Universidade de Brasília
Instituto de Ciências Exatas
Departamento de Ciência da Computação

Algoritmos Paralelos e Eficientes para Consultas IP no Intel(R) Xeon Phi(tm) e CPUs Multi-Core

Alexandre Lucchesi Alencar

Dissertação apresentada como requisito parcial para
conclusão do Mestrado em Informática

Orientador
Prof. Dr. George Luiz Medeiros Teodoro

Brasília
2017

Ficha catalográfica elaborada automaticamente,
com os dados fornecidos pelo(a) autor(a)

Lucchesi Alencar, Alexandre
LAL368a Algoritmos Paralelos e Eficientes para Consultas IP no Intel(R) Xeon Phi(tm) e CPUs Multi-Core / Alexandre Lucchesi Alencar; orientador George Luiz Medeiros Teodoro. - Brasília, 2017.
67 p.

Dissertação (Mestrado - Mestrado em Informática) -- Universidade de Brasília, 2017.

1. Consultas IP. 2. Roteadores em Software. 3. Intel(R) Xeon Phi(tm). 4. Casamento de Maior Prefixo. I. Medeiros Teodoro, George Luiz, orient. II. Título.

Universidade de Brasília
Instituto de Ciências Exatas
Departamento de Ciência da Computação

Algoritmos Paralelos e Eficientes para Consultas IP no Intel(R) Xeon Phi(tm) e CPUs Multi-Core

Alexandre Lucchesi Alencar

Dissertação apresentada como requisito parcial para
conclusão do Mestrado em Informática

Prof. Dr. George Luiz Medeiros Teodoro (Orientador)
CIC/UnB

Prof. Dr. André C. Drummond
CIC/UnB

Prof. Dr. Dorgival O. G. Neto
CIC/UnB

Prof.ª Dr.ª Célia Ghedini Ralha
Coordenadora do Programa de Pós-graduação em Informática

Brasília, 29 de junho de 2017

Dedicatória

Eu dedico este trabalho às pessoas que eu mais amo e sem as quais ele não teria sentido: aos meus pais e ao amor da minha vida, Ingrid.

Agradecimentos

Primeiramente, a Deus, minha família e minha noiva pela compreensão e por todo o apoio dado durante esta fase da minha vida. Aos meus professores, em especial, ao professor George, pelo qual eu tive o privilégio de ser orientado, sempre com ótimos conselhos, puxões de orelha e direcionamentos que foram vitais para que este trabalho fosse mais longe. Aos bons amigos que fiz durante o Mestrado e às amizades antigas que se fortaleceram.

A todos vocês, o meu muito obrigado!

Resumo

Roteadores em software são uma solução promissora para lidar com o encaminhamento de pacotes devido ao seu bom custo-benefício e flexibilidade. Contudo, é desafiador o desenvolvimento de roteadores em software capazes de atingir as taxas de encaminhamento de pacotes necessárias. O uso de sistemas e técnicas de computação paralela pode ser uma abordagem viável para melhorar o desempenho dessas soluções. A fase de consulta IP constitui uma operação central no encaminhamento de pacotes, que é implementada através de um algoritmo de Casamento de Maior Prefixo (CMP). Assim, este trabalho propõe e avalia o uso de técnicas e processadores paralelos no desenvolvimento de um algoritmo otimizado que emprega filtros de Bloom (BFs) e tabelas *hash* para a execução de consultas IP. Especificamente, tem-se como alvo a implementação desse algoritmo no coprocessador *many-core* Intel® Xeon Phi™ (Intel Phi), mas também avalia-se o seu desempenho em CPUs *multi-core* e em um modelo de execução cooperativa que usa ambos os processadores com várias otimizações. Os resultados experimentais mostram que foi possível atingir altas taxas de consultas IP — até 182,7 Mlps (milhões de pacotes por segundo) ou 119,9 Gbps para pacotes IPv6 de 84B — em um único Intel Phi. Este desempenho indica que o Intel Phi é uma plataforma promissora para a implantação de algoritmos de consultas IP. Além disso, comparou-se o desempenho do algoritmo BFs com uma abordagem eficiente baseada na Multi-Index Hybrid Trie (MIHT), na qual o algoritmo BFs foi até 5,39× mais rápido. Esta comparação mostra que o algoritmo sequencial mais eficiente pode não ser a melhor opção em uma configuração paralela. Alternativamente, é necessário avaliar as características dos processadores, as demandas de computação/dados dos algoritmos e as estruturas de dados empregadas para analisar como os algoritmos podem se beneficiar de um dispositivo de computação paralelo, potenciais limitações na escalabilidade e oportunidades de otimização. Estas descobertas também são importantes para novos esforços no desenvolvimento de algoritmos nessa área, os quais têm sido, em sua maioria, focados em soluções sequenciais.

Palavras-chave: Consultas IP, Roteadores em Software, Intel® Xeon Phi™, Casamento de Maior Prefixo

Abstract

Software routers are a promising solution to deal with packet forwarding because of their good cost benefit and flexibility. However, it is challenging to develop software routers that can attain the required packet forwarding rates. The use of parallel computing systems and techniques may be a viable approach to improve the performance of these solutions. The IP lookup phase is a core operation in packet forwarding, which is implemented via a Longest Prefix Matching (LPM) algorithm to find the next hop address for every input packet. Therefore, this work proposes and evaluates the use of parallel processors and techniques in the development of an optimized algorithm that employs Bloom filters (BFs) and hash tables to the IP lookup problem. Specifically, we target the implementation on the Intel® Xeon Phi™ (Intel Phi) many-core coprocessor, but we also evaluate its performance on multi-core CPUs and on a cooperative execution model that uses both processors with several optimizations. The experimental results show that we were able to attain high IP lookup throughputs — up to 182.7 Mlps (million packets per second) or 119.9 Gbps for 84B IPv6 packets — on a single Intel Phi. This performance indicates that the Intel Phi is a very promising platform for deployment of IP lookup algorithms. We have also compared the BFs algorithm to an efficient approach based on the Multi-Index Hybrid Trie (MIHT) in which the BFs algorithm was up to 5.39× faster. This comparison shows that the most efficient sequential algorithm may not be the best option in a parallel setting. Instead, it is necessary to evaluate the processors characteristics, algorithms compute/data demands, and data structures employed to analyze how the algorithms will benefit from parallel computing devices, potential limitations on scalability and opportunities for optimizations. These findings are also important to new efforts in algorithmic developments in the topic, which have been highly focused on sequential solutions.

Keywords: IP Lookup, Software Routers, Intel® Xeon Phi™, Longest Prefix Matching

Sumário

Lista de Figuras

Lista de Tabelas

Capítulo 1

Introdução

O uso de roteadores em software é motivado pela sua extensibilidade, fácil programação e boa relação custo-benefício. No entanto, atingir altas taxas de encaminhamento de pacotes com esses roteadores é uma tarefa difícil [15], que pode ser viabilizada com o uso de algoritmos mais eficientes e/ou com a utilização de técnicas de paralelismo em computação de alto desempenho. O cálculo do endereço de próximo salto para endereços IP é uma operação central que ocorre durante a fase de encaminhamento de pacotes de rede nos roteadores. Com o surgimento da arquitetura Classless Inter-Domain Routing (CIDR), os roteadores precisam executar um algoritmo de Casamento de Maior Prefixo (CMP) para determinar o endereço de próximo salto para cada pacote recebido. O CMP consiste na execução de consultas a uma tabela de encaminhamento usando como chave o endereço de destino dos pacotes IP de entrada. Este processo é conhecido como consulta IP.

Neste trabalho, estuda-se o desempenho de algoritmos de CMP em plataformas modernas, como CPUs *multi-core* e o coprocessador *many-core* Intel® Xeon Phi™ (Intel Phi). O Intel Phi é uma plataforma altamente paralela que suporta até 272 *threads* executadas sobre 72 núcleos computacionais em *4-way hyperthreading* (modelo 7290). Ele também é equipado com instruções vetoriais "Single Instruction, Multiple Data" (SIMD) [37, 36, 52] de 512 bits e possui uma alta largura de banda de memória (500 GB/s no modelo 7250). O Intel Phi pode ser instalado em um computador como um coprocessador (através do barramento PCIe) ou pode ser usado sozinho, como um sistema independente, na nova geração Knights Landing (KNL). A possibilidade de se instalar o Intel Phi como um sistema independente foi um dos principais critérios que motivaram o seu uso neste trabalho, ou seja, a conexão de coprocessadores — Intel Phi ou Graphics Processing Units (GPUs) — à CPU através do barramento PCIe representa um grande gargalo para a obtenção de um alto desempenho em aplicações intensivas de dados, pois o PCIe limita a vazão da aplicação àquela do canal utilizado e adiciona um custo e atraso extras com essa transferência. Trabalhos anteriores que usaram GPUs para o processamento de consultas

IP discutiram esse problema [15, 19]. Diante disso, embora o número de núcleos computacionais encontrados nos coprocessadores Intel Phi atuais seja muito menor do que o de GPUs modernas, espera-se que o Intel Phi emerja como uma plataforma atrativa para o desenvolvimento e implantação de algoritmos paralelos e eficientes de consultas IP, principalmente em gerações futuras do coprocessador.

Assim, este trabalho faz um estudo da paralelização no Intel Phi de dois algoritmos de CMP conhecidos como eficientes. O primeiro algoritmo traz uma abordagem baseada no uso de filtros de Bloom e tabelas *hash* [9], enquanto o segundo utiliza uma estrutura de dados denominada Multi-Index Hybrid Trie (MIHT) [28]. Ambos os algoritmos suportam a construção e o gerenciamento de tabelas de encaminhamento dinâmicas tanto para o IPv4 quanto para o IPv6. Esses algoritmos foram selecionados porque (i) eles são conhecidos por serem algoritmos sequenciais eficientes e (ii) eles empregam diferentes estratégias e estruturas de dados para calcular o CMP e, como consequência, eles podem diferir em adequabilidade para a execução paralela. Resumidamente, a MIHT combina características-chave de árvores B^+ e Priority Tries [28]. Por outro lado, a abordagem com filtros de Bloom usa filtros de Bloom e tabelas *hash* [9]. Um filtro de Bloom é uma estrutura de dados probabilística para a realização de consultas de pertencimento em conjuntos [4]. Esta estrutura funciona como um filtro para evitar consultas a tabelas *hash* durante uma consulta IP, quando o segmento específico do endereço IP em questão não está armazenado no filtro. A escolha deste algoritmo foi motivada pelo fato de ele ser mais regular, isto é, filtros de Bloom e tabelas *hash* são estruturas de dados lineares, tornando o algoritmo potencialmente adequado para a execução em um sistema paralelo.

A implementação do algoritmo baseado em filtros de Bloom explora todos os recursos do Intel Phi para mitigar as principais desvantagens do algoritmo, que são o alto custo de computação de *hashes* durante operações de consulta ou armazenamento e os potenciais requisitos de largura de banda de memória. A estratégia adotada consiste no uso de vetorização para reduzir os custos de computação dos múltiplos *hashes* realizados pelo algoritmo e no uso de paralelismo em nível de tarefas para aumentar a concorrência e a vazão do sistema. Para avaliar o algoritmo baseado em filtros de Bloom, realizou-se uma comparação de desempenho com uma versão paralela da MIHT. A MIHT é um algoritmo sequencial eficiente que apresenta um desempenho melhor do que conhecidos algoritmos de consultas IP baseados em árvores ou *tries*: a Binary Trie, a Prefix Tree, a Priority Trie, a DTBM, a 4-MPT e a 4-PCMST [28]. A avaliação experimental das versões paralelas dos dois algoritmos implementados revelou que o algoritmo otimizado baseado em filtros de Bloom foi melhor que a MIHT tanto na execução sequencial quanto na execução paralela no Intel Phi. Em uma execução paralela usando diferentes conjuntos de prefixos IPv4 e IPv6, o algoritmo baseado em filtros de Bloom foi, respectivamente, 3,82× e 5,39×

mais rápido que a MIHT. Os resultados mostram que, embora a MIHT seja um algoritmo muito eficiente no que se refere ao uso de memória, ele apresenta menos oportunidades de otimização e pior escalabilidade no Intel Phi. Por exemplo, instruções vetoriais SIMD, disponíveis na maior parte das arquiteturas modernas de dispositivos computacionais, podem ser aproveitadas para melhorar o desempenho da abordagem baseada em filtros de Bloom, porém elas são de difícil aplicação para a MIHT devido à natureza irregular das estruturas de dados que compõem o algoritmo.

As abordagens utilizando a MIHT e filtros de Bloom alcançaram *speedups* de até 40× e 115×, respectivamente, em testes de escalabilidade no Intel Phi. As diferenças de escalabilidade ocorrem devido às características dos algoritmos que resultam em diferentes demandas de memória e oportunidades para otimizações. Consultas IP na MIHT envolvem operações de computação muito baratas, mas requerem vários acessos à memória durante a travessia das árvores. Isso resulta em uma baixa razão computação/acesso à memória, fazendo com que o gargalo do sistema seja a largura de banda de memória. Consequentemente, os ganhos obtidos com o aumento do número de núcleos computacionais fica limitado pela largura de banda para a MIHT. Por outro lado, a abordagem com filtros de Bloom é mais intensiva em computação, devido aos cálculos de *hash* dispendiosos que possibilitam reduzir o número de acessos à memória durante uma consulta. Como resultado, esta abordagem tem uma maior proporção computação/acesso à memória e uma melhor escalabilidade.

1.1 Problema

Roteadores em software baseados em PC constituem uma plataforma extensível, de fácil programação e de bom custo-benefício para o processamento de pacotes [15]. Embora a facilidade de programação e baixo custo sejam as principais motivações para o uso de roteadores em software, o fator desempenho é ainda um desafio.

1.2 Contribuições

Este trabalho avaliou o uso de arquiteturas híbridas, compostas de CPUs e coprocessadores Intel Phi, no processamento de pacotes de rede visando o desenvolvimento de roteadores em software de alto desempenho que atuem como alternativas viáveis às implementações em hardware em contextos específicos. Por meio de uma análise comparativa de algoritmos de encaminhamento IP em ambientes paralelos e através da exploração do potencial de paralelismo do Intel Phi, este trabalho estuda os compromissos de desempenho de versões paralelas e eficientes desses algoritmos.

As principais contribuições deste trabalho são sintetizadas a seguir:

- O projeto e a implementação de um algoritmo baseado em filtros de Bloom para a execução de consultas IP que explora completamente os recursos computacionais do coprocessador Intel Phi.
- Uma avaliação dos compromissos de se usar o coprocessador Intel Phi e CPUs modernas para a execução de consultas IP usando dois algoritmos que abordam o problema usando estratégias diferentes (filtros de Bloom e MIHT).
- Uma abordagem inovadora que combina as técnicas de Programação Dinâmica e Expansão Controlada de Prefixos (ECPPD) [46] para aprimorar o desempenho de consultas IPv6. Essa otimização resultou em ganhos de desempenho de até 5,9× no algoritmo baseado em filtros de Bloom.
- Uma avaliação experimental que mostra que o algoritmo sequencial mais eficiente pode não ser a melhor solução em um ambiente com alto poder de paralelismo. Até onde se sabe, este é o primeiro trabalho a avaliar sistematicamente o Intel Phi usando múltiplos algoritmos e arquiteturas de dispositivos para a execução de consultas IP.
- A proposição e a avaliação de um modelo de execução cooperativa que usa o Intel Phi e todas as CPUs *multi-core* disponíveis no sistema para o processamento simultâneo de consultas IP. Otimizando-se as transferências de dados que ocorrem pelo barramento PCIe, atingiu-se um *speedup* de aproximadamente 1,33× em relação às execuções que usam apenas o Intel Phi.

Essas contribuições foram relatadas nos artigos "Parallel and Efficient IP Lookup using Bloom Filters on Intel® Xeon Phi™" publicado na trilha principal do XXXV Simpósio Brasileiro de Redes de Computadores e Sistemas Distribuídos (SBRC 2017) e premiado como *best-paper* [29].

1.3 Organização do Texto

Este trabalho está organizado em cinco capítulos. O Capítulo 2 apresenta uma fundamentação teórica acerca dos aspectos gerais referentes a arquiteturas paralelas, redes de computadores e os trabalhos relacionados, essenciais para o entendimento deste trabalho. O Capítulo 3 descreve os dois algoritmos de encaminhamento IP implementados, otimizados e paralelizados neste trabalho: algoritmo filtros de Bloom e algoritmo Multi-Index Hybrid Trie (MIHT). O funcionamento de ambos os algoritmos é explicado juntamente com uma discussão envolvendo os detalhes de implementação e as estratégias de paralelização empregadas. O Capítulo 4 apresenta os resultados de desempenho obtidos por

meio de uma avaliação experimental dos algoritmos implementados, explicando como os seus respectivos parâmetros de configuração e como as características de hardware afetam o desempenho. Por fim, o Capítulo 5 apresenta as considerações finais, uma síntese das principais contribuições deste trabalho e as direções futuras.

Capítulo 2

Referencial Teórico

Para uma melhor compreensão acerca dos conceitos abordados nos capítulos subsequentes, este capítulo revisa os temas julgados necessários para um bom entendimento deste trabalho. Em primeiro lugar, uma discussão sobre os principais tópicos envolvendo redes de computadores, tais como conceitos arquiteturais, endereçamento e notações é apresentada. Em seguida, os tópicos referentes a arquiteturas de hardware paralelas são introduzidos. Por fim, os trabalhos relacionados são apresentados.

2.1 Redes de Computadores

Esta seção introduz conceitos básicos de redes, descrevendo a camada de rede e apresentando dispositivos denominados roteadores, que são elementos centrais em redes de computadores e cujas funções, sobretudo o encaminhamento de pacotes, constituem o foco deste trabalho.

2.1.1 Camada de Rede e Encaminhamento IP

A camada de rede da Internet é responsável por mover pacotes da camada de rede, conhecidos como datagramas, de um *host* para outro. Na pilha de rede do *host* de origem, a camada de rede recebe um pacote da camada de transporte e provê o serviço de entregá-lo à camada de transporte do *host* de destino de maneira automatizada. A camada de rede da Internet inclui o procotolo IP, que define os campos no datagrama e como os sistemas finais e roteadores devem agir sobre esses campos. Todos os componentes da Internet que possuem uma camada de rede devem executar o protocolo IP. A camada de rede da Internet também contém protocolos de roteamento, que determinam as rotas que um datagrama deve seguir entre *hosts* de origem e *hosts* de destino. A Internet é uma rede de redes e possui muitos protocolos de roteamento. Dentro de uma rede, um administrador

de redes pode executar qualquer protocolo que ele desejar ou definir as rotas manualmente, sem necessariamente executar um protocolo de roteamento. Embora a camada de rede abranja o protocolo IP e numerosos protocolos de roteamento, ela é geralmente referida simplesmente como *camada IP*, refletindo o fato de que o IP é o componente que liga os demais componentes da Internet. Em outras palavras, o encaminhamento de pacotes pode, em teoria, funcionar normalmente na ausência de protocolos de roteamento com a configuração manual das rotas nos roteadores.

A principal função de um roteador é encaminhar pacotes em direção ao seu destino final [30]. Para isso, um roteador deve decidir para onde enviar cada pacote de entrada. Precisamente, a decisão de encaminhamento consiste em dois componentes: (i) encontrar o endereço de próximo salto do roteador para o qual o pacote deve ser encaminhado; e (ii) determinar a interface de saída para a qual o pacote deve ser enviado. Esta informação de encaminhamento, referida como informação de *próximo salto*, é armazenada em uma tabela de encaminhamento populada, geralmente, a partir das informações obtidas pelos algoritmos de roteamento. É importante notar a interação entre as funções de encaminhamento e roteamento: os algoritmos de roteamento, quando usados, são responsáveis por determinar os valores (ou rotas) que devem ser inseridos nas tabelas de encaminhamento dos roteadores, enquanto os algoritmos de encaminhamento usam esses valores para decidir para onde enviar os pacotes recebidos. A tabela de encaminhamento é consultada utilizando-se como chave o endereço de destino do pacote de entrada, e esta operação recebe o nome de *consulta IP*. Uma vez que a informação de próximo salto é recuperada, o roteador pode transferir o pacote da interface de entrada para a interface de saída.

