UNIVERSIDADE FEDERAL DE SANTA CATARINA
DEPARTAMENTO DE INFORMÁTICA E ESTATÍSTICA

Igor da Silva Solecki

REPRESENTAÇÃO OTIMIZADA DE TOPOLOGIAS DE MÁQUINA

Florianópolis
2017

Igor da Silva Solecki

REPRESENTAÇÃO OTIMIZADA DE TOPOLOGIAS DE MÁQUINA

Trabalho de Conclusão de Curso submetido ao Curso de Bacharelado em Ciências da Computação para a obtenção do Grau de Bacharel em Ciências da Computação.
Orientador: Prof. Dr. Laércio Lima Pilla
Universidade Federal de Santa Catarina

Florianópolis
2017

RESUMO

O objetivo deste trabalho é desenvolver novas formas otimizadas de representar e disponibilizar informações sobre topologias de máquina para o uso em aplicações de alto desempenho. O projeto *Hardware Locality* (hwloc), estado da arte em representação de topologias, foi analisado para identificar pontos passíveis de otimizações. Os algoritmos e estruturas desenvolvidos foram testados e comparados com outras abordagens.

Palavras-chave: hierarquia de memória. topologia de máquina. computação de alto desempenho.

ABSTRACT

The goal of this project is to develop new optimized ways of representing and providing informations about machine topologies for the use in high performance applications. The Hardware Locality (hwloc) project, state of the art in topologies representation, was analised in order to identify points that could be optimized. The developed algorithms and structures were tested and compared with other approaches.

Keywords: memory hierarchy. machine topology. high performance computing.

LISTA DE FIGURAS

LISTA DE TABELAS

LISTA DE ABREVIATURAS E SIGLAS

ACMP	Ancestral Comum Mais Próximo
ECL	Laboratório de Computação Embarcada - *Embedded Computing Lab*
hwloc	*Hardware Locality*
MPI	*Message Passing Interface*
NUMA	*Non-Uniform Memory Access*
UMA	*Uniform Memory Access*

SUMÁRIO

1 INTRODUÇÃO

Atualmente, arquiteturas de computadores são construídas de forma hierárquica quanto à memória. Essa hierarquia diz respeito à passagem de dados entre os diferentes níveis, ou seja, quais caminhos existem para que dados sejam comunicados entre pontos dessa hierarquia. O que motiva sua existência é o fato de que diferentes tipos de memória possuem tamanhos, velocidades de acesso e custos distintos, e ela permite que o espaço das memórias maiores esteja disponível sem que se perca a velocidade das menores e mais rápidas (PATTERSON; HENNESSY, 2011). Níveis de memória mais baixos possuem maior capacidade de armazenamento, porém seu tempo de acesso é maior. Parte dessa hierarquia é composta por um ou mais níveis de *cache*, memórias com capacidade reduzida, mas maior velocidade, permitindo que, em um dado momento, um conjunto de dados qualquer possa ser acessado mais rapidamente. Sobre essas hierarquias e fazendo uso delas estão as unidades de processamento, acessando e operando sobre os dados em memória. Quanto mais próximo for o nível de memória em que esses dados estiverem, menor o tempo de acesso. Quando essas unidades precisam se comunicar entre si, elas fazem uso da hierarquia de memória.

A hierarquia pode ser organizada de várias formas, podendo se tornar complexa e de grande profundidade. Uma possibilidade na organização é o compartilhamento de alguns níveis de *cache*, ou seja, duas unidades de processamento ou mais estarão acima de uma mesma *cache* da hierarquia. Isso permite, por exemplo, realizar comunicações com eficiência, pois o tempo entre algum dado ser atualizado e o novo valor ser visto é determinado pelo tempo de acesso à *cache* compartilhada. Uma grande variedade de organizações pode ser encontrada ao se considerar arquiteturas como multicore, em que várias unidades de processamento chamadas de núcleos (*cores*) fazem parte de um mesmo circuito integrado, ou *Non-Uniform Memory Access* (NUMA), onde a rede que interconecta os nós dá origem a mais níveis de hierarquia. Esta rede, por si só, pode ser organizada de várias formas distintas. Essa organização compreendendo hierarquia de memória e unidades de

processamento, na sua totalidade, define uma topologia de máquina.

