Proceedings of the 13th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*. New York, NY, USA: ACM, 1990. (SIGIR '90), p. 405–411. ISBN 0-89791-408-2. Disponível em: ⟨http://doi.acm.org/10.1145/96749.98245⟩.

DEHNE, F.; FABRI, A.; RAU-CHAPLIN, A. Scalable parallel computational geometry for coarse grained multicomputers. *International Journal of Computational Geometry & Applications*, World Scientific, v. 6, n. 03, p. 379–400, 1996.

DIMOPOULOS, C.; NEPOMNYACHIY, S.; SUEL, T. A candidate filtering mechanism for fast top-k query processing on modern cpus. In: ACM. *Proceedings of the 36th international ACM SIGIR conference on Research and development in information retrieval*. [S.l.], 2013. p. 723–732.

DIMOPOULOS, C.; NEPOMNYACHIY, S.; SUEL, T. Optimizing top-k document retrieval strategies for block-max indexes. In: ACM. *Proceedings of the sixth ACM international conference on Web search and data mining*. [S.l.], 2013. p. 113–122.

DING, S.; HE, J.; YAN, H.; SUEL, T. Using graphics processors for high performance ir query processing. In: ACM. *Proceedings of the 18th international conference on World wide web*. [S.l.], 2009. p. 421–430.

DING, S.; SUEL, T. Faster top-k document retrieval using block-max indexes. In: ACM. *Proceedings of the 34th international ACM SIGIR conference on Research and development in Information Retrieval*. [S.l.], 2011. p. 993–1002.

FOLEY, D. Nvlink, pascal and stacked memory: Feeding the appetite for big data. *Nvidia. com*, 2014.

GAIOSO, R.; GIL-COSTA, V.; GUARDIA, H.; SENGER, H. A parallel implementation of wand on gpus. In: IEEE. *Parallel, Distributed and Network-based Processing (PDP), 2018 26th Euromicro International Conference on*. [S.l.], 2018. p. 10–17.

GREEN, O.; MCCOLL, R.; BADER, D. A. Gpu merge path: a gpu merging algorithm. In: ACM. *Proceedings of the 26th ACM international conference on Supercomputing*. [S.l.], 2012. p. 331–340.

HERLIHY, M.; SHAVIT, N. *The art of multiprocessor programming*. [S.l.]: Morgan Kaufmann, 2011.

HUANG, H.; REN, M.; ZHAO, Y.; STONES, R. J.; ZHANG, R.; WANG, G.; LIU, X. Gpu-accelerated block-max query processing. In: SPRINGER. *International Conference on Algorithms and Architectures for Parallel Processing*. [S.l.], 2017. p. 225–238.

JIANG, K.; YANG, Y.-x. Efficient dynamic pruning on largest scores first (lsf) retrieval. *Frontiers of Information Technology & Electronic Engineering*, Springer, v. 17, p. 1–14, 2016.

JONASSEN, S.; BRATSBERG, S. E. Efficient compressed inverted index skipping for disjunctive text-queries. In: *Advances in Information Retrieval*. [S.l.]: Springer, 2011. p. 530–542.

JONES, K. S.; WALKER, S.; ROBERTSON, S. E. A probabilistic model of information retrieval: development and comparative experiments: Part 2. *Information Processing & Management*, Elsevier, v. 36, n. 6, p. 809–840, 2000.

KARP, R. M.; RAMACHANDRAN, V. *Parallel Algorithms For Shared-Memory Machines, Handbook of Theoretical Computer Science (vol. A): Algorithms and Complexity*. [S.l.]: MIT Press, Cambridge, MA, 1991.

LIMA, A. C.; BRANCO, R. G.; CÁCERES, E. N.; GAIOSO, R. R.; FERRAZ, S.; SONG, S. W.; MARTINS, W. S. Efficient bsp/cgm algorithms for the maximum subsequence sum and related problems. *Procedia Computer Science*, Elsevier, v. 51, p. 2754–2758, 2015.

LIU, Y.; WANG, J.; SWANSON, S. Griffin: uniting cpu and gpu in information retrieval systems for intra-query parallelism. In: ACM. *Proceedings of the 23rd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. [S.l.], 2018. p. 327–337.

LONG, X.; SUEL, T. Optimized query execution in large search engines with global page ordering. In: VLDB ENDOWMENT. *Proceedings of the 29th international conference on Very large data bases-Volume 29*. [S.l.], 2003. p. 129–140.

MACDONALD, C.; SANTOS, R. L.; OUNIS, I.; HE, B. About learning models with multiple query-dependent features. *ACM Transactions on Information Systems (TOIS)*, ACM, v. 31, n. 3, p. 11, 2013.

MAGGS, B. M.; MATHESON, L. R.; TARJAN, R. E. Models of parallel computation: A survey and synthesis. In: IEEE. *System Sciences, 1995. Vol. II. Proceedings of the Twenty-Eighth Hawaii International Conference on*. [S.l.], 1995. v. 2, p. 61–70.

MALLIA, A.; OTTAVIANO, G.; PORCIANI, E.; TONELLOTTO, N.; VENTURINI, R. Faster blockmax wand with variable-sized blocks. In: ACM. *Proceedings of the 40th International ACM SIGIR Conference on Research and Development in Information Retrieval*. [S.l.], 2017. p. 625–634.