Em uma rede de datagramas IP, toda vez que um sistema deseja enviar um pacote, ele configura o cabeçalho do pacote com o endereço IP do sistema de destino e lança o pacote na rede. Conforme um pacote é transmitido de um nó de origem a um nó de destino, ele passa por uma série de roteadores. Cada um desses roteadores usa o endereço de destino do pacote para encaminhar o pacote. Especificamente, cada roteador tem uma tabela de encaminhamento que mapeia endereços de destino em interfaces de saída. Quando um pacote chega no roteador, o roteador usa o endereço de destino do pacote para consultar a interface do enlace de saída apropriada na tabela de encaminhamento do roteador. O roteador então encaminha o pacote para esta interface. Todo esse processo é conhecido como encaminhamento IP. O processamento realizado em cada roteador para mapear os endereços IP de destino em interfaces de saída envolve a execução de um algoritmo de CMP, cujo funcionamento é apresentado, em detalhes, na próxima seção.

2.1.2 Endereçamento IP e Casamento de Maior Prefixo

Antes de falar sobre endereçamento, é preciso definir o conceito de *interface*. Em uma rede, um *host* tipicamente tem apenas um único enlace que o conecta à rede. Toda vez que o *host* deseja enviar um datagrama ele utiliza o mesmo enlace. A fronteira entre um *host* e o enlace físico é conhecida como uma interface. Diferentemente de um *host*, um roteador possui múltiplas interfaces, uma para cada um dos seus enlaces. Como todo *host* ou roteador é capaz de enviar e receber datagramas IP, o protocolo IP requer que todas as interfaces dos *hosts* e dos roteadores tenham um endereço IP. Assim, um endereço IP é tecnicamente associado a uma interface, e não ao *host* ou roteador que contém aquela interface.

Cada endereço IPv4 possui tamanho igual a 32 bits (ou 4 bytes), formando um total de 2^{32} endereços possíveis, o que resulta em aproxidamente 4 bilhões de endereços IPv4 possíveis. No IPv4, os endereços são tipicamente escritos em uma notação decimal com pontos, na qual cada byte do endereço é escrito na sua forma decimal e é separado por um ponto dos outros bytes no endereço [23]. Por exemplo, considere o endereço IPv4 193.32.216.9. O 193 é o valor decimal equivalente aos primeiros 8 bits do endereço; o 32 é o valor decimal equivalente ao segundo byte do endereço, e assim por diante. Adicionalmente, os 32 bits que compõem um endereço IPv4 podem ser divididos em duas partes: uma parte que identifica a rede e outra parte que identifica o *host*. Assim, um endereço IPv4 representado usando-se a notação decimal com pontos da forma $a.b.c.d/x$ indica que os x bits mais significativos constituem a porção de rede do endereço, frequentemente referida como *prefixo de rede*. Os $32 - x$ bits restantes de um endereço são usados para distinguir os diferentes dispositivos ligados em uma rede. Esses bits de menor ordem são os bits que são considerados ao encaminhar pacotes dentro da própria rede, enquanto os bits de maior ordem (que formam o prefixo) são usados pelos roteadores para encaminhar pacotes entre redes diferentes [23]. Em uma rede, os endereços IP atribuídos a todos os *hosts* compartilham o mesmo prefixo, isto é, os bits que formam a parte mais à esquerda dos endereços são iguais. Considere a rede IPv4 identificada por: 223.1.1.0/24, onde a notação /24 indica que os 24 bits mais à esquerda dos endereços de 32 bits definem o endereço da rede. Qualquer dispositivo ligado à rede 223.1.1.0/24 receberá um endereço IPv4 da forma 223.1.1.xxx, onde "xxx" identifica os *hosts* individuais e deve ser único para cada *host* pertencente a essa rede.

No IPv6, os endereços possuem tamanho igual a 128 bits (ou 16 bytes) e são representados na forma canônica em 8 grupos de 16 bits cada, onde cada grupo é escrito como 4 dígitos hexadecimais e os grupos são separados por *dois pontos* (:) (ex.: 2001:0db8:0000:0000:0000:ff00:0042:8329). Por motivo de conveniência, endereços IPv6 podem ser abreviados em alguns casos usando-se uma notação mais concisa observando-

se as regras especificadas na RFC 5952 [40]. A RFC 7608 [41] determina que o tamanho de um prefixo IPv6 pode ser qualquer número entre entre zero e 128, embora algumas subredes que utilizam autoconfiguração *stateless* de endereços (SLAAC) para a alocação de endereços usem convencionalmente prefixos de 64 bits. Como trabalhos anteriores usaram prefixos de 64 bits nas avaliações de desempenho, essa convenção também foi adotada neste trabalho. Assim, os 64 primeiros bits de um endereço IPv6 são sempre usados para identificar a rede a qual o *host* pertence, embora os tamanhos de prefixos IPv6 possam ser menores do que 64. Além do tamanho dos endereços, no contexto de consultas IP e algoritmos de CMP, a única diferença relevante entre o IPv4 e o IPv6 é o tamanho máximo de um prefixo de rede. Especificamente, o tamanho máximo de um prefixo de rede no IPv4 é 32 bits e no IPv6 é 64 bits. Como no protocolo IPv4 todos os endereços são de 32 bits, uma implementação força-bruta de tabela de encaminhamento teria uma entrada para cada endereço de destino possível. Armazenar essa tabela como um vetor, e usar o endereço de destino para indexá-lo, demandaria 16 GiB de espaço de armazenamento. Para o IPv6, uma implementação força bruta seria inviável, atualmente, em termos de demanda de memória. Assim, é necessário o emprego de alternativas mais sofisticadas para reduzir os requisitos de memória para armazenar os prefixos de rede. O problema associado ao uso de prefixos de rede é que um endereço de entrada pode casar com múltiplos prefixos armazenados. Nesse caso, o roteador deve usar a regra de *casamento do maior prefixo* (CMP); isto é, o roteador deve encontrar o maior prefixo com o qual ocorreu um casamento com o endereço de entrada e encaminhar o pacote para a interface de saída associada a esse prefixo. Além disso, outro problema é que a redução nas necessidades de memória resulta em um aumento na complexidade computacional, uma vez que o CMP não exige somente que um casamento seja encontrado, mas que o melhor casamento (com o prefixo mais longo) seja encontrado.

Uma implementação eficiente do algoritmo de CMP é crucial para a obtenção de um bom desempenho em um roteador. A arquitetura de um roteador em hardware pode ser, basicamente, dividida em quatro componentes principais: portas de entrada, matriz de comutação, portas de saída e processador de roteamento [38]. Os pacotes são recebidos nas portas de entrada e enfileirados. A matriz de comutação conecta as portas de entrada às portas de saída, retirando os pacotes das filas de entrada, processando-os e enfileirando-os nas filas de saída. O processador de roteamento é responsável por atualizar as tabelas de encaminhamento. Nota-se que, nessa arquitetura, um desempenho ruim na matriz de comutação pode ocasionar retenção de pacotes nas filas de entrada e causar a perda de pacotes quando as filas estiverem cheias. Kurose et al. [23] alega que o desempenho demandado para efetuar o processamento dos pacotes é inviável para uma implementação em software. Considerando um enlace de entrada de 10 Gbps, datagramas IPv4 de 64 by-

tes e um atraso de apenas 51.2 ns decorrente do processamento de um datagrama por uma porta de entrada (antes que ela possa receber outro datagrama) — eles chegam a seguinte conclusão: se N portas são combinadas em uma placa de rede (como é frequentemente feito na prática), o *pipeline* de processamento de datagramas deve operar N vezes mais rápido, o que é muito rápido para uma implementação em software. Contudo, conforme apresentado na próxima seção, estudos recentes mostram que é possível mitigar o baixo desempenho associado a implementações em software com o uso de paralelismo.

2.2 Arquiteturas Paralelas

Esta seção apresenta conceitos básicos sobre arquiteturas paralelas, processadores vetoriais e as principais características do coprocessador Intel Phi, alvo deste trabalho.

2.2.1 Classificação de Arquiteturas de Computadores

Um sistema para classificação de arquitetura de computadores amplamente adotado em computação paralela é conhecido como taxonomia de Flynn [12]. Ele classifica um sistema de computação de acordo com o número de fluxos de instruções e fluxos de dados que esse sistema consegue gerenciar simultaneamente. Esta seção apresenta três classificações da taxonomia de Flynn comumente encontradas nos processadores atuais: Single-Instruction, Single-Data (SISD); Single-Instruction, Multiple-Data (SIMD); e Multiple-Instruction, Multiple-Data (MIMD).

Um sistema clássico que segue o modelo de von Neumann é um sistema que possui um único fluxo de instruções e um único fluxo de dados, pois, em um dado instante de tempo, ele executa apenas uma instrução e é capaz de recuperar ou armazenar apenas um item de dados de cada vez [36]. Portanto, um sistema de von Neumann é classificado como um sistema SISD. Por outro lado, sistemas que possuem um únixo fluxo de instruções e múltiplos fluxos de dados constituem sistemas paralelos conhecidos como SIMD. Sistemas SIMD operam em múltiplos fluxos de dados aplicando a mesma instrução a múltiplos itens de dados, de tal forma que um sistema SIMD pode ser abstraído como sendo um sistema que possui uma única unidade de controle e múltiplas unidades lógicas e aritméticas (ULAs). A mesma instrução é transmitida da unidade de controle para todas as ULAs, sendo que cada ULA ou aplica a operação ao item de dados corrente ou a ignora, ficando inativa. O paralelismo que é obtido por meio da divisão dos dados entre diferentes processadores que aplicam as mesmas instruções ao seu subconjunto de dados é chamado *paralelismo a nível de dados* (PND). Entre os processadores que fazem uso desse tipo de paralelismo, destacam-se os processadores vetoriais [36].

Além de SISD e SIMD, existem sistemas paralelos que suportam a execução simultânea de múltiplos fluxos de instrução e múltiplos fluxos de dados. Esses sistemas são conhecidos como MIMD e consistem, tipicamente, de uma coleção de núcleos ou unidades de processamento completamente independentes, os quais possuem suas próprias unidades de controle e ULAs. Além disso, diferentemente de sistemas SIMD, sistemas MIMD são geralmente *assíncronos*, isto é, os processadores nesses sistemas podem operar no seu próprio ritmo. Existem dois tipos de sistemas MIMD: sistemas de memória compartilhada e sistemas de memória distribuída. Em um sistema de memória compartilhada, uma coleção de processadores autônomos é conectada a um sistema de memória através de uma rede de interconexão, e cada processador pode acessar todas as posições de memória disponíveis. Em um sistema de memória compartilhada, os processadores geralmente se comunicam implicitamente acessando estrutura de dados compartilhadas. Em um sistema de memória distribuída, cada processador possui sua própria memória privada, e os pares processador-memória se comunicam por meio de uma rede de interconexão. Assim, nesses sistemas os processadores geralmente se comunicam explicitamente por trocas de mensagens ou usando funções especiais que provêem acesso à memória de outro processador [36]. O coprocessador Intel Phi, usado neste trabalho, pode ser classificado como um sistema MIMD de memória compartilhada, composto por 61 núcleos interconectados por uma rede em anel de alta velocidade. Este processador é apresentado em detalhes na próxima seção.

2.2.2 Intel® Xeon Phi™

O coprocessador Intel® Xeon Phi™ (Intel Phi) é baseado na arquitetura Intel® Many Integrated Core (MIC) e consiste de vários núcleos de processamento simplificados, ordenados e de baixo consumo energético, equipados com uma poderosa unidade de processamento vetorial de 512 bits (unidade SIMD). A arquitetura MIC combina características de CPUs *multi-core* de propósito geral e aceleradores de uso específico, como as GPUs, para prover um ambiente de alto desempenho e de fácil programação. Ela é baseada no conjunto de instruções x86 e suporta modelos tradicionais de programação em memória compartilhada para paralelização e comunicação entre os núcleos, tais como OpenMP (Open Multi-Processing), Pthreads (POSIX Threads Programming), MPI (Message Passing Interface) etc. Quando o Intel Phi é instalado como um coprocessador, as aplicações podem ser executadas em modo nativo ou em modo *offload*. Em modo nativo, o usuário acessa diretamente o coprocessador e executa a aplicação. Isso só é possível porque o Intel Phi executa um sistema operacional, ou seja, pode-se acessar o coprocessador a partir do *host* através de uma conexão Secure Shell (SSH) através do barramento PCI Express (PCIe). Em modo *offload*, o programador seleciona frações da aplicação, geral-

Tabela 2.1: Características dos (co)processadores Intel Phi utilizados.

Processador	Nome	Núcleos	Frequência	Largura de banda	Modo de execução
7120P	KNC	61	1,33 GHz	352 GB/s (GDDR5)	Coprocessador
7250	KNL	68	1,60 GHz	500 GB/s (MCDRAM)	*Standalone*

mente anotando o código-fonte com *pragmas* específicos, para serem automaticamente descarregadas e executadas no coprocessador em tempo de execução.

Neste trabalho, utilizou-se duas gerações do Intel Phi, cujas características são apresentadas na Tabela 2.1. O Intel Phi 7120P possui 61 núcleos em *four-way hyperthreading* suportando a execução simultânea de até 244 *threads* e um desempenho teórico de pico de até 1,2 teraflops. Cada núcleo opera a uma frequência máxima de 1,333 GHz, possui 512 KB de memória cache L1 privada e uma cache L2 de aproximadamente 30 MB (agregado de todos os núcleos) que é mantida em coerência entre os núcleos através de um mecanismo de rótulos de diretórios. Os núcleos computacionais, caches e controladores de memória são interconectados através de um anel bidirecional com alta largura de banda (352 GB/s). O Intel Phi 7120P possui 16 GB de memória principal GDDR5. A combinação de *hyperthreading* e ampla largura de banda de memória é eficiente para aliviar a latência de acesso à memória, beneficiando aplicações intensivas em dados. Por outro lado, o Intel Phi 7250 possui maiores capacidades computacionais e maior largura de banda de memória: são 68 núcleos operando a 1,60 GHz em *four-way hyperthreading* (execução de até 272 *threads*) e uma largura de banda de memória de 500 GB/s. Contudo, a característica mais notória do 7250, em relação ao 7120P, é que ele pode ser instalado em modo *standalone* como o processador principal da máquina. O 7120P deve ser conectado diretamente em um canal PCIe no *host*, podendo ser usado somente como um coprocessador. O modo *standalone* traz melhorias de desempenho cruciais para aplicações intensivas em dados porque ele remove o *overhead* de comunicação via PCIe presente no 7120P e também em GPUs.

2.3 Trabalhos Relacionados

Roteadores em software baseados em PC constituem uma plataforma extensível, de fácil programação e de bom custo-benefício para o processamento de pacotes [15]. Apesar da facilidade de programação e baixo custo serem as principais motivações para roteadores em software, o fator desempenho é ainda um desafio. Esta seção apresenta o estado atual e as principais contribuições relativas à paralelização de consultas IP de trabalhos anteriores envolvendo algoritmos de CMP e roteadores em software.

O desenvolvimento de algoritmos de CMP eficientes para implementação em roteadores em software é um tema extensivamente estudado. Contudo, um aspecto importante desses algoritmos é que eles são, em sua maioria, voltados para a execução sequencial, requerendo modificações para a implantação em ambientes paralelos. De forma geral, os algoritmos de consultas IP podem ser classificados em três categorias principais: (i) algoritmos que são construídos utilizando tabelas *hash* [5, 9]; (ii) algoritmos que utilizam *tries* binárias [16, 45, 28]; e (iii) algoritmos que realizam uma busca binária nos prefixos de rede [7, 55]. Algoritmos de consulta IP baseados em árvores ou *tries* são muito populares [44] e, geralmente, a busca pelo CMP nesses algoritmos envolve a travessia de um conjunto de nós de forma sequencial. Assim, os algoritmos dessa classe são projetados visando a redução do número de acessos à memória como forma de acelerar o processo de consulta. Para atingir esse objetivo, a Multi-Index Hybrid Trie (MIHT) [28], por exemplo, emprega estruturas de dados que são conhecidas por usarem a memória de modo eficiente — a saber: árvores B^+ e Priority Tries [27]. Recentemente, o uso de estruturas de dados comprimidas também foi proposto como uma alternativa para a otimização do uso de memória [39, 2]. Outra classe notória de algoritmos para consultas IP se baseia no uso de filtros de Bloom [9, 26] e tabelas *hash*. Ao contrário de esquemas baseados em árvores ou *tries*, que geralmente compartilham a característica de serem intensivos em memória, esses algoritmos são intensivos em computação e podem requerer diversos cálculos de *hash* durante as operações de consulta ou armazenamento. Os cálculos de *hash* são usados nos filtros de Bloom como forma de evitar acessos desnecessários a tabelas *hash*, onde os prefixos de rede são, de fato, armazenados.

Uma ampla variedade de arquiteturas de hardware também foi usada para a implementação de algoritmos de consultas IP, incluindo CPUs, FPGAs, GPUs e processadores *many-core* [54]. Este trabalho objetiva, primariamente, a avaliação do Intel Phi como alternativa às GPUs para a implementação de roteadores em software de alto desepenho. Assim, a Tabela 2.2 sintetiza as principais características de trabalhos anteriores que usaram GPUs para a execução de consultas IP. Embora os autores tenham usado hardware, dados e metodologias de teste diferentes em cada trabalho, esforçou-se para prover uma comparação justa entre eles [1]. Por exemplo, utilizando-se *buffers* de pacotes maiores, geralmente, obtém-se taxas de consulta também maiores. Contudo, *buffers* maiores também implicam em latências maiores. Diante disso, optou-se por mostrar na Tabela 2.2 apenas as maiores taxas de consulta alcançadas (ao se utilizar endereços de entrada aleatórios) e suas respectivas latências (no pior caso), conforme apresentado em cada artigo. Note que algumas implementações em GPU apresentam, em configurações

[1] O desempenho reportado em cada trabalho foi normalizado usando-se a métrica comum "milhões de consultas por segundo" (Mlps).

Tabela 2.2: Comparação entre trabalhos anteriores que usaram GPUs.

Artigo	Algoritmo	GPU	IPv4 (Mlps)	IPv6 (Mlps)	Transf. PCIe?	Latência
[31]	Radix Tree	GTX280	0,035	–	Yes	–
[54]	SAIL_L	Tesla C2075	547	–	Yes	152 μs
[8]	Multi-bit Trie	Tesla C2075	2.900	3.600	No	–
[15]	DIR-24-8-BASIC	2x GTX480	76,17	–	Yes	260 μs
[15]	Busca binária em HTs	2x GTX480	–	74,22	Yes	400 μs
[24]	GAMT	Tesla C2075	1.072	658	Yes	100 μs
[29]	Bloomfwd	Phi 7120P	169,6	182,7	No	–

específicas, altas taxas de consultas IP, mas elas incorrem em latências muito altas devido às transferências de dados entre o *host* e a GPU através do barramento PCIe. Observe também que o algoritmo Bloomfwd é uma contribuição deste trabalho, cujos detalhes de implementação e desempenho são discutidos nas seções a seguir.

Em [33], o algoritmo Parallel Bloom Filter (PBF) foi implementado no coprocessador Intel Phi. O PBF foi proposto com dois objetivos principais: reduzir o *overhead* de sincronização entre as *threads* e melhorar a localidade de *cache* em plataformas *many-core*. Assim como o PBF, no presente trabalho, também implementou-se um algoritmo que usa filtros de Bloom no Intel Phi. Entretanto, este algoritmo foi especializado para a execução de consultas IP, sendo diferente do PBF tanto a nível algorítmico quanto de implementação. O PBF usa *locks* nos filtros de Bloom para evitar *data races* em operações concorrentes de atualização ou consulta e garante a consistência sequencial através do reordenamento das respostas após o processamento das requisições (que ocorre em paralelo). Por outro lado, a abordagem implementada neste trabalho é construída sobre as ideias de [9] e inclui diversas otimizações visando a sua execução eficiente no Intel Phi. Em outras palavras, o PBF apresenta uma abordagem genérica para o processamento de requisições e o envio de respostas que impõe restrições que não são aplicáveis no contexto de consultas IP (ex.: uso de *locks*, reordenamento das respostas, entre outros), enquanto o algoritmo projetado foi otimizado especificamente para a execução de consultas IPv4 e IPv6. Conforme apresentado na avaliação experimental, as otimizações implementadas são cruciais para a obtenção de um alto desempenho.

Enquanto algoritmos de CMP sequenciais já foram amplamente estudados em diversos trabalhos, apenas alguns projetos [15, 10, 25] concentraram-se na execução paralela desses algoritmos em sistemas *multi-/many-core*. Além disso, não foi encontrado em nenhum trabalho o estudo dos compromissos de paralelização e execução dos algoritmos de CMP sequenciais em ambientes paralelos e, como consequência, não existe evidência de que um determinado algoritmo sequencial eficiente é a melhor escolha em um sistema *multi-/many-core*. No levantamento bibliográfico realizado, não foram encontrados ou-

tros trabalhos que estudaram o problema com essa abordagem. Entretanto, dois estudos chamam atenção por explorarem a utilização de sistemas paralelos para a construção de roteadores em software eficientes. O primeiro deles é o RouteBricks, que procurou aumentar a taxa de vazão de encaminhamento de pacotes usando um *cluster* de computadores. O segundo, denominado PacketShader, buscou otimizar o desempenho de entrada e saída de pacotes e acelerar o processamento dos pacotes usando GPUs. Existe também um terceiro estudo que, apesar de não ter sido voltado para a construção de roteadores em software de alto desempenho, merece ser citado. Trata-se do Click, que propôs uma arquitetura modular para roteadores em software e que atuou como precursor do RouteBricks.

O RouteBricks [10], apresenta um desempenho de encaminhamento próximo a 10 Gbps em uma única máquina para pacotes IPv4 de 64 bits. O encaminhamento IPv4 é baseado em consultas em tabelas de encaminhamento e, portanto, trata-se de um algoritmo intensivo em dados. Uma abordagem para o problema, adotada pelo PacketShader, consiste na utilização de General-Purpose Graphics Processing Units (GPGPU) para a implementação de roteadores em software [15]. A técnica busca aproveitar o paralelismo intrínseco em aplicações de rede que realizam o encaminhamento *stateless* de pacotes. Otimizando o mecanismo de I/O de pacotes e explorando o poder de paralelismo das GPUs, o PacketShader supera o desempenho do RouteBricks, atingindo taxas de 40 Gbps para pacotes IPv4 de 64 bits. No que se refere ao desempenho, o PacketShader apresenta implementações bem sucedidas das principais aplicações de rede [15]. As contribuições do PacketShader são duas: (i) otimização do caminho de dados através do desenvolvimento de um motor de entrada e saída de pacotes eficiente; e (ii) criação de um *framework* para possibilitar a aceleração com GPU de aplicações de rede. Em função das características das GPUs utilizadas, o PacketShader limita a execução de apenas um *kernel* [2] por dispositivo. Isso dificulta a definição de roteadores multifuncionais (ex.: que suportam IPv4 e IPsec ao mesmo tempo), pois impõe uma restrição onde todas as funções têm que ser implementadas no mesmo *kernel*. Por fim, o PacketShader restringe todo o poder de processamento da CPU para a entrada e saída de pacotes, descarregando efetivamente todos os pacotes para serem processados na GPU, independentemente do volume de tráfego. Utilizar a CPU para processar cargas de trabalho leves, por exemplo, pode resultar em menor latência no tratamento às requisições e maior eficiência energética. Adicionalmente, empregar a CPU e a GPU cooperativamente para processar os endereços de entrada pode resultar em um ganho extra de desempenho, conforme apresentado na Seção 4.6.

Para um desempenho ótimo, programas executados em GPUs devem seguir uma série de diretivas, como utilizar estruturas de dados lineares (ex.: vetores e tabelas *hash*, ao invés de árvores), minimizar os níveis de divergência no código, entre outros. Em

[2]Um *kernel* é um programa executado na GPU.

2010, o PacketShader apresentou esses desafios no contexto de roteadores em software e sugeriu que algumas das dificuldades poderiam ser aliviadas com o surgimento de novas arquiteturas de GPUs, como a Intel® Larrabee — uma arquitetura Multiple-Instruction Multiple-Data (MIMD) constituída de múltiplos núcleos x86 [15]. Contudo, o projeto Larrabee foi descontinuado ainda em 2010. Em 2012, a Intel® anunciou um novo projeto: o coprocessador Intel® Xeon Phi™ — baseado na arquitetura MIC, sucessora da Larrabee. Esta arquitetura consiste em vários núcleos de processamento leves, ordenados e de baixo consumo energético. Cada núcleo possui uma poderosa unidade de processamento vetorial de 512 bits (unidade SIMD) [18]. Além disso, a arquitetura MIC provê uma plataforma de fácil programação, sendo compatível com as mesmas linguagens, ferramentas, compiladores e bibliotecas que uma CPU.

O Click [22] apresenta uma arquitetura de software modular para a criação de roteadores em software. Roteadores Click são construídos a partir de componentes de baixa granularidade, possibilitando a aplicação de extensões ao longo do caminho de encaminhamento. Esses componentes são módulos de processamento de pacotes denominados elementos, que possuem uma interface simplificada e que podem ser conectados para a construção de roteadores. Baseado no Click e com o objetivo de maximizar o desempenho de roteadores em software, o RouteBricks [10] utilizou técnicas de paralelismo para propor uma nova arquitetura de roteadores. Assim como o Open vSwitch [34], o Click é um candidato para ser integrado ao algoritmo otimizado baseado em filtros de Bloom para consultas IP descrito neste trabalho.

Capítulo 3

Avaliação e Paralelização dos Algoritmos de Consulta

Esta seção descreve os algoritmos utilizados neste trabalho e discute suas implementações paralelas e eficientes. Os algoritmos escolhidos realizam as consultas IP usando (i) uma Multi-Index Hybrid Trie (MIHT), que é construída a partir de uma árvore B^+ e Priority Tries [28] ou (ii) filtros de Bloom juntamente com tabelas de *hash* [9] (abordagem filtros de Bloom). A abordagem MIHT é um método sequencial muito eficiente, tendo superado o desempenho de diversos algoritmos baseados em árvores [28]. Entretanto, sua natureza irregular limita as otimizações que podem ser empregadas, como o uso efetivo de instruções SIMD que estão disponíveis na maioria dos arquiteturas de processadores modernos. A abordagem filtros de Bloom, por outro lado, é conhecida por ser mais intensiva em computação, porém, consiste de algoritmos e estruturas de dados mais regulares. Dessa forma, esta abordagem pode se beneficiar mais com a paralelização e otimizações que visam processadores paralelos. O restante desta seção descreve os algoritmos de consulta e suas implementações eficientes visando o Intel Phi. As paralelizações na CPU usaram a mesma abordagem e os mesmos códigos-fonte do Intel Phi para paralelismo a nível de tarefas e auto-vetorização para utilização de instruções SIMD.

3.1 Abordagem com Filtros de Bloom e Tabelas Hash

O uso de filtros de Bloom acoplados a tabelas *hash* para a execução de consultas IP foi proposto por [9]. A versão mais básica do algoritmo utiliza 32 e 64 pares de filtros de Bloom e tabelas *hash*, respectivamente, para consultas IPv4 e IPv6. Dharmapurikar et al. também descreveu o uso de Filtros de Bloom com Contadores (CBFs) [11] para a construção de tabelas de encaminhamento dinâmicas e o uso da técnica Expansão Controlada de Prefixos (ECP) [46] para reduzir o número de pares de filtros de Bloom e

tabelas *hash* requeridos pelo algoritmo para o IPv4 para apenas 2 pares e um vetor de acesso direto (VAD). Por questões de simplicidade, explicaremos primeiro o algoritmo básico no contexto do IPv4. Em seguida, apresentam-se as modificações necessárias para a sua execução eficiente no Intel Phi tanto para o IPv4 quanto para o IPv6.

3.1.1 Descrição do Algoritmo

Um filtro de Bloom é uma estrutura de dados eficiente para consultas de pertencimento com taxas de erro por falso positivo configuráveis [4], amplamente utilizada para armazenamento em cache na Web, detecção de intrusão, roteamento baseado em conteúdo, e casamento de maior prefixo [9]. Em essência, um filtro de Bloom padrão consiste em um vetor de bits utilizado para representar de forma eficiente um conjunto de mensagens. Uma mensagem é armazenada nesta estrutura calculando-se funções de *hash* sobre ela e atribuindo-se o valor 1 aos bits endereçados pelos *hashes* resultantes no vetor de bits. Para verificar se uma mensagem de entrada está em um filtro de Bloom, o mesmo conjunto de funções *hash* é calculado sobre ela. A mensagem é dita como pertencente ao conjunto com uma determinada probabilidade se e somente se todos os bits no vetor de bits endereçados pelo resultado das funções de *hash* estiverem com valor 1. Um filtro de Bloom padrão não permite a remoção de mensagens, uma vez que não controla o caso em que um bit é ativado várias vezes devido a inserção de mensagens diferentes. Esse problema foi resolvido utilizando-se filtros de Bloom com contadores [11], que armazenam um vetor de contadores adicional associando um contador a cada bit do vetor de bits. Esses contadores são usados para armazenar o número de vezes que um determinado bit foi ativado ou desativado. Implementou-se essa abordagem a fim de permitir uma comparação justa com o algoritmo MIHT, que suporta a construção de tabelas de encaminhamento dinâmicas.

Como um filtro de Bloom é uma estrutura de dados para a execução de *buscas exatas*, as operações de consulta no algoritmo filtros de Bloom são realizadas utilizando-se múltiplos filtros — um para cada tamanho distinto de prefixo IP. Uma tabela *hash* é também associada com cada tamanho de prefixo ou filtro. As tabelas *hash* armazenam pares [prefixo, próximo salto] e qualquer outra informação de roteamento relevante, tais como métrica, interface, etc. Como o tamanho dos endereços de rede no IPv4 é 32 bits (vide Seção 2.1.2), é necessária a alocação no algoritmo de 32 filtros de Bloom com as respectivas 32 tabelas *hash*. Se uma rota padrão existir, que é equivalente a um prefixo de tamanho zero (ex.: 0.0.0.0/0), ela é armazenada em um campo separado na estrutura de dados da tabela de encaminhamento. Seja $F = \{(f_1, t_1), (f_2, t_2), \ldots, (f_{32}, t_{32})\}$ o conjunto de filtros de Bloom (f_i) e tabelas *hash* associadas (t_i) que formam uma tabela de encaminhamento IPv4, onde (f_1, t_1) corresponde às estruturas de dados que armazenam prefixos de 1 bit, (f_2, t_2) corresponde às estruturas de dados que armazenam prefixos de 2 bits, e assim por

diante. Além disso, seja $len(f_i)$ o tamanho do vetor de bits do i-ésimo filtro de Bloom, onde $1 \leq i \leq 32$. Observe que o vetor de bits e o vetor de contadores possuem o mesmo tamanho. No algoritmo, foram empregados filtros de Bloom assimétricos [9], que implicam que $len(f_i)$ pode ser diferente de $len(f_j)$, para $1 \leq i, j \leq 32$ e $i \neq j$. A motivação para filtros de Bloom assimétricos é a alocação ótima de memória para cada estrutura de dados de acordo com o número esperado de elementos a serem armazenados nela. A construção da tabela de encaminhamento é descrita a seguir. Para cada prefixo de rede p de tamanho l a ser armazenado, k funções de *hash* são computadas, produzindo k valores de *hash*: $H = \{h_1, h_2, \ldots, h_k\}$. O algoritmo usa H para ativar os k bits correspondentes aos índices $I = \{h_i \bmod len(f_l) \mid 1 \leq i \leq k\}$ no vetor de bits do filtro de Bloom f_l. Ele também incrementa os contadores correspondentes no vetor de contadores em f_l.

O processo de consulta é similar ao de inserção. Dada uma entrada contendo o endereço de destino DA, o algoritmo primeiro extrai os seus segmentos ou prefixos. Seja $S_{DA} = \{s_1, s_2, \ldots, s_{32}\}$ o conjunto de todos os segmentos de um endereço DA em particular, onde s_i é o segmento que corresponde aos primeiros $1 \leq i \leq 32$ bits de DA. Para cada $s_i \in S_{DA}$, k funções de *hash* são computadas, produzindo k valores de *hash* para cada segmento: $H = \{(h_1, h_2, \ldots, h_k)_1, (h_1, h_2, \ldots, h_k)_2, \ldots, (h_1, h_2, \ldots, h_k)_{32}\}$. Os filtros de Bloom são então consultados utilizando-se H. O elemento $H'_i \in H$ é usado para consultar o filtro de Bloom $f_i \in F$. O processo é o seguinte: o algoritmo verifica os k bits no vetor de bits de f_i usando os índices $I = \{h_j \bmod len(f_i) \mid h_j \in H'_i, 1 \leq j \leq k \text{ e } 1 \leq i \leq 32\}$. O resultado desse processo é um *vetor de casamentos* $M = \{m_1, m_2, \ldots, m_{32}\}$ contendo as respostas de cada filtro de Bloom, isto é, cada $m_i \in M$ indica se um casamento ocorreu ou não em f_i. O vetor de casamentos M é usado para consultar as tabelas *hash* associadas. A busca começa pela execução de consultas sequenciais às tabelas *hash*, percorrendo M de trás para frente, ou seja, começando em m_{32}, então m_{31}, e assim por diante. Isso ocorre porque o algoritmo precisa encontrar o maior prefixo com o qual ocorre um casamento. Se o algoritmo encontrar o próximo salto (um casamento) para determinado DA no par (f_i, t_i), então ele é o CMP. Como filtros de Bloom podem produzir falso-positivos mas nunca produzem falso-negativos, sempre que um filtro não casar com um segmento, ou seja, $m_i \in M$ indica *não-casamento*, o algoritmo pode seguramente pular para o próximo filtro de Bloom f_{i-1} (se $i \geq 2$) sem precisar consultar sua tabela *hash* associada t_i. Esse processo continua até que um casamento seja encontrado ou todos os filtros (para cada tamanho de prefixo) sejam consultados. Observe que falso-positivos levam apenas a buscas extras mal sucedidas, sendo o resultado do algoritmo o mesmo independentemente da taxa de falso-positivo. Os processos de inserção e consulta de endereços IP, descritos anteriormente, bem como as estruturas de dados utilizadas e suas interações no algoritmo filtros de Bloom são apresentados na Figura 3.1. O pseudocódigo de uma versão otimizada

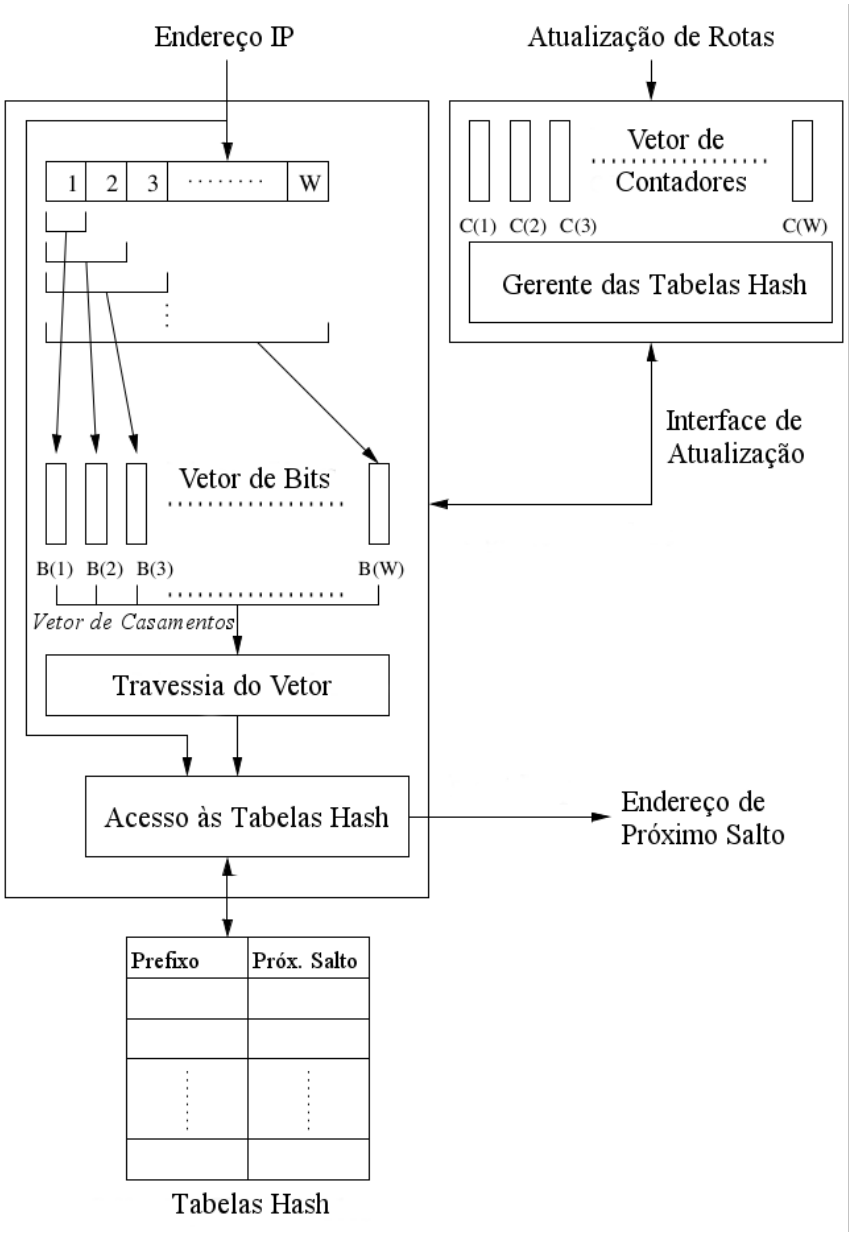

Figura 3.1: Algoritmo filtros de Bloom: estruturas de dados e interações para a inserção/consulta de prefixos de rede. W é igual ao tamanho dos endereços IP de entrada, sendo 32 para o IPv4. A inserção de um prefixo é um procedimento de atualização de rotas, que consiste em 3 etapas: (i) inserção do prefixo na tabela *hash* adequada (de acordo com o tamanho do prefixo); (ii) incrementar os respectivos contadores no vetor de contadores (endereçados pelo módulo dos cálculos de *hash* e o tamanho do vetor); e (iii) ativar os bits correspondentes no vetor de bits. Adaptada de [9].

do algoritmo de consulta é apresentado na Seção 3.1.2.

A probabilidade de falso positivo é um aspecto chave dos filtros de Bloom que é definida a partir de três parâmetros: o número n de mensagens armazenadas no filtro, o tamanho m do filtro, e o número k de funções de *hash* usadas para armazenar/consultar

uma mensagem. Dado um valor para n, os valores de m e k podem ser derivados para se atingir a taxa de falso-positivo desejada. Conforme descrito em [9], se a probabilidade de falso-positivo for fixada, então o tamanho do filtro, m, precisa crescer linearmente com o tamanho do conjunto de mensagens, n, para um dado k. A configuração apropriada desses parâmetros é crucial para se obter um bom desempenho. Uma avaliação detalhada acerca do impacto desses parâmetros é apresentada na Seção 4.2.

3.1.2 Paralelização e Detalhes de Implementação

Otimizações e Ajuste Fino do Algoritmo

A eficácia de um filtro de Bloom depende da probabilidade de ocorrência de falso-positivos. Conforme discutido na seção anterior, essa probabilidade é determinada por três parâmetros que podem ser configurados sem afetar os resultados do algoritmo. A escolha desses parâmetros foi feita da seguinte forma: uma probabilidade de falso-positivo desejada foi fixada em um valor e, sabendo-se o número total de prefixos a serem armazenados na tabela de encaminhamento para cada tamanho distinto de prefixo, o número de funções de *hash* e o tamanho de cada filtro de Bloom foram calculados. [9]. A avaliação experimental analisou o impacto do valor da taxa de falso-positivo no desempenho do algoritmo. Além disso, foram aplicadas duas otimizações para minimizar o número de cálculos de *hash* em uma operação de consulta ou inserção. A primeira otimização consiste no uso de duas funções de *hash* $h_1(x)$ e $h_2(x)$ para simular valores de *hash* adicionais da forma $g_i(x) = h_1(x) + i \times h_2(x)$. Esta técnica pode ser efetivamente aplicada aos filtros de Bloom e estruturas de dados relacionadas, tais como tabelas *hash*, sem perdas nas probabilidades assintóticas de falso-positivos [20]. A segunda otimização é o reúso de valores de *hash* calculados durante as consultas de pertencimento nos filtros de Bloom para endereçar as tabelas *hash* correspondentes. Assim, é possível evitar o cálculo de outro *hash* quando uma tabela *hash* precisar ser acessada. O Algoritmo 1 apresenta em detalhes essas otimizações. O algoritmo recebe como entrada o endereço de destino DA; os 32 filtros de Bloom (ou vetores de bits); as 32 tabelas *hash*; o número de funções de *hash* a serem calculadas em cada operação de consulta; e a rota padrão, contendo o endereço de próximo salto que deve ser utilizado por padrão. As linhas 1—2 correspondem à inicialização do algoritmo: a variável *encontrado* indica se o CMP foi encontrado e é inicializada com o valor Falso; i é o índice que determina o conjunto de estruturas de dados (filtros de Bloom e tabelas *hash*) que devem ser visitadas em cada iteração. A variável i é inicializada com o valor 32 porque as estruturas de dados devem ser visitadas em ordem, começando nas estruturas que armazenam os maiores prefixos até as que armazenam os menores prefixos (vide Seção 3.1.1). O laço na linha 3 é executado até o CMP ser encontrado ou após todas

Algoritmo 1: CONSULTA_BLOOM($DA, B, H, n, rota\_padrão$): algoritmo básico de consulta IPv4 usando filtros de Bloom e tabelas hash.