MANNING, C. D.; RAGHAVAN, P.; SCHÜTZE, H. et al. *Introduction to information retrieval*. [S.l.]: Cambridge university press Cambridge, 2008. v. 1.

MARON, M. E.; KUHNS, J. L. On relevance, probabilistic indexing and information retrieval. *Journal of the ACM (JACM)*, ACM, v. 7, n. 3, p. 216–244, 1960.

MATSUMOTO, K.; NAKASATO, N.; SEDUKHIN, S. G. Blocked all-pairs shortest paths algorithm for hybrid cpu-gpu system. In: IEEE. *High Performance Computing and Communications (HPCC), 2011 IEEE 13th International Conference on*. [S.l.], 2011. p. 145–152.

MENDOZA, M.; MARIN, M.; GIL-COSTA, V.; FERRAROTTI, F. Reducing hardware hit by queries in web search engines. *Inf. Process. Manage.*, v. 52, n. 6, p. 1031–1052, 2016.

MEYER, U.; SANDERS, P.; SIBEYN, J. *Algorithms for memory hierarchies: advanced lectures*. [S.l.]: Springer-Verlag, 2003.

OUNIS, I.; AMATI, G.; PLACHOURAS, V.; HE, B.; MACDONALD, C.; JOHNSON, D. Terrier information retrieval platform. In: SPRINGER. *European Conference on Information Retrieval*. [S.l.], 2005. p. 517–519.

ROBERTSON, S.; ZARAGOZA, H. *The probabilistic relevance framework: BM25 and beyond*. [S.l.]: Now Publishers Inc, 2009.

ROBERTSON, S. E.; JONES, K. S. Relevance weighting of search terms. *Journal of the American Society for Information science*, Wiley Online Library, v. 27, n. 3, p. 129–146, 1976.

ROBERTSON, S. E.; WALKER, S.; HANCOCK-BEAULIEU, M. M. Large test collection experiments on an operational, interactive system: Okapi at trec. *Information Processing & Management*, Elsevier, v. 31, n. 3, p. 345–360, 1995.

ROJAS, O.; GIL-COSTA, V.; MARIN, M. Efficient parallel block-max wand algorithm. In: *Euro-Par 2013 Parallel Processing*. [S.l.]: Springer, 2013. p. 394–405.

ROSSI, C.; MOURA, E. S. de; CARVALHO, A. L.; SILVA, A. S. da. Fast document-at-a-time query processing using two-tier indexes. In: ACM. *Proceedings of the 36th international ACM SIGIR conference on Research and development in information retrieval*. [S.l.], 2013. p. 183–192.

SALTON, G. The smart retrieval system—experiments in automatic document processing. Prentice-Hall, Inc., 1971.

SHAN, D.; DING, S.; HE, J.; YAN, H.; LI, X. Optimized top-k processing with global page scores on block-max indexes. In: ACM. *Proceedings of the fifth ACM international conference on Web search and data mining*. [S.l.], 2012. p. 423–432.

SKILLICORN, D. B. *Foundations of Parallel Programming*. [S.l.]: Cambridge University Press, 2005.

SKILLICORN, D. B.; TALIA, D. Models and languages for parallel computation. *ACM Computing Surveys (CSUR)*, ACM, v. 30, n. 2, p. 123–169, 1998.

SONG, X.; ZHANG, X.; YANG, Y.; QUAN, J.; JIANG, K. On structures of inverted index for query processing efficiency. In: *Information Retrieval Technology*. [S.l.]: Springer, 2015. p. 3–14.

STROHMAN, T.; TURTLE, H.; CROFT, W. B. Optimization strategies for complex queries. In: ACM. *Proceedings of the 28th annual international ACM SIGIR conference on Research and development in information retrieval*. [S.l.], 2005. p. 219–225.

TADROS, R. *Accelerating web search using GPUs*. Tese (Doutorado) — University of British Columbia, 2015.

TERRIER, I. *Platform*. 2014.

TURTLE, H.; FLOOD, J. Query evaluation: strategies and optimizations. *Information Processing & Management*, Elsevier, v. 31, n. 6, p. 831–850, 1995.

VALIANT, L. G. *Bulk-synchronous parallel computers*. [S.l.]: Harvard University, Center for Research in Computing Technology, Aiken . . . , 1989.

VALIANT, L. G. A bridging model for parallel computation. *Communications of the ACM*, ACM, v. 33, n. 8, p. 103–111, 1990.

VOLKOV, V. Better performance at lower occupancy. In: . [S.l.: s.n.], 2010.

WU, L.-L.; CHUANG, Y.-L.; CHEN, P.-Y. Motivation for using search engines: A two-factor model. *Journal of the American society for information science and technology*, Wiley Online Library, v. 59, n. 11, p. 1829–1840, 2008.

YAN, H.; DING, S.; SUEL, T. Inverted index compression and query processing with optimized document ordering. In: ACM. *Proceedings of the 18th international conference on World wide web*. [S.l.], 2009. p. 401–410.

ZHU, M.; SHI, S.; YU, N.; WEN, J.-R. Can phrase indexing help to process non-phrase queries? In: ACM. *Proceedings of the 17th ACM conference on Information and knowledge management*. [S.l.], 2008. p. 679–688.

ZOBEL, J.; MOFFAT, A. Inverted files for text search engines. *ACM computing surveys (CSUR)*, ACM, v. 38, n. 2, p. 6, 2006.