Entrada: DA {Endereço de destino}, B {Vetor de vetores de bits}, H {Vetor de tabelas hash}, n {Número de hashes} e $rota\_padrão$ {Endereço de próximo salto padrão}.
Saída: O endereço de próximo salto.

```
1  encontrado := Falso
2  i := 32
3  enquanto (¬encontrado) ∧ (i > 0) faça
       // p é o prefixo correspondente aos i primeiros bits de DA
4      p := DA[31 : (32 − i)]
       // Fase de consulta ao filtro de Bloom (vetor de bits).
5      vetor_bits := B_i
6      h_1 := hash(p) // Calcula o hash de p.
7      k := h_1 mod |vetor_bits|
8      talvez := vetor_bits_k
9      se talvez = Verdadeiro então
           // O bit correspondente ao primeiro cálculo de hash está ativo.
10         se n > 1 então
               // Verifica os demais bits.
11             h_2 := hash(h_1)
12             k := h_2 mod |vetor_bits|
13             talvez := vetor_bits_k
14             j := 2
15             enquanto (talvez = Verdadeiro) ∧ (j < n) faça
16                 k := (h_1 + j × h_2) mod |vetor_bits|
17                 talvez := vetor_bits_k
18                 j := j + 1
19             fim
20         fim
           // Fase de consulta à tabela hash.
21         se talvez = Verdadeiro então
22             tabela_hash := H_i
23             próximo_salto := CONSULTA_TABELA_HASH(tabela_hash, h_1, p)
24             encontrado := próximo_salto ≠ nil
25         fim
26     fim
27     i := i − 1
28 fim
29 se ¬encontrado então
30     próximo_salto := rota_padrão
31 fim
32 retorna próximo_salto
```

as estruturas de dados terem sido consultadas. Em cada iteração do laço, um segmento diferente do endereço de entrada DA é extraído na linha 4 e utilizado nas consultas. Por exemplo, na primeira iteração, onde $i = 32$, os 32 bits que compõem DA são atribuídos a p e utilizados; na segunda iteração, apenas os primeiros 31 bits de DA são utilizados, e assim por diante. Essa segmentação do endereço de entrada é ilustrada na Figura 3.1. Nas linhas 5—8, o filtro de Bloom correspondente a i-ésima iteração é consultado. Para

isso, calcula-se o endereço k do bit a ser verificado como o módulo entre o valor de *hash* calculado sobre p e o tamanho do vetor de bits. O conteúdo desse bit é atribuído à variável *talvez*, que representa a possibilidade de p estar ou não armazenado na tabela *hash* associada. Se o valor de *talvez* for Falso, então certamente p não está armazenado na tabela *hash*, e o algoritmo pode prosseguir para a próxima iteração (linha 27). Caso contrário, se o número de cálculos de *hash* n for maior que 1, o algoritmo verifica os demais bits do vetor de bits e atualiza o conteúdo de *talvez* (linhas 10—20). O número de cálculos de *hash* é um parâmetro de configuração do algoritmo filtros de Bloom. Nas linhas 21—25, o valor de *talvez* é novamente verificado e, quando ele é Verdadeiro, a tabela *hash* associada é consultada através da função CONSULTA_TABELA_HASH(). Note que um dos argumentos passados para a função CONSULTA_TABELA_HASH() é o primeiro valor de *hash* calculado sobre p: h_1. Isso possibilita o reúso do *hash* previamente calculado para consultar a tabela *hash*, quando a função de *hash* usada em ambas as estruturas (filtros de Bloom e tabelas *hash*) é a mesma. Note também que a variável i é o que associa um filtro de Bloom a uma tabela *hash*. O código de CONSULTA_TABELA_HASH() foi omitido por questões de simplicidade: ele simplesmente acessa uma lista encadeada ("balde") indexada pelo módulo do *hash* (h_1) e o número de baldes, e realiza uma travessia fazendo uma comparação elemento a elemento do valor armazenado em cada nó com a chave (p). Se a consulta à tabela *hash* retornar um endereço de próximo salto válido ($\neq nil$), então a variável *encontrado* recebe o valor Verdadeiro, encerrando a execução do laço. Caso contrário, tem-se a ocorrência de um falso-positivo e o algoritmo prossegue para a próxima iteração. Ao final da execução do laço, se o endereço de próximo salto não foi encontrado (*encontrado* = Falso), então ele recebe o valor da rota padrão. O valor final de *próximo_salto* é retornado e o algoritmo encerra sua execução.

Paralelismo a Nível de Tarefas (PNT)

A abordagem com filtros de Bloom expõe múltiplas oportunidades para paralelismo. Por exemplo, em [9] foi sugerido a execução de consultas em paralelo aos vários filtros de Bloom (associados com diferentes tamanhos de prefixos) para determinado endereço de entrada; estratégia mencionada como adequada para uma implementação do algoritmo filtros de Bloom em hardware. Nesta estratégia, uma passagem final é realizada para selecionar o filtro de Bloom associado ao maior prefixo no qual ocorreu um casamento com o endereço. A mesma abordagem poderia ser usada para uma paralelização baseada em software, disparando-se uma *thread* para consultar cada filtro de Bloom. Contudo, em um cenário real de uso do IPv4, os prefixos não são uniformemente distribuídos de acordo com os tamanhos de prefixo e, como consequência, é mais provável que um casamento ocorra com prefixos de tamanhos que concentram a maior parte dos endereços. Assim,

consultar todos os filtros de Bloom em paralelo pode não ser eficiente, porque a maioria das vezes os resultados dos filtros de Bloom associados com tamanhos menores que 24 bits não serão usados. Ao invés disso, é computacionalmente mais eficiente consultar de forma sequencial os filtros de Bloom começando naqueles associados com tamanhos maiores de prefixos. A outra opção para paralelismo a nível de tarefas, a qual é usada neste trabalho, é executar em paralelo a computação da consulta para múltiplas mensagens de entrada, atribuindo uma ou múltiplas mensagens para cada *thread* disponível. Dessa forma, é possível conduzir o processamento de cada endereço usando o algoritmo mais eficiente computacionalmente e melhorar a vazão do sistema através da computação concorrente da consulta IP para diferentes mensagens. Isso é possível porque o processamento de cada endereço é independente e, como tal, não é necessária a utilização de nenhum mecanismo de sincronização entre as computações executadas para diferentes mensagens. A implementação da paralelização nesse nível empregou a interface de programação Open Multi-Processing (OpenMP) [35], que foi usada para anotar o laço principal do algoritmo que itera sobre as mensagens de entrada em busca de seus respectivos endereços de próximo salto.

Vetorização

O uso de vetorização é importante para extrair todo o potencial de desempenho do Intel Phi (vide Section 2.2.2), que é equipado com uma unidade de processamento vetorial de 512 bits. As instruções SIMD foram usadas para computar eficientemente os valores de *hash* para múltiplos endereços de entrada ao mesmo tempo. O alvo da vetorização foram os cálculos de *hash* porque essa é a fase mais intensiva em computação do algoritmo. O trabalho original [9] e implementações anteriores de algoritmos que empregam filtros de Bloom para o problema do CMP [26, 33] não discutem suas decisões ou razões sobre as funções de *hash* utilizadas. Observe que a vetorização pode ser usada também durante a extração dos segmentos de um endereço de entrada que são verificados com cada filtro de Bloom. Esta otimização é, particularmente, eficiente para a implementação da versão básica do algoritmo apresentada anteriormente, que requer a segmentação de endereços IPv4 e IPv6 em 32 e 64 prefixos, respectivamente. Contudo, vetorizar essa operação na versão otimizada do algoritmo não resulta ganhos de desempenho devido ao emprego da técnica Expansão Controlada de Prefixos (ECP), apresentada na Seção 3.2.

Assim, neste trabalho, múltiplas funções de *hash* foram implementadas, vetorizadas e avaliadas: a MurmurHash3 [1] (Murmur), o método multiplicativo de Knuth [21] (Knuth), e uma função de *hash* denominada aqui como H2 [32] [1]. A Murmur é amplamente utilizada

[1] O nome H2 foi escolhido porque essa função de *hash* foi desenvolvida por Thomas Mueller, o principal desenvolvedor do software de código-aberto H2 Database Engine.

no contexto de filtros de Bloom, mas sua versão original recebe como entrada uma string de tamanho variável. A fim de melhorar sua eficiência, ela foi especializada neste trabalho para operar apenas com chaves inteiras de 32 bits (para o IPv4) e 64 bits (para o IPv6). A Knuth é uma função de *hash* simples da forma: $h(x) = x \times c \bmod 2^l$, onde c deve ser um multiplicador de ordem igual ao tamanho do *hash* (2^l), mas que não tenha fatores em comum com ele. Para uma saída de 32 bits, $l = 32$, e utilizou-se o número primo $c = 2.654.435.761$. A H2 recebe como entrada uma chave e mistura os seus bits realizando-se operações bit a bit, conforme apresentado no Algoritmo 2. O algoritmo primeiramente desloca a chave de entrada 16 bits à direita e realiza uma operação de ou-exclusivo usando o resultado obtido com a chave original. O resultado é então multiplicado pela constante 0x45d9f3b (em notação hexadecimal) e acumulado na chave. Este processo é executado duas vezes e o valor de *hash* resultante é obtido repetindo-se a primeira etapa na chave acumulada.

Algoritmo 2: Definição da função de hash H2.

Entrada: x {Uma chave inteira e sem sinal de 32 bits}.
Saída: O valor de hash calculado.

```
1 x := ((x ≫ 16 ⊕ x) × 0x45d9f3b
2 x := ((x ≫ 16 ⊕ x) × 0x45d9f3b
3 x := ((x ≫ 16 ⊕ x)
4 return x
```

É importante ressaltar que a constante 0x45d9f3b usada na H2 foi calculada pelo autor desta função de *hash* executando-se um programa de testes especial por muitas horas [2]. Este programa calcula o efeito avalanche (o número de bits da saída que mudam se um único bit da entrada for alterado; que deve ser, em média, 16 bits, para a H2), a independência das mudanças nos bits da saída (os bits de saída não devem depender um do outro), e a probabilidade de uma mudança em cada bit da saída se algum bit da entrada é alterado. Apesar de simples, essa função de *hash* se mostrou muito eficaz na prática [32].

Uma vetorização manual dessas funções de *hash* foi realizada usando as funções de baixo nível da biblioteca Intel® Intrinsics [17]. Os mecanismos de auto-vetorização disponíveis no compilador Intel® C Compiler também foram avaliados, mas os códigos vetorizados manualmente mostraram-se mais eficientes. Os recursos computacionais SIMD são usados no algoritmo para calcular os valores de *hash* para diferentes endereços de entrada em paralelo, acelerando as consultas.

[2]Endereço do código-fonte do programa: `https://github.com/h2database/h2database/blob/master/h2/src/test/org/h2/test/store/CalculateHashConstant.java`. Acessado em 30/05/2017.

3.2 Expansão Controlada de Prefixos para o IPv4

CIDR	Prefixo em bits	Iface
10.54.34.192/30	00001010 00110110 00100010 110000*	A

a) FIB contendo apenas uma rota de 30 bits (antes da ECP).

CIDR	Prefixo em bits	Iface
10.54.34.192/32	00001010 00110110 00100010 11000000	A
10.54.34.193/32	00001010 00110110 00100010 11000001	A
10.54.34.194/32	00001010 00110110 00100010 11000010	A
10.54.34.195/32	00001010 00110110 00100010 11000011	A

b) FIB equivalente após a aplicação da ECP para 32 bits.

Figura 3.2: Exemplo de Expansão Controlada de Prefixos (ECP) para o IPv4. A Figura a) ilustra uma tabela de encaminhamento (FIB) que mapeia um prefixo /30 na interface de saída "A". A Figura b) apresenta o efeito de expandir este prefixo em múltiplos prefixos /32. Note que as FIBs em a) e b) são equivalentes.

A abordagem básica do algoritmo filtros de Bloom, apresentada nas seções anteriores, emprega um filtro de Bloom e uma tabela *hash* para cada tamanho distinto de prefixo IP. Isso significa que são necessários 32 e 64 pares de filtros de Bloom e tabelas *hash* para representar todos os tamanhos possíveis no IPv4 e no IPv6, respectivamente. Como essas estruturas de dados são visitadas sequencialmente durante as operações de consulta (vide Seção 3.1.1), o algoritmo pode ser ineficiente, no caso médio, devido ao alto número de acessos à memória.

Existem duas técnicas descritas em [9] que são eficazes para limitar o pior caso do algoritmo e acelerar o cálculo do CMP para o IPv4 no algoritmo filtros de Bloom. A primeira consiste no uso de uma nova estrutura de dados: um Vetor de Acesso Direto (VAD) — para armazenar prefixos curtos. Um VAD permite usar parte dos bits que compõem o endereço de entrada para indexar um vetor em memória. Cada posição desse vetor armazena um endereço de próximo salto, possibilitando a determinação do CMP com apenas um acesso à memória. A construção de um VAD pressupõe o emprego da segunda técnica de otimização, conhecida como Expansão Controlada de Prefixos (ECP) [46, 9].

A ECP consiste na expansão de prefixos de tamanho menor para múltiplos prefixos equivalentes de tamanho maior, a partir do preenchimento de todas as possibilidades de sufixos de bits do prefixo original. Por exemplo, considere o prefixo $(10^*, A)$, que associa o prefixo de 2 bits 10^* à interface de saída A, e um VAD para armazenar prefixos de

3 bits contendo $2^3 = 8$ posições. A expansão do prefixo $(10^*, A)$ (de tamanho 2) para o tamanho 3 envolve atribuir o valor A a duas entradas do VAD, correspondentes aos prefixos 100^* e 101^*. Similarmente, todo prefixo de tamanho menor ou igual a 20 bits pode ser expandido para um conjunto de índices que endereçam posições válidas de um VAD com capacidade para armazenar até 2^{20} endereços de próximo salto (ou interfaces de saída). Note que a ECP é uma operação de pré-processamento aplicada individualmente sobre os prefixos de rede durante as inserções e/ou atualizações que resultam na tabela de encaminhamento. A Figura 3.2 mostra um exemplo de aplicação da ECP para uma tabela de encaminhamento IPv4.

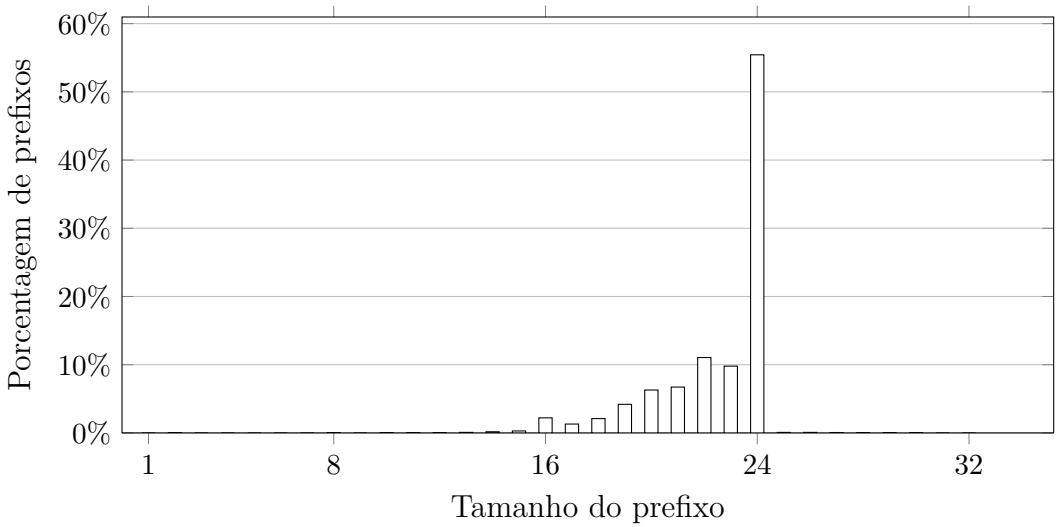

Figura 3.3: Distribuição de prefixos IPv4 na base de dados AS65000. O número total de prefixos únicos é 622,607 e apenas 1,625 são maiores do que 24 bits.

A ECP possibilita também a redução do número total de estruturas de dados necessárias para o correto funcionamento do algoritmo, pois permite agrupar prefixos de tamanhos distintos e armazená-los nos mesmos pares de filtros de Bloom e tabelas *hash*. Como pode ser visto na Figura 3.3, que apresenta a distribuição dos prefixos em uma base de dados IPv4 real, coletada do sistema autônomo AS65000 [3], os prefixos de rede não são distribuídos de maneira uniforme em relação aos tamanhos de prefixo possíveis. Além disso, nota-se que os conjuntos associados aos menores e/ou maiores tamanhos de prefixos estão praticamente vazios. Conforme observado em [14, 9], essa assimetria tende a se manifestar também em outras bases de dados de prefixos IPv4, que concentram a maior parte dos prefixos entre os tamanhos 16 e 24 bits.

Nas bases de dados estudadas em [9], por exemplo, 24.6% dos prefixos possuem tamanhos dentro do intervalo $[1, 20]$, 75.2% dos prefixos estão no intervalo $[21, 24]$ e 0.2% dos prefixos estão no intervalo $[25, 32]$. As bases de dados utilizadas na avaliação experimental (vide Seção 4.1) apresentam características similares a essas. Com base nesse padrão

do IPv4, Dharmapurikar et al. sugeriu o uso da ECP para melhorar o desempenho do algoritmo através da redução do número de tamanhos de prefixos distintos para níveis fixos. Especificamente, a otimização proposta consiste em garantir que existam apenas prefixos de tamanho 20, 24 ou 32 bits na tabela de encaminhamento. O primeiro grupo (prefixos de 20 bits) deve ser armazenado em um VAD, enquanto os outros dois grupos devem ser armazenados em pares diferentes de filtros de Bloom e tabelas *hash*: G_1 e G_2 — onde G_1 armazena os prefixos de 24 bits e, G_2, os prefixos de 32 bits. Com exceção do VAD adicionado, os processos de inserção e consulta após essa modificação permanecem idênticos aos da abordagem básica do algoritmo (explicada na Seção 3.1.1).

A construção da tabela de encaminhamento funciona da seguinte maneira. Seja p um prefixo de rede IPv4 e $len(p)$ o tamanho de p em bits. Seja $e_i(q)$ a operação que expande todo prefixo de rede q, tal que $len(q) \leq i$, em um ou mais prefixos de tamanho i. Para inserir ou atualizar qualquer prefixo p, se $p \leq 20$, armazena-se o endereço de próximo salto associado a p em cada posição do VAD indexada pelo resultado da aplicação de $e_{20}(p)$. A rota padrão, se existir, também é armazenada no VAD e deve preencher todas as posições "vazias" do vetor. Se $20 < p \leq 24$, aplica-se $e_{24}(p)$ e armazena-se o(s) prefixo(s) resultantes (juntamente com o endereço de próximo salto associado) em G_1. Por fim, se $24 < p \leq 32$, aplica-se $e_{32}(p)$ e o armazena-se o(s) prefixo(s) resultantes (juntamente com o endereço de próximo salto associado) em G_2.

Durante as consultas, o algoritmo realiza uma busca sequencial em G_2, G_1 e no VAD, nesta ordem. A busca começa em G_2 porque é onde os maiores prefixos estão armazenados. Se o CMP não for encontrado em G_2 ou G_1, o endereço de próximo salto armazenado na posição do VAD indexada pelos 20 primeiros bits do endereço de entrada é retornado (pode ser a rota padrão). Conforme detalhado em [9], com essa transformação no algoritmo, é possível limitar o cenário de pior caso das consultas IP a duas buscas (em G_2 e em G_1) e um acesso à memória (VAD). Observe que o compromisso da ECP são buscas mais rápidas sob o custo de um maior consumo de memória, como mostrado na Tabela 4.1.

3.3 ECP com Programação Dinâmica para o IPv6

Dharmapurikar et al. relatou que o uso da técnica ECP não era eficiente para o IPv6, devido às "distâncias" maiores entre os limites hierárquicos dos endereços IPv6, o que iria requerer um uso de memória muito alto após as expansões. Contudo, neste trabalho, propôs-se e implementou-se um algoritmo baseado em programação dinâmica (ECPPD) para agrupar tamanhos de prefixos distintos e realizar as expansões com uma demanda adicional de memória limitada. Diferentemente do algoritmo para o IPv4 (apresentado na seção anterior), que sempre usa os tamanhos 20, 24 e 32 como níveis fixos de expansão,

a ECPPD estima e escolhe para cada base de dados os níveis que resultam no menor consumo de memória possível. A partir desses níveis, o algoritmo aloca um filtro de Bloom e uma tabela *hash* para cada nível escolhido, executa a ECP sobre a base de dados de prefixos e armazena os prefixos expandidos nas estruturas de dados correspondentes. Cada prefixo é expandido para o nível imediatamente posterior ao seu tamanho, dentre os níveis escolhidos. Se o tamanho de um prefixo casa com um nível escolhido, então ele não precisa ser expandido e é armazenado diretamente. No caso do IPv6, geralmente, o uso de um VAD não é eficiente porque o número de prefixos IPv6 em bases de dados reais ainda é muito pequeno, fazendo com que o vetor fique, em sua maioria, vazio. Contudo, nada impede que um VAD ou outra estrutura de dados seja incorporada ao algoritmo no futuro de forma completamente transparente à ECPPD.

O algoritmo ECPPD funciona da seguinte maneira. Seja $L = \{l_1, l_2, \ldots, l_{64}\}$ a distribuição de prefixos por tamanho de uma tabela de encaminhamento IPv6, onde l_i é o número de prefixos únicos de tamanho i (em bits). Dado um número desejado de níveis de expansão n (ou número alvo de pares de filtros de Bloom e tabelas *hash*), o algoritmo usa programação dinâmica para computar o conjunto de tamanhos a ser usado com o objetivo de minimizar o número total de prefixos na tabela de encaminhamento resultante. A ECPPD sempre começa escolhendo o tamanho 64, pois este é o maior tamanho de prefixo possível para o IPv6 e, portanto, sua inclusão é necessária para a corretude do algoritmo, ou seja, todo prefixo IPv6 pode, teoricamente, ser expandido para um ou mais prefixos de 64 bits. Seja $S = \{64\}$ o conjunto inicial de tamanhos de prefixo resultantes e $C = \{1, 2, \ldots, 63\}$ o conjunto inicial de prefixos candidatos. Enquanto $|S| < n$, o algoritmo iterativamente remove um elemento $l \in C$ e o insere em S. Em cada iteração, o tamanho l é selecionado mapeando-se uma função de custo f sobre todos os possíveis conjuntos de tamanhos e escolhendo-se o tamanho associado ao menor custo. Por exemplo, na segunda iteração (assumindo $n \geq 2$), f é mapeada sobre o conjunto $Q = \{\{l, 64\} \mid l \in C\}$ e o valor l do conjunto que resultou no menor custo é selecionado. A função de custo f recebe como entrada L e um conjunto de níveis de expansão $Q' \in Q$. Ela então computa o número resultante de prefixos ao expandir L para Q'. O número máximo de prefixos, resultante da expansão de um tamanho de prefixo $l_i \in L$ para $q \in Q'$ (tal que $i < q$), é definido como $2^{q-i} \times l_i$. Observe que f não leva em consideração o problema da captura de prefixos [53], que acontece quando um prefixo é expandido para um ou mais prefixos que já existem na base de dados. Neste caso, os prefixos maiores já existentes "capturam" o prefixo expandido, que é ignorado. Assim, embora não seja possível garantir que a ECPPD retorna a solução ótima, ela geralmente retorna soluções que funcionam melhor na prática para o algoritmo baseado em filtros de Bloom do que quando se usa diretamente a base de dados sem nenhum pré-processamento, conforme

apresentado na Seção 4.5.

3.4 Execução Cooperativa e Transferência de Dados

Nesta seção, apresenta-se uma variação do algoritmo baseado em filtros de Bloom que usa a CPU e o Intel Phi cooperativamente para realizar consultas IP quando o Intel Phi está instalado como um coprocessador (Intel Phi 7120P). Neste caso, os endereços IP são armazenados na memória do *host* e são transferidos em *batches* através do barramento PCIe para serem processados no Intel Phi. Então, depois que o algoritmo de consulta é executado no coprocessador, os endereços de próximo salto computados são copiados da memória do dispositivo de volta para a memória do *host*. Assim, os tempos de transferência de dados podem impactar significativamente o desempenho geral da aplicação, e estratégias para reduzir o custo das transferências devem ser usadas. Além disso, a disponibilidade da CPU cria também oportunidades para se aproveitar este processador como um dispositivo de computação adicional para realizar consultas IP.

As transferências de dados necessárias para se executar uma computação no Intel Phi são tipicamente realizadas de maneira síncrona (*sync*), colocando-se em um *pipeline* de execução: (i) a cópia dos dados de entrada (do *host* para o coprocessador); (ii) a computação a ser executada sobre os dados de entrada; e (iii) a cópia dos resultados calculados (do coprocessador para o *host*). Para reduzir o tempo que a CPU e o Intel Phi ficam ociosos durante as transferências de dados, implementou-se uma estratégia de transferência assíncrona (*async*) usando um esquema com dois *buffers* de dados. Nesta estratégia, enquanto o Intel Phi está na fase de computação do *pipeline sync*, inicia-se concorrentemente a transferência de dados de outro *batch* de endereços IP para um segundo *buffer* no coprocessador. Dois *buffers* de dados são alocados também no *host*, e esse mesmo procedimento ocorre também durante as transferências dos resultados computados no Intel Phi, ou seja, uma segunda computação pode ser iniciada no Intel Phi enquanto os resultados da anterior estão sendo copiados para a memória do *host*.

Para usar cooperativamente os dois dispositivos, divide-se os endereços IP de entrada em dois conjuntos que são processados independentemente e concorrentemente pela CPU e pelo Intel Phi. A divisão de trabalho deve ser realizada visando minimizar o desbalanceamento de carga entre os processadores. Caso contrário, um processador pode levar muito mais tempo para processar o seu conjunto de endereços, e isso pode prejudicar os potenciais ganhos de desempenho desse tipo de abordagem. Para realizar o particionamento dos dados, leva-se em consideração o desempenho relativo dos processadores, que deve ser equivalente aos tamanhos relativos dos conjuntos de endereços IP atribuídos para computação em cada dispositivo. O desempenho relativo é computado em uma etapa de

profiling que antecede a execução do algoritmo de consulta. Adicionalmente, para evitar que as *threads* de CPU que estão computando consultas IP interfiram com a *thread* de CPU responsável por gerenciar o Intel Phi e as transferências de dados, um núcleo físico da CPU é alocado exclusivamente para executar a segunda função. Dessa forma, na configuração usada no sistema onde os experimentos foram executados (vide Seção 4.1), apenas 15 dos 16 núcleos físicos de CPU são usados para a execução de consultas IP, e 30 *threads* são disparadas nestes núcleos (aproveitando o mecanismo de *Hyper-Threading*). Essa abordagem é similar à utilizada em outras soluções para áreas de aplicação distintas [51, 50, 48].

3.5 Abordagem com Multi-Index Hybrid Trie

3.5.1 Descrição do Algoritmo

Este algoritmo utiliza uma estrutura de dados conhecida como Multi-Index Hybrid Trie (MIHT) [28]. A MIHT foi construída combinando as vantagens de árvores B^+ e Priority Tries [27] para projetar tabelas de encaminhamento dinâmicas. Uma MIHT consiste em uma árvore B^+ e múltiplas Priority Tries. Uma árvore B^+ é uma generalização de uma árvore de busca binária na qual um nó pode ter mais do que dois filhos. Uma árvore B^+ de ordem m é uma árvore ordenada que satisfaz as seguintes propriedades: (i) cada nó tem no máximo m filhos; (ii) cada nó, exceto a raiz, tem pelo menos $\frac{m}{2}$ filhos; (iii) a raiz tem pelo menos 2 filhos; (iv) todas as folhas ocorrem no mesmo nível; e (v) a informação satélite é armazenada nas folhas e apenas chaves e ponteiros para os filhos são armazenados em nós internos. Embora Priority Tries possam ser usadas sozinhas para construir tabelas de encaminhamento dinâmicas [27], elas são usadas na MIHT como subestruturas auxiliares para a construção de um algoritmo mais eficiente. Uma Priority Trie é similar a uma *trie* binária, na qual cada nó tem no máximo dois filhos e os nós a serem visitados durante a travessia da estrutura, em uma operação de busca, são determinados com base nos bits da chave (nesse caso, o endereço de entrada). Contudo, Priority Tries têm duas vantagens principais em relação às *tries* binárias no contexto de consultas IP. Primeiro, em uma Priority Trie, prefixos são atribuídos de modo reverso, ou seja, prefixos maiores são associados com nós em níveis mais altos e prefixos menores são associados com nós em níveis mais baixos na estrutura, possibilitando o encerramento imediato da busca assim que um casamento ocorre (uma *trie* binária sempre requer a travessia até uma folha). Segundo, de forma contrária à *trie* binária, não existem nós internos vazios em uma Priority Trie — todo nó armazena informação de roteamento para aprimorar o uso de memória. O Algoritmo 3 apresenta o processo de consulta em uma Priority Trie. O

Algoritmo 3: CONSULTA_PRIORITY($s, raiz, rota\_padrão$): algoritmo de consulta em uma Priority Trie. Adaptado de [27].

Entrada: s {Valor a ser consultado}, *raiz* {Raiz da Priority Trie} e *rota_padrão* {Endereço de próximo salto padrão}.
Saída: O endereço de próximo salto.

```
1  próximo_salto := rota_padrão
2  x := raiz
3  L := 0 // Marca o nível corrente durante a travessia na árvore.
4  repita
5      se ocorrer casamento entre s e o valor armazenado em x então
6          próximo_salto := o endereço de próximo salto associado a x
7          se x for um nó prioridade então
               // O endereço de próximo salto encontrado corresponde ao CMP.
8              retorna próximo_salto
9          fim
10     fim
11     L := L + 1
12     y é o filho do nó x identificado pelo L-ésimo bit de s
       // 0:  filho à esquerda, 1:  filho à direita.
13     x := y
14 até x ser um nó inválido
15 retorna próximo_salto
```

algoritmo recebe como entrada um valor s a ser consultado, um ponteiro para a raiz de uma Priority Trie e a rota padrão, e retorna o endereço de próximo salto. Nas linhas 1—3, as variáveis são inicializadas: o endereço de próximo salto é definido como a rota padrão; a variável x, que aponta para o próximo nó a ser visitado, recebe a raiz da Priority Trie; e L, que contém o nível corrente da árvore durante uma travessia é inicializado com 0 (nível da raiz). O laço na linha 4 é executado até o CMP ser encontrado ou até um nó-folha ser visitado (quando x passa a apontar para um nó inválido). Quando ocorre um casamento entre s e um valor armazenado em um nó-prioridade, o CMP é imediatamente determinado. Nesse caso, não é necessário continuar a travessia da árvore e o algoritmo retorna o endereço de próximo salto associado ao nó-prioridade (linha 8). Quando não ocorre um casamento entre s e o valor armazenado no nó corrente x ou quando ocorre um casamento, mas x não é um nó-prioridade, o algoritmo seleciona o próximo nó a ser visitado, y, a partir do L-ésimo bit de s. Conforme mencionado anteriormente, esse processo é similar ao que ocorre em uma *trie* binária, ou seja, se o L-ésimo bit de s for 0, y é igual ao filho à esquerda de x. Caso contrário, y é igual ao filho à direita de x. O valor de x é então atualizado e o processo se repete enquanto x for um nó válido, isto é, a estrutura não tiver sido completamente varrida.

Para construir uma tabela de encaminhamento usando a MIHT, cada prefixo de rede é dividido em duas partes: um prefixo (a chave) e um sufixo. Seja $p = p_0 p_1 \dots p_{l-1}^*$

um prefixo de rede e $q = p_i p_{i+1} \dots p_{l-1}$ para $0 \leq i \leq l-1$ um sufixo de p. O tamanho de um prefixo p, denotado $len(p)$, é o número de símbolos diferentes de *. Por exemplo, $len(p_0 p_1 \dots p^*_{l-1}) = l$. Para um inteiro $k \leq l$, a k-ésima chave/prefixo de p, denotada $prefix\_key_k(p)$, é o valor $(p_0 p_1 \dots p_{l-1})_2$. O k-ésimo sufixo de p, denotado por $suffix_k(p)$, é definido como $suffix_k(p) = p_k p_{k+1} \dots p_{l-1}$, onde $0 \leq k \leq l$. Por exemplo, $prefix\_key_4(00010^*) = prefix\_key_4(00011^*) = (0001)_2 = 1$ e $suffix_4(00010^*) = 0^*$ [28]. Seja $len(p)$ o tamanho de um prefixo de rede. Para todo prefixo de rede p cujo $len(p) \geq k$, seu k-ésimo prefixo (ou chave) é armazenado em uma árvore B^+ e uma Priority Trie é alocada. Lembrando que, em uma árvore B^+, os dados são armazenados apenas em nós externos (ou folhas). Na MIHT, os dados consistem de ponteiros para Priority Tries. Se a chave já existir na árvore B^+, o ponteiro para a Priority Trie associada, que foi previamente alocada, é recuperado e usado. Os k-ésimos sufixos de todos os prefixos, juntamente com os respectivos endereços de próximo salto e outras informações de roteamento, são armazenados em Priority Tries. Prefixos de rede cujo tamanho seja menor que k são "sem chave", isto é, eles são armazenados diretamente como sufixos em uma Priority Trie especial, denominada $PT[-1]$. Todos os sufixos armazenados em uma Priority Trie em particular compartilham o mesmo prefixo. A raiz da MIHT tem dois ponteiros: um para a árvore B^+ e outro para a $PT[-1]$.

Para buscar um endereço de destino DA, o algoritmo extrai sua chave aplicando $prefix\_key_k(DA)$ e consulta a árvore B^+ de maneira *top-down*, começando na raiz. Uma travessia da árvore é realizada partindo-se da raiz até atingir um nó-folha, utilizando-se o valor da chave para a execução de buscas binárias em cada nó visitado. Quando a chave é encontrada em um dado nó-folha v, o algoritmo busca por $suffix_k(DA)$ na Priority Trie correspondente, que é acessada através do ponteiro para Priority Trie armazenado junto à chave em v. Se ocorrer o casamento de algum prefixo com $suffix_k(DA)$ em alguma Priority Trie, então o CMP e o endereço de próximo salto foram determinados com sucesso, e a busca é encerrada. Se $prefix\_key_k(DA)$ não for encontrado na árvore B^+ ou se ele for encontrado na árvore B^+, mas $suffix_k(DA)$ não for encontrado na Priority Trie associada, o algoritmo busca DA na $PT[-1]$. O Algoritmo 4 descreve o processo de consulta na MIHT. O algoritmo recebe como entrada o endereço de destino DA a ser consultado; um ponteiro para a raiz da árvore B^+; um ponteiro para a Priority Trie $PT[-1]$, que armazena os prefixos menores que k; e a rota padrão. A variável v marca o nó corrente da árvore B^+ durante sua travessia e é inicializada com o valor da raiz na linha 1. O laço da linha 2 é executado enquanto v não for um nó-folha ou, em outras palavras, enquanto v for um i-node interno. Neste laço, o algoritmo seleciona o próximo nó a ser visitado comparando o valor da chave do endereço de entrada ($prefix\_key_k(DA)$) com as $k-1$ chaves armazenadas no nó v. Para isso, recupera-se o filho de v associado ao

Algoritmo 4: CONSULTA_MIHT($DA, B^+, PT[-1], rota\_padrão$): algoritmo de consulta IP em uma (k,m)-MIHT. Adaptado de [28].

Entrada: DA {Endereço de destino}, B^+ {Raiz da árvore B^+}, $PT[-1]$ {Priority Trie que armazena prefixos menores que k} e $rota\_padrão$ {Endereço de próximo salto padrão}.
Saída: O endereço de próximo salto.

```
1  próximo_salto := rota_padrão
2  v := B+
3  enquanto ¬folha(v) faça
4  |  encontre i tal que I_i(v) ≤ prefix_key_k(DA) < I_{i+1}(v)
5  |  v := filho_i(v)
6  fim
7  encontre i tal que I_i(v) ≤ prefix_key_k(DA) < I_{i+1}(v)
8  se prefix_key_k(DA) = I_i(v) então
   |  // A chave prefix_key_k(DA) foi encontrada em um i-node externo.
9  |  próximo_salto := CONSULTA_PRIORITY(suffix_k(DA), filho_i(v), rota_padrão)
10 |  se próximo_salto ≠ rota_padrão então
   |  |  // CMP foi encontrado.
11 |  |  return próximo_salto
12 |  fim
13 fim
14 retorna CONSULTA_PRIORITY(DA, PT[-1], rota_padrão)
```

índice i que satisfaz a condição de que a i-ésima chave armazenada no nó v ($I_i(v)$) deve ser menor ou igual a $prefix\_key_k(DA)$ e que I_{i+1} deve ser maior que $prefix\_key_k(DA)$. Como as chaves estão ordenadas de forma crescente, é possível realizar nesse passo uma busca binária. Na linha 6 o algoritmo procura pela chave no i-node externo. Se a chave existir, então consulta-se a Priority Trie correspondente passando-se como argumento $suffix\_key_k(DA)$. Se o resultado dessa consulta for diferente da rota padrão, então o CMP foi encontrado. Caso contrário, ou caso a chave não exista no i-node externo, o algoritmo procura DA em $PT[-1]$.

Operações de consulta IP são realizadas associando-se cada prefixo de rede a uma chave de tamanho k na MIHT. Assim, o problema de encontrar o CMP é transformado em um problema que consiste, inicialmente, em encontrar um índice em uma árvore B^+. A partir dessa transformação, a altura da MIHT é menor que W (o tamanho dos endereços de entrada), o que acelera a velocidade de consulta. Existem dois parâmetros que afetam o desempenho da MIHT: o tamanho k das chaves e a ordem m da árvore B^+. Uma (k, m)-MIHT é uma estrutura de dados que combina uma árvore B^+ de ordem m e Priority Tries, e que contém dois tipos de nós: *index node* (i-node) e *data node* (d-node). Um i-node pode ser *interno* ou *externo*. Um i-node interno é um nó em que cada filho é também um i-node. Um i-node externo é um nó-folha na árvore B^+ que armazena chaves e ponteiros para d-nodes. Por fim, um d-node é uma Priority Trie que armazena os endereços de próximo salto. A Figura 3.4 apresenta um exemplo de uma (4,4)-MIHT. Nesse caso, o tamanho das

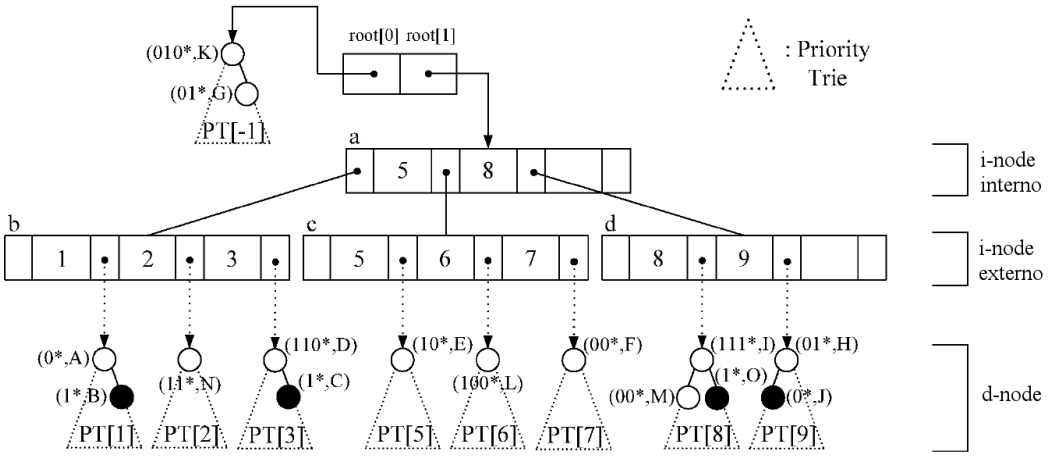

Figura 3.4: Exemplo de uma (4,4)-MIHT. As Priority Tries foram projetadas para armazenar prefixos mais longos em níveis mais altos da árvore, possibilitando, em alguns casos, a determinação do CMP sem que seja necessário realizar uma travessia completa da estrutura até um nó-folha. Para tanto, os nós são classificados em dois tipos: *nó-prioridade* (cor branca) e *nó-comum* (cor preta). Um nó-prioridade é caracterizado por armazenar um prefixo de tamanho maior do que o nível em que o nó se encontra na estrutura, por exemplo, na $PT[8]$, o nó-raiz (nível 0) é um nó-prioridade, pois ele armazena o prefixo $(111^*, I)$, que possui tamanho 3. Contrariamente, em um nó-comum, o tamanho do prefixo armazenado é exatamente igual ao nível do nó. Como consequência, quando ocorre um casamento em um nó-prioridade, o CMP é imediatamente determinado. Adaptada de [28].

chaves k é igual a 4. Isso implica que prefixos menores que 4 bits — como o 010^* (3 bits) e o 01^* (2 bits), que mapeiam os endereços de entrada para as interfaces de saída K e G, respectivamente — são armazenados na $PT[-1]$. Os demais prefixos, de tamanho maior ou igual a 4 bits, possuem os 4 primeiros bits (correspondentes à aplicação da função $prefix\_key_4$) armazenados como um inteiro (chave) na árvore B^+ e os bits restantes (correspondentes à aplicação da função $suffix\_key_4$) na Priority Trie correspondente $(PT[1], PT[2], PT[3] \ldots)$. Como $m = 4$, todos os nós da árvore B^+ armazenam $m - 1 = 3$ chaves e $m = 4$ ponteiros para nós-filho. Mostrou-se em [28] que o melhor desempenho de consulta é obtido configurando-se $k = m = 16$ para o IPv4 e $k = m = 32$ para o IPv6. Portanto, estes valores são usados nas avaliações deste trabalho.

3.5.2 Paralelização e Detalhes de Implementação

Otimizações e Ajuste Fino do Algoritmo

Este algoritmo já é altamente otimizado para implementação em software. Conforme apresentado em maiores detalhes em [28], diversas soluções para tabelas de encaminhamento baseadas em *tries* foram propostas, mas todas apresentam pelo menos uma das

seguintes desvantagens: (i) é necessário espaço extra de memória porque alguns nós dessas estruturas não contêm informações de roteamento (nós supérfluos); (ii) a altura da árvore é limitada por $O(W)$, onde W é o tamanho em bits do maior prefixo possível ($W = 32$ para o IPv4 e $W = 64$ para o IPv6), fazendo com que as operações de consulta tenham que percorrer até W nós no pior caso; (iii) em uma consulta, no caminho de busca partindo da raiz, existem muitos prefixos que não casam com o endereço de destino; (iv) as tabelas de encaminhamento são estáticas e não podem ser atualizadas em tempo hábil. Para reduzir o tempo necessário para executar as operações sobre as tabelas de encaminhamento (inserção, remoção e atualização), na MITH cada prefixo é associado a uma chave, transformando o problema de encontrar o CMP em um problema que consiste, inicialmente, em encontrar um índice em uma árvore B^+. Com essa transformação, a altura da MIHT é menor que W, acelerando as operações de consulta e atualização. Além disso, os bits supérfluos de cada prefixo não precisam ser salvos na MIHT, reduzindo ainda mais os requisitos de memória. Por fim, na maioria das vezes, em operações de atualização, não ocorre divisão ou mesclagem de nós na árvore B^+, o que significa que operações de atualização envolvem apenas as Priority Tries. Assim, a implementação da MIHT usada neste trabalho incorporou as otimizações originalmente propostas pelos autores, bem como integrou algumas otimizações visando a sua execução eficiente no Intel Phi. Por exemplo, para otimizar a vazão de acesso à memória, os nós da árvore B^+ foram alinhados em endereços de memória múltiplos de 64 bytes. Essa otimização é útil porque ela aproveita o fato de que cada nó armazena 16 inteiros de 32 bits, cujo espaço em memória corresponde exatamente ao tamanho da palavra do coprocessador. Isso permite que um nó seja completamente lido em um acesso à memória, acelerando a travessia da estrutura. Conforme sugerido no algoritmo original, as 16 chaves em um nó são ordenadas e realiza-se uma busca binária em cada nó visitado para encontrar rapidamente o próximo i-node ou d-node.

Paralelismo a Nível de Tarefas (PNT)

A MIHT é construída a partir de estruturas de dados baseadas em árvores e o processo de consulta consiste, basicamente, em percorrer essas estruturas seguindo os ponteiros e realizando os devidos acessos à memória. Assim, de forma similar ao que foi desenvolvido na abordagem com filtros de Bloom, paralelizou-se as execuções das consultas na MIHT entre diferentes endereços IP de entrada. A implementação da paralelização nesse nível também empregou a interface de programação OpenMP, com cada *thread* computacional sendo responsável por processar de forma independente um ou mais endereços de entrada.

Vetorização

A MIHT é construída a partir de árvores, que são estruturas de dados que possuem natureza irregular e tornam o uso de operações SIMD muito desafiador. Analisando-se o algoritmo foi possível perceber uma oportunidade para a aplicação de instruções vetoriais SIMD durante a busca binária realizada em cada nó da árvore B^+. Contudo, uma vez que essa busca é muito rápida por causa do número pequeno de elementos em um nó e pelo fato de que os elementos já estão ordenados, o desempenho da MIHT não pôde ser melhorado com o uso de instruções vetoriais.

Capítulo 4

Resultados

Este capítulo apresenta uma avaliação de desempenho do algoritmo otimizado baseado em filtros de Bloom (denominado Bloomfwd) tanto para o IPv4 quanto para o IPv6 sob diferentes circunstâncias e configurações. As avaliações incluem a análise do impacto dos parâmetros do algoritmo e da eficácia das otimizações implementadas no que se refere ao desempenho. Especificamente, são avaliados os efeitos da escolha das funções de *hash*, da escolha da taxa de falso positivo e do uso da ECP/ECPPD. Posteriormente, apresenta-se uma análise comparativa de desempenho do Bloomfwd com uma implementação do algoritmo otimizado original (Baseline) e com a MIHT usando-se múltiplas bases de dados de prefixos e de endereços. Essa análise contempla a avaliação de desempenho dos algoritmos no Intel Phi 7120P (KNC) e no Intel Phi 7250 (KNL). Por fim, são apresentados os resultados da execução cooperativa do Bloomfwd, que consiste em empregar a CPU e o Intel Phi em conjunto para processar os endereços de entrada.

4.1 Configuração Experimental e Bases de Dados

Os experimentos executados com o coprocessador Intel Phi 7120P foram conduzidos em uma máquina equipada com duas CPUs Intel® Xeon E5-2640 v3 (um total de 16 núcleos físicos de processamento em *two-way hyperthreading*), 64 GB de memória principal e sistema operacional CentOS 7. Por outro lado, os experimentos com o Intel Phi 7250 foram executados em um nó do supercomputador de Stampede [47]. Os códigos-fonte de todos os algoritmos foram desenvolvidos em C11 e compilados com o Intel® C Compiler 16.0.3 tanto para a CPU quanto para o Intel Phi usando o nível de otimização -O3. Os dados dos resultados foram calculados como a média aritmética de três execuções e o maior desvio padrão obtido em todos os experimentos foi 3.38%.

Foram usadas 7 bases de dados de prefixos reais para o IPv4, cujas características são sintetizadas na Tabela 4.1. As bases de dados AS65000 e SYDNEY foram obtidas de [3]

Tabela 4.1: Características das bases de dados de prefixos IPv4 usadas.

Base de dados	Localização	Original			Após a ECP		
		≤ 20	$21-24$	$25-32$	$=20$	$=24$	$=32$
AS65000	-	104,283	516,699	1625	1,048,576	971,555	113,397
SYDNEY	Sydney	102,696	553,811	10,862	1,048,576	1,037,247	67,397
DE-CIX	Frankfurt	102,984	535,074	9287	1,048,576	1,007,513	209,488
LINX	London	100,331	519,503	354	1,048,576	982,940	19,863
MSK-IX	Moscow	102,555	528,728	9529	1,048,576	1,004,073	203,360
NYIIX	New York	102,085	528,455	3637	1,048,576	1,000,128	151,391
PTTMetro-SP	Sao Paulo	103,733	544,703	4095	1,048,576	1,024,899	147,201

e [43], respectivamente. As demais bases de dados foram obtidas de [42]. A Tabela 4.1 apresenta a quantidade de endereços em cada base de dados e o número total de prefixos antes e depois de se executar a ECP para agrupá-los em conjuntos de prefixos de tamanho igual a 20, 24 e 32 bits.

Tabela 4.2: Resultado da aplicação da ECPPD na base de dados de prefixos AS65000-V6.

Base Resultante	Número desejado de tamanhos distintos	Níveis de expansão sugeridos pelo algoritmo	Número estimado de prefixos	Número real de prefixos
ECP8	8	$\{24, 29, 33, 38, 40, 44, 48, 64\}$	61,208	60,261
ECP7	7	$\{29, 33, 38, 40, 44, 48, 64\}$	77,700	76,638
ECP6	6	$\{29, 33, 38, 44, 48, 64\}$	104,625	103,094
ECP5	5	$\{33, 38, 44, 48, 64\}$	385,200	380,217
ECP4	4	$\{33, 44, 48, 64\}$	1,185,300	1,166,135
ECP3	3	$\{44, 48, 64\}$	650,417,961	—

Para o IPv6, utilizou-se a base de dados AS65000-V6. Como o IPv6 ainda não é muito utilizado (em comparação com o IPv4), as bases de dados disponíveis possuem um número pequeno de prefixos. A AS65000-V6 tem 31.645 prefixos distribuídos em 34 tamanhos de prefixo distintos. A Tabela 4.2 mostra os efeitos de se aplicar a ECPPD na base de dados AS65000-V6. A expansão com 3 níveis não é possível por causa da grande quantidade de memória requerida. Note que, embora o algoritmo ignore o problema da captura de prefixos (introduzido na Seção 3.3) ao computar os níveis, suas estimativas são muito próximas dos resultados obtidos.

4.2 Efeito das Funções de Hash e da Taxa de FP

A taxa de falso positivo (TFP) é um aspecto chave para a eficácia de um filtro de Bloom porque ela afeta os requisitos de memória e o número de cálculos de *hash* por consulta. A TFP não afeta os resultados do algoritmo, mas apenas o número de vezes que um filtro de Bloom retorna que determinado valor está presente na tabela *hash* associada quando, na verdade, ele não está. Quando isso ocorre, o algoritmo realiza uma busca desnecessária na tabela *hash*, incorrendo apenas em uma penalidade de desempenho (a corretude do algoritmo não é afetada). Consultar uma tabela *hash* consiste em atravessar uma lista encadeada (nos *buckets*), o que pode se tornar custoso conforme a TFP aumenta. Por outro lado, uma TFP muito baixa requer um número maior de cálculos de *hash* e um alto uso de memória. A TFP de um filtro de Bloom é determinada por três fatores: o número n de entradas programadas no filtro, o tamanho m do filtro e o número k de funções de *hash* usadas para programar/consultar o filtro. Dado um valor n, os valores m e k podem ser derivados conforme detalhado em [4] para atingir a TFP desejada.

O compromisso entre aumentar o número de cálculos de *hash* e os requisitos de memória da aplicação para evitar o custo extra de um falso positivo é complexo. Assim, isso foi avaliado experimentalmente medindo-se o desempenho da aplicação em milhões de consultas por segundo (Mlps) usando-se várias configurações de TFP e funções de *hash*. Funções de *hash* são usadas no algoritmo filtros de Bloom tanto para consultar os filtros de Bloom quanto para endereçar os *buckets* das tabelas *hash* associadas a cada filtro. Dessa forma, é possível usar combinações de funções de *hash* para computar os múltiplos *hash* de um filtro de Bloom ou o *hash* único que endereça uma tabela *hash* em particular. As funções de *hash* usadas foram apresentadas na Seção 3.1.2. A base de dados de prefixos IPv4 usada foi a AS65000 e o conjunto de endereços de entrada consistiu em 2^{26} endereços gerados de forma pseudo-aleatória.

Os resultados apresentados na Figura 4.1 mostram que o desempenho da aplicação é fortemente afetado pela TFP e pela combinação de funções de *hash* escolhidas. Conforme apresentado, o uso da Knuth resultou em um desempenho médio inferior comparado aos outros métodos. A razão para os resultados observados é que esta função de *hash* preserva a divisibilidade, por exemplo, se as chaves inteiras forem todas divisíveis por 2 ou por 4, os *hash* resultantes também serão. Isso gera um problema quando a Knuth é usada em conjunto com filtros de Bloom ou tabelas *hash* em geral, pois muitos valores diferentes acabam endereçando as mesmas posições do vetor de bits e apenas metade ou um quarto dos *buckets* acabam sendo usados, respectivamente.

Por outro lado, a Murmur e a H2 são funções de *hash* mais sofisticadas que proveem melhores distribuições estatísticas, consequentemente todas as configurações usando qualquer combinação delas atingiu taxas de consultas similares. Contudo, o melhor desem-

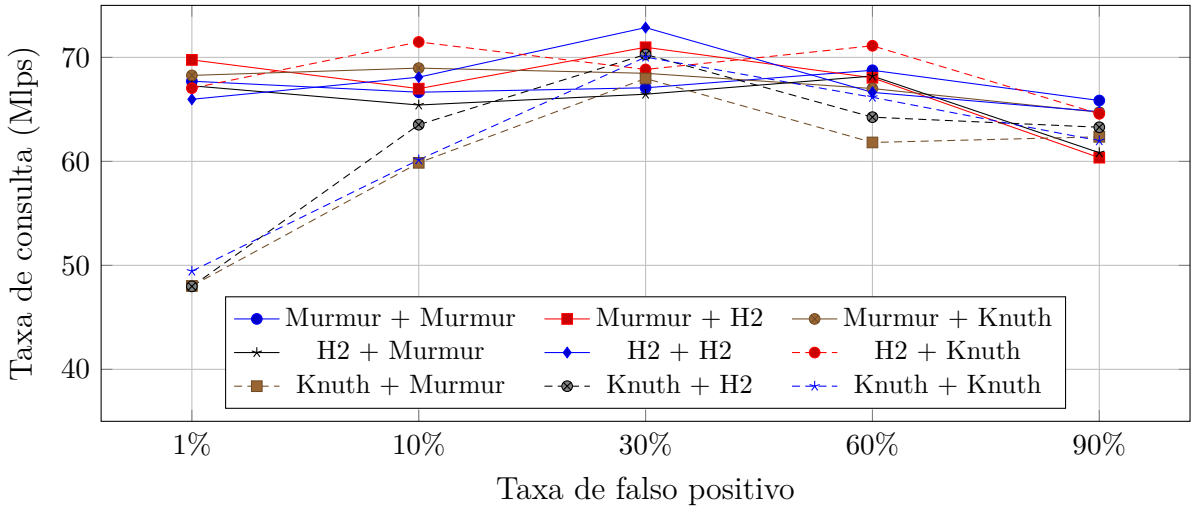

Figura 4.1: Desempenho de múltiplas funções de *hash* e taxas de falso positivo usando 244 *threads* no Intel Phi. A entrada "Murmur + H2" significa que a função de *hash* Murmur foi usada com os filtros de Bloom e a H2 foi usada para endereçar as tabelas *hash*.

penho médio foi alcançado com 30% de TFP, onde os resultados foram menos dispersos para todas as funções de *hash*. Adicionalmente, o melhor desempenho foi obtido quando a função de *hash* H2 foi usada em ambos os estágios do algoritmo. Isso ocorre, em parte, porque nesse caso é possível reutilizar o *hash* calculado para consultar o filtro de Bloom para endereçar também a tabela *hash* associada e, como consequência, menos cálculos de *hash* são executados. Assim, a configuração de 30% de TFP e a função de *hash* H2 em ambos os estágios do algoritmo é usada nos experimentos das próximas seções.

4.3 Escalabilidade: Filtros de Bloom e MIHT

Esta seção avalia os ganhos de desempenho das abordagens baseadas em filtros de Bloom e da MIHT conforme o número de núcleos computacionais usados no Intel Phi 7120P e na CPU aumentam. Compara-se o Bloomfwd com a MIHT e com uma implementação do algoritmo original denominada Baseline. A diferença entre o Bloomfwd e a Baseline está nos cálculos de *hash*, isto é, a Baseline usa a função padrão *rand()* do C sem vetorização [9, 28]. A implementação da MIHT para o IPv4 utilizada nas avaliações foi otimizada de acordo com o proposto pelo autor do algoritmo para a (16,16)-MIHT.

As taxas de consulta (em escala logarítmica) e *speedups* para ambos os algoritmos e processadores são apresentados na Figura 4.2. Conforme mostrado, o desempenho da MIHT ($\approx$ 0,46 Mlps) é melhor que o da Baseline ($\approx$ 0,31 Mlps) para a execução sequencial no Intel Phi. Contudo, conforme o número de *threads* computacionais aumenta, a diferença de desempenho reduz rapidamente devido a melhor escalabilidade da aborda-

gem baseada em filtros de Bloom. Por exemplo, o *speedup* máximo obtido pela Baseline em relação a sua versão sequencial é aproxidamente 61×, enquanto a MIHT atinge um *speedup* máximo de apenas 40× em relação a sua versão sequencial. Por outro lado, o Bloomfwd é o algoritmo mais rápido com um único núcleo e ainda atinge uma melhor escalabilidade no Intel Phi (116×). Além disso, ele é pelo menos 3.7× mais rápido do que os outros algoritmos. As diferenças entre as taxas de consulta do Bloomfwd e da Baseline destacam a importância do uso de vetorização e da escolha das funções de *hash* para o desempenho.

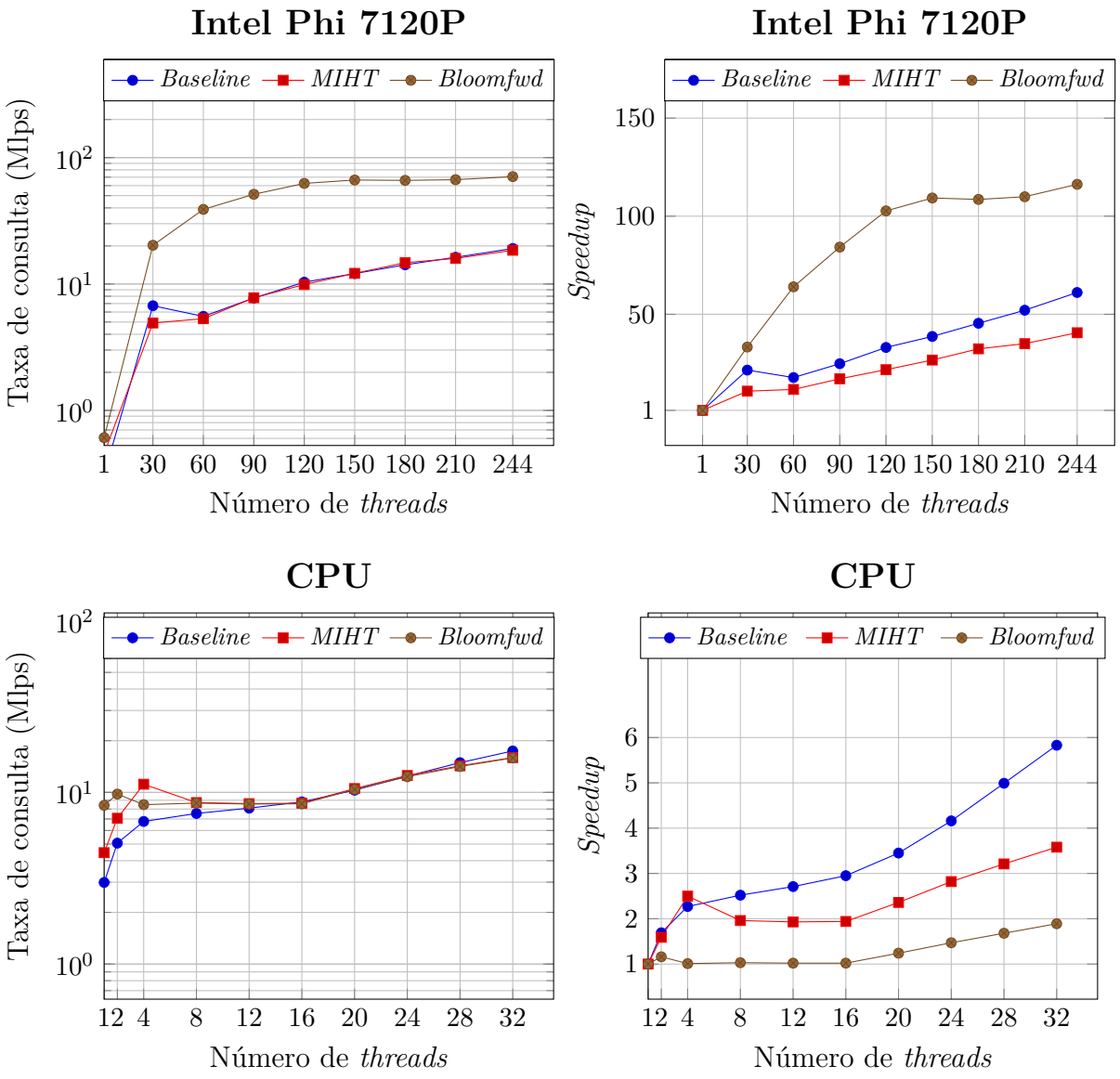

Figura 4.2: Taxa de consulta e escalabilidade dos algoritmos para o IPv4 no Intel Phi 7120P e na CPU usando a base de dados de prefixos AS65000.

A análise dos resultados da CPU mostra que todos os algoritmos atingiram, em escala, taxas de consulta muito similares em configurações *multithread*, apesar de atingirem

speedups diferentes. Os algoritmos obtiveram desempenhos similares na CPU porque a largura de banda de memória desse processador é muito menor do que a do Intel Phi, o que limita a escalabilidade das soluções. Como o Bloomfwd é o algoritmo sequencial mais rápido, ele é o primeiro a alcançar os limites de largura de banda de memória quando o número de núcleos usados aumenta. Observações similares foram feitas em outros trabalhos [48, 13, 49], que avaliaram o desempenho do Intel Phi em comparação com outros processadores.

4.4 Impacto dos Dados de Entrada no Desempenho

O experimento apresentado nesta seção visa avaliar o impacto que conjuntos de endereços de entrada com diferentes características têm sobre o algoritmo. Para investigar os efeitos dos endereços de entrada no desempenho, realizou-se consultas sobre a base de dados de prefixos AS65000 usando-se 2^{26} endereços IPv4 gerados de forma pseudo-aleatória e variou-se a *taxa de casamento*. A taxa de casamento representa a relação entre o número de endereços que casam com pelo menos um prefixo da base de dados e o número total de endereços de entrada, por exemplo, uma taxa de casamento de 80% implica que 20% dos endereços de entrada não casam com qualquer prefixo da base de dados e que, portanto, eles devem ser encaminhados para a rota padrão.

Durante a geração dos diferentes conjuntos de endereço de entrada, garante-se que todos os endereços gerados tenham a mesma probabilidade de casar com qualquer prefixo armazenado na base de dados. Além disso, endereços IPv4 reservados pelo IETF/IANA são filtrados e descartados. A escolha de se usar uma entrada aleatória foi motivada pelo fato de que esse pode ser considerado o cenário de pior caso em um roteador e porque esse é o método mais comumente empregado na literatura. As taxas de consulta obtidas para o Bloomfwd tanto na CPU quanto no Intel Phi 7120P são mostradas na Figura 4.3.

Conforme apresentado na Figura 4.3, o conjunto de dados de entrada tem pouco impacto no desempenho geral da aplicação. A razão para isso é que o caso em que um endereço casa com algum prefixo na base de dados *não é necessariamente mais rápido* do que o caso em que o endereço termina na rota padrão, e vice-versa. Por exemplo, considere um endereço que não casa com qualquer prefixo da tabela de encaminhamento. Se nenhum falso positivo ocorrer, isto é, os dois filtros de Bloom corretamente respondem que não é necessário consultar suas respectivas tabelas *hash*, a busca rapidamente termina consultando-se o DLA com um acesso adicional à memória. Contudo, se para outro prefixo um falso positivo ocorrer no primeiro, mas não no segundo filtro de Bloom, ou se existe uma grande quantidade de valores armazenados nos *buckets* consultados nas tabelas *hash*, então este caso provavelmente será mais lento do que o primeiro. Dessa forma, a

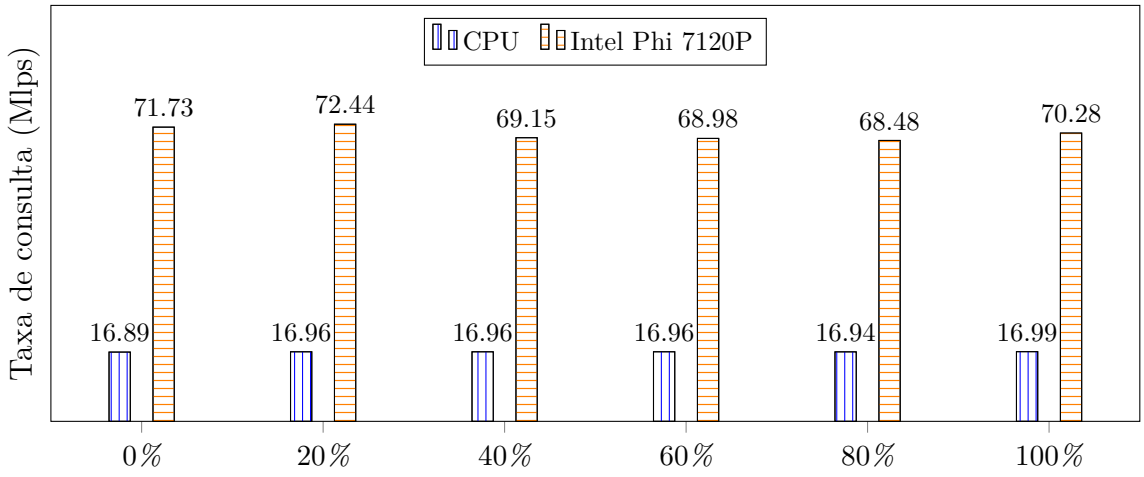

Figura 4.3: Desempenho conforme a taxa de casamento é variada. A entrada 80% significa que esta porcentagem de endereços casa com igual probabilidade pelo menos um prefixo da tabela de encaminhamento, enquanto 20% dos endereços terminam na rota padrão.

velocidade e a boa distribuição estatística das funções de *hash* para controlar a TFP e também minimizar o número de colisões nas tabelas *hash* são um importante aspecto para limitar as variações no desempenho resultantes das características dos conjuntos de dados de entrada.

Alguns trabalhos anteriores também utilizaram endereços coletados de traços IPv4 reais. Então gerou-se um novo conjunto de entrada contendo 2^{26} endereços obtidos do conjunto de dados "CAIDA Anonymized Internet Traces" [6] e mediu-se o desempenho da MIHT e do Bloomfwd para a base de dados de prefixos AS65000. Diferentemente do que ocorre com endereços aleatórios, muitos endereços se repetem e aparecem próximos uns dos outros nessa carga de trabalho. Isso ocorre porque, em um cenário real, é comum que pacotes contendo o mesmo endereço de destino cheguem em rajadas nos roteadores. A MIHT e o Bloomfwd alcançaram taxas de consulta de 19,7 Mlps e 83,76 Mlps, respectivamente, no Intel Phi 7120P. Houve um ganho de desempenho de aproximadamente 16% tanto para a MIHT quanto para o Bloomfwd, que ocorrem por causa do aumento no número de *cache hits* decorrentes da repetição dos endereços de entrada.

4.5 Desempenho de Consultas IPv4 e IPv6

Esta seção avalia o desempenho do Bloomfwd com diferentes bases de dados de prefixos IPv4 e IPv6 no Intel Phi 7120P e no Intel Phi 7250. Primeiramente, discute-se o desempenho para as 6 bases de dados de prefixos IPv4 remanescentes (apresentadas na Tabela 4.1) usando-se um conjunto de endereços de entrada com 2^{26} endereços IPv4 aleatórios. Os resultados, apresentados na Figura 4.4, mostram que os ganhos de desempenho

do Bloomfwd, em relação à MIHT, são de aproximadamente 4× para o 7120P, independentemente da base de dados usada. Adicionalmente, executou-se o Bloomfwd no Intel Phi 7250, e o algoritmo atingiu um *speedup* extra de aproximadamente 2,4×, em relação à execução no 7120P, e uma vazão máxima de 169,65 Mlps. Essa diferença significativa de desempenho entre os dois processadores não poderia ser derivada diretamente de suas diferentes características, apresentadas na Seção 2.2.2. Assim, executou-se o STREAM *benchmark* (uma aplicação intensiva de dados) em ambos os processadores e constatou-se que, na prática, o 7250 atinge uma largura de banda de memória aproximadamente 2,7× maior que o 7120P, o que explica os ganhos do Bloomfwd no 7250.

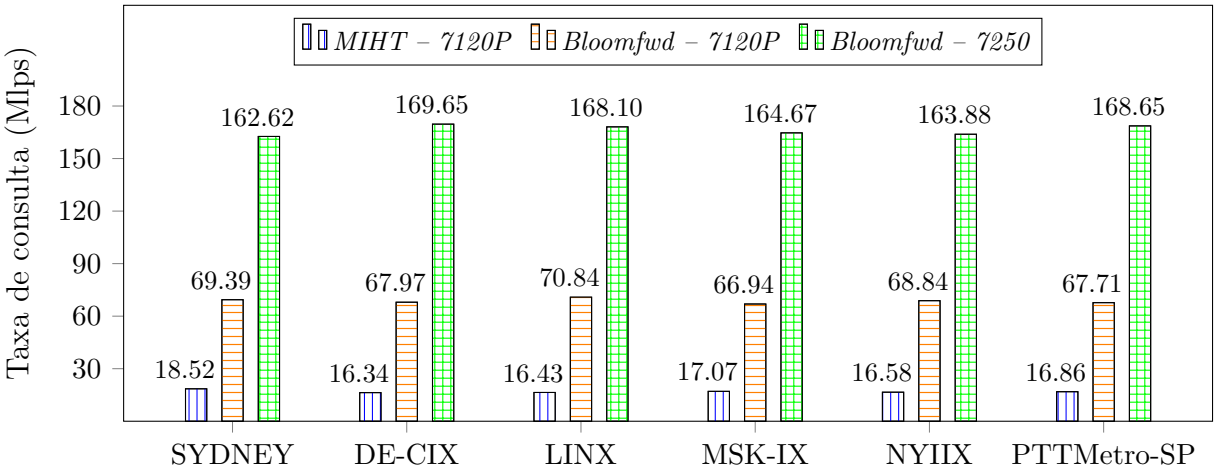

Figura 4.4: Desempenho do Bloomfwd e MIHT para 6 conjuntos de dados de prefixos IPv4 em dois coprocessadores Intel Phi: 7120P e 7250.

Avaliou-se também o desempenho da implementação otimizada do algoritmo baseado em filtros de Bloom para o IPv6 (Bloomfwd-v6) e o impacto de se usar a otimização ECPPD, proposta na Seção 3.3. Este experimento compara as taxas de consulta do Bloomfwd-v6 com a versão correspondente da MIHT para o IPv6 (MIHT-v6). A implementação da MIHT-v6 corresponde ao algoritmo (32,32)-MIHT [28].

A Figura 4.5 mostra os resultados para a base de dados AS65000-V6 com e sem a aplicação da ECPPD para múltiplos níveis de expansão e 2^{26} endereços de entrada aleatórios. Conforme apresentado, o desempenho do Bloomfwd-v6 melhora significativamente com o uso da ECPPD, e a expansão com 7 níveis é aproximadamente 5,9× mais rápida do que a execução sem ECP (AS65000-v6). Essa versão é também 5,3× mais rápida que a MIHT-v6. Contudo, reduzindo-se a quantidade de tamanhos de prefixos distintos para valores menores do que 7, o desempenho do algoritmo piora devido a maior demanda de memória e ao maior custo de busca. O desempenho da MIHT-v6 é similar em todos os casos porque esse algoritmo agrupa as rotas por chaves em Priority Tries (PTs), ao invés de tamanhos de prefixos (como na abordagem baseada em filtros de Bloom) e, por esse

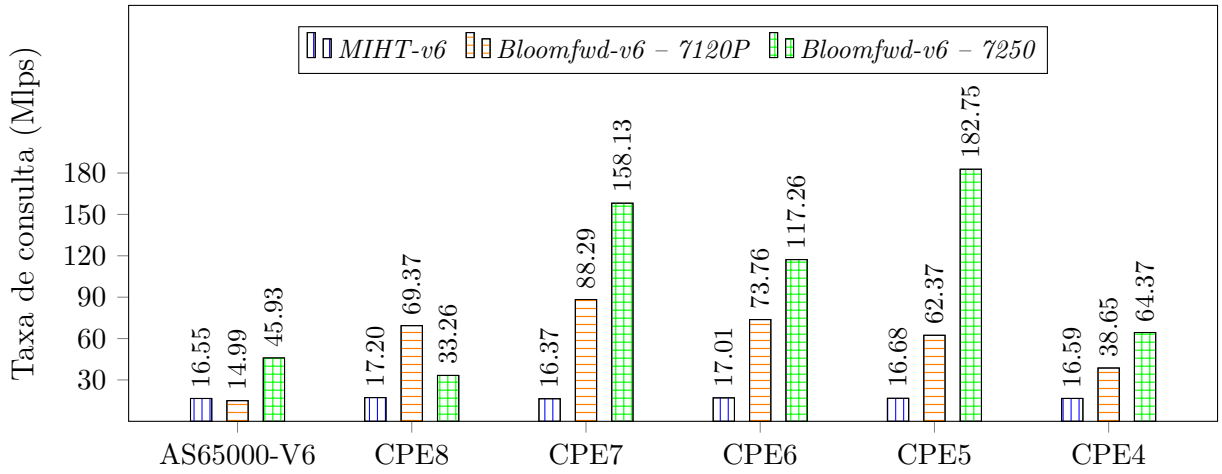

Figura 4.5: Desempenho do Bloomfwd-v6 e MIHT-v6 para a base de dados de prefixos AS65000-V6 e 2^{26} endereços IP aleatórios.

motivo, o número de tamanhos de prefixos distintos na tabela de encaminhamento não afeta diretamente o desempenho da MIHT-v6. O Bloomfwd-v6 no Intel Phi 7250 atingiu uma vazão máxima de 182,75 Mlps e, conforme apresentado, há uma grande variação no desempenho entre os diferentes níveis de expansão. A escolha dos níveis de expansão é complexa e envolve muitos aspectos, tais como capacidades de *caching* e os seus efeitos sobre os demais parâmetros do algoritmo (ex.: número de cálculos de *hash* para manter a taxa de falso positivo), requerendo uma avaliação experimental para uma escolha adequada.

4.6 Execução Cooperativa

Esta seção avalia os ganhos de desempenho decorrentes da implementação cooperativa do Bloomfwd (apresentada na Seção 4.6) usando uma estratégia assíncrona que preenche e processa dois *buffers* de endereços concorrentemente (*double-buffering*). A execução síncrona foi omitida porque a execução assíncrona obteve um desempenho melhor, alcançando um *speedup* de 1,13× em relação à primeira. Para melhor explorar o *double-buffering*, os *buffers* na estratégia assíncrona devem ser dimensionados com tamanhos apropriados. Esses tamanhos foram definidos experimentalmente usando-se como critério de escolha a maximização da vazão do número de consultas IP. A Figura 4.6 mostra o impacto que o tamanho dos *buffers* têm no desempenho da aplicação.

Conclui-se da Figura 4.6 que o melhor desempenho é obtido usando-se um *buffer* de tamanho igual a 4M (*mebipackets*). Contudo, além do tempo de execução, outro aspecto que deve ser levado em consideração ao se escolher o tamanho do *buffer* em um cenário real — que contemple operações de I/O para a recepção e transmissão dos

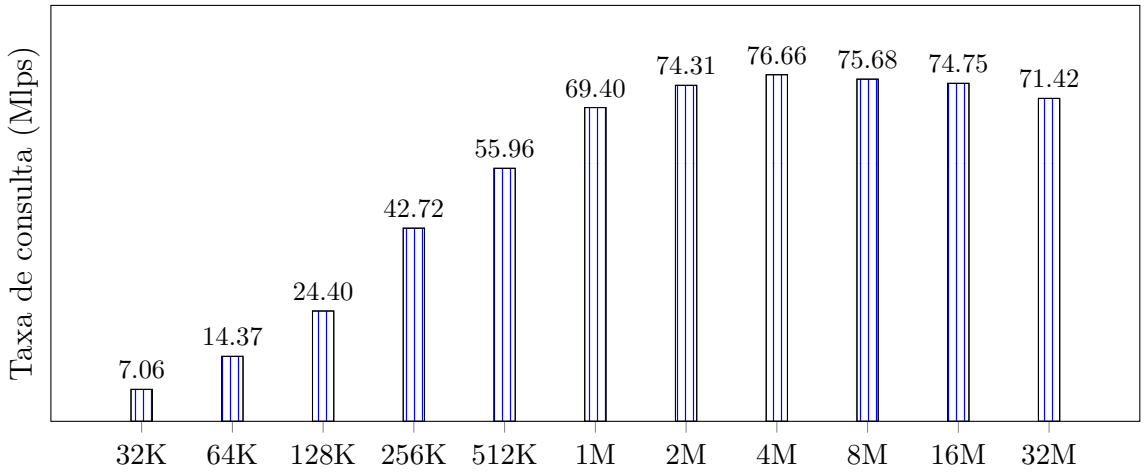

Figura 4.6: Desempenho da versão assíncrona para diferentes tamanhos de *buffer* e uma entrada contendo 2^{30} endereços.

pacotes — é a velocidade dos adaptadores de rede (NICs) instalados na máquina e a latência máxima tolerada para o sistema. Em outras palavras, existe um compromisso entre o tamanho do *buffer* e a latência média desejada do sistema. Como o desempenho com *buffers* de 2M foi muito próximo ao desempenho máximo (obtido com *buffers* de 4M) e, levando em consideração que o tempo requerido para encher o *buffer* no primeiro caso é, aproximadamente, 2× menor, optou-se por utilizar esta configuração no experimento a seguir, que mede o desempenho da estratégia assíncrona usando a CPU e o Intel Phi 7120P para processar 2^{30} endereços IPv4 gerados de forma pseudo-aleatória. Nesta avaliação, a quantidade de trabalho (% de endereços IP) atribuídos para processamento no Intel Phi é variada. Em cada caso, os endereços IP restantes são processados pela CPU. Os experimentos apresentados nas seções anteriores mostram que o Intel Phi 7120P é aproximadamente 4× mais rápido que a CPU. Assim, a estratégia empregada para dividir a carga de trabalho atribui 80% dos endereços IP para o Intel Phi e o restante para a CPU. Os resultados, apresentados na Figura 4.7, confirmam que esta é a configuração que resulta no melhor desempenho, alcançando um *speedup* de 1,18× sobre a execução que usa apenas o Intel Phi. O ganho combinado da execução assíncrona e cooperativa foi de aproximadamente 1,33× em relação à execução que usa apenas o Intel Phi.

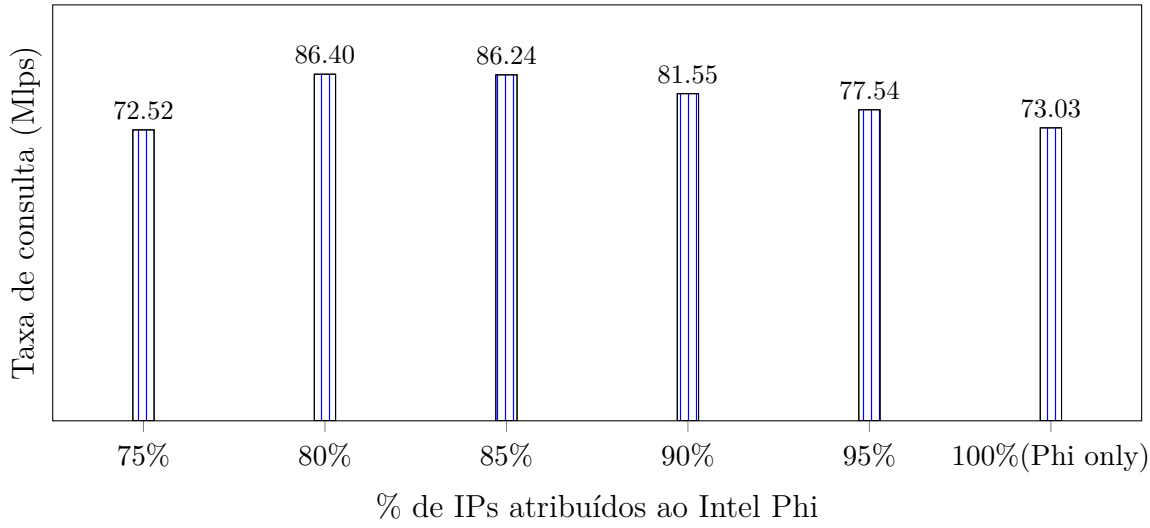

Figura 4.7: Execução cooperativa entre a CPU e o Intel Phi 7120P do Bloomfwd para a base de dados de prefixos AS65000 e 2^{30} endereços aleatórios.

Capítulo 5

Conclusão

Neste trabalho, projetou-se, implementou-se e avaliou-se o desempenho de algoritmos que abordam o problema de consultas IP usando estratégias de execução e estruturas de dados diferentes (abordagem baseada em filtros de Bloom e MIHT). Esses algoritmos foram avalidos em CPUs *multi-core* modernas e no coprocessador *many-core* Intel Phi. A MIHT é conhecida por ser um algoritmo sequencial eficiente [28]. Contudo, ela emprega estruturas de dados irregulares (B^+ e Priority Tries), que tipicamente levam a oportunidades reduzidas para a execução otimizada em sistemas paralelos. O algoritmo baseado em filtros de Bloom, por outro lado, é um algoritmo mais intensivo em computação e que possui uma versão sequencial menos eficiente, porém, ele é constituído a partir de estruturas de dados regulares e de fácil paralelização (filtros de Bloom e tabelas *hash*).

Em função das suas características, o algoritmo baseado em filtros de Bloom oferece mais oportunidades de otimização do que a MIHT, por exemplo, com o uso de instruções SIMD. Além disso, ele é mais escalável com respeito ao número de núcleos computacionais usados. Conforme apresentado na seção de resultados, o desempenho do algoritmo otimizado baseado em filtros de Bloom superou significativamente o desempenho da MIHT no Intel Phi, sendo capaz de computar até 169,6 Mlps (84,6 Gbps para pacotes IPv4 de 64B) e 182,7 Mlps (119,9 Gbps para pacotes IPv6 de 84B). As execuções na CPU também atingiram um bom desempenho, mas a baixa largura de banda desse processador limita a escalabilidade dos algoritmos. Mesmo assim, o algoritmo otimizado baseado em filtros de Bloom foi melhor que a MIHT também na execução sequencial na CPU.

Embora a boa escalabilidade da abordagem baseada em filtros de Bloom mostre que ela é uma opção melhor para dispositivos com um número grande de núcleos computacionais, propôs-se e avaliou-se um modelo de execução cooperativa que emprega o coprocessador Intel Phi e todas as CPUs *multi-core* disponíveis no sistema para processar as consultas IP conjuntamente. Otimizando-se as transferências de dados que ocorrem pelo barramento PCIe, atingiu-se um *speedup* de aproximadamente 1,33× em relação às execuções que usam

apenas o Intel Phi. Este trabalho propôs também uma otimização algorítmica que combina técnicas de Programação Dinâmica e a Expansão Controlada de Prefixos (ECPPD) para aprimorar o desempenho de consultas IPv6. Essa abordagem inovadora resultou em ganhos de desempenho de até 5,9× no algoritmo baseado em filtros de Bloom.

Após todas as otimizações implementadas, a avaliação experimental mostrou que o algoritmo sequencial mais eficiente nem sempre é a melhor solução em um ambiente com alto poder de paralelismo. Pelo contrário, a possibilidade de se adaptar um algoritmo para que ele utilize completamente os recursos computacionais do processador em questão pode levar a um maior desempenho. Os resultados obtidos e as recentes melhorias no Intel Phi, tais como maior largura de banda de memória, maior número de núcleos computacionais e a possibilidade de usá-lo como um processador independente (ou *standalone*), motiva o uso das técnicas propostas e torna o Intel Phi uma plataforma atrativa e promissora para a implementação de roteadores em software de alto desempenho. Até onde se sabe, este é o primeiro trabalho a avaliar sistematicamente o Intel Phi usando múltiplos algoritmos e configurações para a execução de consultas IP.

A implementação de ambos os algoritmos para o IPv6 considerou prefixos de tamanho máximo igual a 64 bits. Embora essa convenção vá de encontro ao especificado na RFC 7608 [41], ela foi adotada para manter a consistência com trabalhos anteriores, que utilizaram esse tamanho nas avaliações de desempenho. Contudo, a adequação da MIHT e do algoritmo baseado em filtros de Bloom para que eles trabalhem com prefixos de até 128 bits é trivial. A primeira mudança consiste em atualizar o tipo de dados que representa um prefixo de rede para um tipo capaz de representar eficientemente os possíveis 128 bits que compõem cada prefixo. Adicionalmente, no algoritmo baseado em filtros de Bloom, pares adicionais de filtros de Bloom e tabelas *hash* devem ser alocados para armazenar os prefixos de 65 bits ou mais. Observe que, em decorrência dessa mudança, os níveis de expansão escolhidos pelo algoritmo ECPPD será diferente, pois o próprio ECPPD deve ser atualizado para começar escolhendo o tamanho de prefixo 128, ao invés de 64 (vide Seção 3.3). No que se refere à MIHT, possivelmente os valores dos parâmetros k e m tenham que ser alterados, ou seja, de acordo com Dharmapurikar et al. [28], os valores que resultam no melhor desempenho médio para o IPv6 são $(k, m) = (32, 32)$, porém eles só consideraram prefixos de até 64 bits [28]. Esses parâmetros afetam o número de índices armazenados em cada nó da árvore B^+, a altura da estrutura de dados e a lógica de armazenamento dos elementos nas múltiplas Priority Tries. Como o trabalho anterior não considerou bases contendo prefixos de até 128 bits, é necessária uma avaliação de desempenho para escolher a melhor configuração de k e m quando prefixos maiores que 64 bits estão presentes.

Diversas plataformas de execução e algoritmos já foram avaliados visando-se a execução eficiente de consultas IP em roteadores em software. Apesar disso, novos dispositivos de hardware e novas abordagens são propostas frequentemente para o problema. Por exemplo, uma classe de algoritmos que tem se destacado neste contexto são aqueles baseados em estruturas de dados comprimidas [39, 2]. Assim, como trabalhos futuros destacam-se a investigação de outros algoritmos de consultas IP buscando-se oportunidades de paralelização e vetorização no Intel Phi. Adicionalmente, planeja-se adaptar o algoritmo otimizado baseado em filtros de Bloom para GPUs modernas e comparar o seu desempenho com o obtido no Intel Phi. Por fim, propõe-se a integração deste algoritmo em um roteador em software existente, como o Open vSwitch [34].

Referências

[1] Austin Appleby. MurmurHash3 Hash Function, 2011. `https://code.google.com/p/smhasher/wiki/MurmurHash3`. 24

[2] Hirochika Asai e Yasuhiro Ohara. Poptrie: A Compressed Trie with Population Count for Fast and Scalable Software IP Routing Table Lookup. In *ACM SIGCOMM Computer Communication Review*, volume 45, pages 57–70. ACM, 2015. 13, 51

[3] BGP Potaroo, 2016. `http://bgp.potaroo.net/`. 27, 38

[4] Burton H Bloom. Space/Time Trade-offs in Hash Coding with Allowable Errors. *Communications of the ACM*, 13(7):422–426, 1970. 2, 18, 40

[5] Andrei Broder e Michael Mitzenmacher. Using Multiple Hash Functions to Improve IP Lookups. In *INFOCOM 2001. Twentieth Annual Joint Conference of the IEEE Computer and Communications Societies.*, volume 3, pages 1454–1463, 2001. 13

[6] The CAIDA UCSD Anonymized Internet Traces 2016, 2016. `http://www.caida.org/data/passive/passive_2016_dataset.xml`. 44

[7] Yeim-Kuan Chang. Fast Binary and Multiway Prefix Searches for Packet Forwarding. *Computer Networks*, 51(3):588–605, 2007. 13

[8] Chu, Hung-Mao and Li, Tsung-Hsien and Wang, Pi-Chung. IP Address Lookup by using GPU. 4(187–198), 2016. 14

[9] Sarang Dharmapurikar, Praveen Krishnamurthy, e David E Taylor. Longest Prefix Matching Using Bloom Filters. *IEEE/ACM Transactions on Networking*, 14(2):397–409, 2006. x, 2, 13, 14, 17, 18, 19, 20, 21, 23, 24, 26, 27, 28, 41

[10] Mihai Dobrescu, Norbert Egi, Katerina Argyraki, Byung-Gon Chun, Kevin Fall, Gianluca Iannaccone, Allan Knies, Maziar Manesh, e Sylvia Ratnasamy. RouteBricks: Exploiting Parallelism to Scale Software Routers. In *Proc. of the ACM SIGOPS 22nd Symposium on Operating Systems Principles*, pages 15–28, 2009. 14, 15, 16

[11] Li Fan, Pei Cao, Jussara Almeida, e Andrei Z Broder. Summary Cache: A Scalable Wide-Area Web Cache Sharing Protocol. *IEEE/ACM Transactions on Networking (TON)*, 8(3):281–293, 2000. 17, 18

[12] Michael J Flynn. Very High-Speed Computing Systems. *Proceedings of the IEEE*, 54(12):1901–1909, 1966. 10

[13] Jeremias M Gomes, George Teodoro, Alba de Melo, Jun Kong, Tahsin Kurc, e Joel H Saltz. Efficient Irregular Wavefront Propagation Algorithms on Intel(R) Xeon Phi(tm). In *2015 27th International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD)*, pages 25–32, 2015. 43

[14] Pankaj Gupta, Steven Lin, e Nick McKeown. Routing Lookups in Hardware at Memory Access Speeds. In *INFOCOM'98. Seventeenth Annual Joint Conference of the IEEE Computer and Communications Societies.*, volume 3, pages 1240–1247, 1998. 27

[15] Sangjin Han, Keon Jang, KyoungSoo Park, e Sue Moon. PacketShader: a GPU-Accelerated Software Router. *ACM SIGCOMM Computer Communication Review*, 41(4):195–206, 2011. 1, 2, 3, 12, 14, 15, 16

[16] Sun-Yuan Hsieh e Ying-Chi Yang. A Classified Multisuffix Trie for IP Lookup and Update. *IEEE Transactions on Computers*, 61(5):726–731, 2012. 13

[17] Intel. Intel Intrinsics Guide. `https://software.intel.com/sites/landingpage/IntrinsicsGuide/`, 2015. 25

[18] Intel Corporation. Intel® Xeon Phi® Coprocessor System Software Developers Guide. http://software.intel.com/en-us/mic-developer, November 2012. 16

[19] Anuj Kalia, Dong Zhou, Michael Kaminsky, e David G Andersen. Raising the Bar for Using GPUs in Software Packet Processing. In *NSDI*, pages 409–423, 2015. 2

[20] Adam Kirsch e Michael Mitzenmacher. Less Hashing, Same Performance: Building a Better Bloom Filter. *Random Structures & Algorithms*, 33(2):187–218, 2008. 21

[21] Donald Ervin Knuth. *The Art of Computer Programming: Sorting and Searching*, volume 3. Pearson Education, 1998. 24

[22] Eddie Kohler, Robert Morris, Benjie Chen, John Jannotti, e M Frans Kaashoek. The Click Modular Router. *ACM Transactions on Computer Systems (TOCS)*, 18(3):263–297, 2000. 16

[23] James F Kurose e Keith W Ross. *Computer Networking: A Top-Down Approach: 6th Edition*. Pearson Higher Ed, 2013. 8, 9

[24] Yanbiao Li, Dafang Zhang, Alex X Liu, e Jintao Zheng. GAMT: A Fast and Scalable IP Lookup Engine for GPU-based Software Routers. In *Proceedings of the ninth ACM/IEEE symposium on Architectures for networking and communications systems*, pages 1–12. IEEE Press, 2013. 14

[25] Hyesook Lim e Yeojin Jung. A Parallel Multiple Hashing Architecture for IP Address Lookup. In *High Performance Switching and Routing, 2004. HPSR. 2004 Workshop on*, pages 91–95, 2004. 14

[26] Hyesook Lim, Kyuhee Lim, Nara Lee, e Kyong-Hye Park. On Adding Bloom Filters to Longest Prefix Matching Algorithms. *IEEE Transactions on Computers*, 63(2):411–423, 2014. 13, 24

[27] Hyesook Lim, Changhoon Yim, e Earl E Swartzlander Jr. Priority Tries for IP Address Lookup. *IEEE Transactions on Computers*, 59(6):784–794, 2010. 13, 31, 32

[28] Chia-Hung Lin, Chia-Yin Hsu, e Sun-Yuan Hsieh. A Multi-Index Hybrid Trie for Lookup and Updates. *IEEE Transactions on Parallel and Distributed Systems*, 25(10):2486–2498, 2014. x, 2, 13, 17, 31, 33, 34, 35, 41, 45, 49, 50

[29] Alexandre Lucchesi, André Drummond, e George Teodoro. Parallel and Efficient IP Lookup using Bloom Filters on Intel® Xeon Phi™. In *Anais do XXXV Simpósio de Redes de Computadores e Sistemas Distribuídos — SBRC 2017*, pages 229–242, 2017. 4, 14

[30] Deepankar Medhi. *Network Routing: Algorithms, Protocols, and Architectures*. Morgan Kaufmann, 2007. 7

[31] Shuai Mu, Xinya Zhang, Nairen Zhang, Jiaxin Lu, Yangdong Steve Deng, e Shu Zhang. IP Routing Processing with Graphic Processors. In *Proceedings of the Conference on Design, Automation and Test in Europe*, pages 93–98. European Design and Automation Association, 2010. 14

[32] Thomas Mueller. H2 Database Engine, 2006. `http://h2database.com`. 24, 25

[33] Sheng Ni, Rentong Guo, Xiaofei Liao, e Hai Jin. Parallel Bloom Filter on Xeon Phi Many-Core Processors. In *International Conference on Algorithms and Architectures for Parallel Processing*, pages 388–405. Springer, 2015. 14, 24

[34] Open vSwitch: An Open Virtual Switch, 2017. `http://openvswitch.org/`. 16, 51

[35] OpenMP API for Parallel Programming, Version 4.0, 2016. `http://openmp.org/wp/`. 24

[36] Peter Pacheco. *An Introduction to Parallel Programming*. Elsevier, 2011. 1, 10, 11

[37] David A Patterson e John L Hennessy. *Computer Organization and Design: The Hardware/Software Interface*. Newnes, 2013. 1

[38] Larry L Peterson e Bruce S Davie. *Computer Networks: a Systems Approach*. Elsevier, 2007. 9

[39] Gábor Rétvári, János Tapolcai, Attila Kőrösi, András Majdán, e Zalán Heszberger. Compressing IP Forwarding Tables: Towards Entropy Bounds and Beyond. In *ACM SIGCOMM Computer Communication Review*, volume 43, pages 111–122. ACM, 2013. 13, 51

[40] RFC 5952, A Recommendation for IPv6 Address Text Representation. `https://tools.ietf.org/html/rfc5952`. 9

[41] RFC 7608, IPv6 Prefix Length Recommendation for Forwarding. `https://tools.ietf.org/html/rfc7608`. 9, 50

[42] RIPE Network Coordination Centre, 2016. `http://data.ris.ripe.net/`. 39

[43] University of Oregon Route Views Project, 2017. `http://www.routeviews.org/`. 39

[44] Miguel Ruiz-Sanchez, Ernst W Biersack, Walid Dabbous, et al. Survey and Taxonomy of IP Address Lookup Algorithms. *IEEE Network*, 15(2):8–23, 2001. 13

[45] Sartaj Sahni e Haibin Lu. Dynamic Tree Bitmap for IP Lookup and Update. In *Networking, 2007. ICN'07. Sixth International Conference on*, pages 79–79, 2007. 13

[46] Venkatachary Srinivasan e George Varghese. Fast Address Lookups using Controlled Prefix Expansion. *ACM Transactions on Computer Systems (TOCS)*, 17(1):1–40, 1999. 4, 17, 26

[47] Stampede User Guide, 2017. `https://portal.tacc.utexas.edu/user-guides/stampede#stampede-knl-cluster`. 38

[48] George Teodoro, Tahsin Kurc, Guilherme Andrade, Jun Kong, Renato Ferreira, e Joel Saltz. Application Performance Analysis and Efficient Execution on Systems with Multi-core CPUs, GPUs and MICs: a Case Study with Microscopy Image Analysis. *The International Journal of High Performance Computing Applications*, 31(1):32–51, 2017. 31, 43

[49] George Teodoro, Tahsin Kurc, Jun Kong, Lee Cooper, e Joel Saltz. Comparative Performance Analysis of Intel(R) Xeon Phi(tm), GPU, and CPU: a Case Study from Microscopy Image Analysis. In *2014 IEEE 28th International Parallel and Distributed Processing Symposium*, pages 1063–1072, 2014. 43

[50] George Teodoro, Tony Pan, Tahsin M Kurc, Jun Kong, Lee AD Cooper, Norbert Podhorszki, Scott Klasky, e Joel H Saltz. High-Throughput Analysis of Large Microscopy Image Datasets on CPU-GPU Cluster Platforms. In *2013 IEEE 27th International Symposium on Parallel and Distributed Processing (IPDPS)*, pages 103–114, 2013. 31

[51] George Teodoro, Rafael Sachetto, Olcay Sertel, Metin N Gurcan, Wagner Meira, Umit Catalyurek, e Renato Ferreira. Coordinating the Use of GPU and CPU for Improving Performance of Compute Intensive Applications. In *IEEE International Conference on Cluster Computing and Workshops, 2009. CLUSTER'09.*, pages 1–10, 2009. 31

[52] Xinmin Tian, Hiroshi Saito, Serguei V Preis, Eric N Garcia, Sergey S Kozhukhov, Michael K Masten, Aleksei G Cherkasov, e Nikolay Panchenko. Practical SIMD Vectorization Techniques for Intel® Xeon Phi Coprocessors. In *2013 IEEE 27th International Parallel and Distributed Processing Symposium Workshops & PhD Forum (IPDPSW)*, pages 1149–1158, 2013. 1

[53] S Venkatachary e G Varghese. Faster IP Lookups using Controlled Prefix Expansion. *PERFORMANCE EVALUATION REVIEW*, 26:1–10, 1998. 29

[54] Tong Yang, Gaogang Xie, YanBiao Li, Qiaobin Fu, Alex X Liu, Qi Li, e Laurent Mathy. Guarantee IP Lookup Performance with FIB Explosion. *ACM SIGCOMM Computer Communication Review*, 44(4):39–50, 2015. 13, 14

[55] Changhoon Yim, Bomi Lee, e Hyesook Lim. Efficient Binary Search for IP Address Lookup. *IEEE Communications Letters*, 9(7):652–654, 2005. 13