A necessidade de plataformas para rodar aplicações de alto desempenho tem dado origem às diversas arquiteturas paralelas modernas existentes. À medida que novas tecnologias surgem, se faz necessário adaptar as arquiteturas para permitir o seu uso eficiente, de modo a se atingir alto desempenho nessas plataformas (BARROSO et al., 2017). Suas topologias são as mais variadas, visando atender às necessidades de várias classes de aplicações com características e comportamentos distintos, que vão desde simulações científicas até o processamento de tarefas em sistemas que dão suporte a redes sociais (FRANCESQUINI; GOLDMAN; MÉHAUT, 2013). Diante da crescente complexidade das topologias dessas máquinas, as suas organizações e as demais características dos elementos que compõe a hierarquia de memória são aspectos de muita relevância para o desempenho de aplicações.

Certas combinações de fatores da aplicação e da arquitetura podem resultar na melhoria ou na degradação do desempenho. Tais fatores podem ser, por exemplo, a quantidade de dados manipulados e o tamanho das *caches*, que podem comportar ou não todos os dados simultaneamente, ou os padrões de troca de mensagens entre tarefas e a localização delas, além dos meios existentes para realizar essas comunicações, que podem resultar em maior ou menor eficiência (FATAHALIAN et al., 2006). Um estudo de caso apresentado por Treibig, Hager e Wellein (2010) mostra como uma certa distribuição de tarefas faz o desempenho cair aproximadamente pela metade.

Ainda, em arquiteturas NUMA, nas quais cada parte da memória está mais próxima de alguns nós, de modo que o tempo de acesso varia conforme a região da memória, é importante que haja proximidade entre os dados acessados por uma thread e o núcleo em que ela reside. Portanto, é essencial o conhecimento da topologia da máquina, que possibilita o devido ajuste das aplicações a ela, de modo a aproveitar ao máximo os recursos disponíveis.

Disso vem a necessidade de haver alguma representação da topologia para fornecer as informações necessárias sobre ela, seja diretamente às aplicações ou a outras partes do sistema, que usarão tais informações

para realizar otimizações estática ou dinamicamente. Como exemplo de uso estático, pode-se citar compilação de algoritmos com conhecimento da hierarquia (FATAHALIAN et al., 2006), ou posicionamento de processos MPI (BROQUEDIS et al., 2010a); e, quanto ao uso dinâmico, posicionamento de threads e dados OpenMP (BROQUEDIS et al., 2010b).

No entanto, a disponibilização de tais informações gera custos adicionais, além de ter outras implicações relacionadas ao tamanho das estruturas de dados que podem afetar o desempenho. Assim, é necessário que haja um compromisso entre o tempo de acesso e o espaço ocupado pela representação utilizada. Tempos de acesso muito grandes podem acabar anulando os ganhos das otimizações. Já se as estruturas de dados forem muito espaçosas, pode ser que não possuam boa localidade espacial, dependendo dos padrões de referência aos dados em acessos consecutivos. Isso pode resultar em perda de desempenho ocasionada por faltas de *cache*, tanto no acesso às informações da topologia quanto no acesso pelas aplicações aos seus próprios dados. Entretanto, é possível que a adição de algumas informações facilitem certas consultas sobre a topologia sem causar tais prejuízos, que é o desejado.

1.1 MOTIVAÇÃO

Os exemplos de usos estáticos e dinâmicos dados acima, além de vários outros existentes, com o uso de benchmarks, servem como justificativa para a realização de esforços para desenvolver reprentações com as características citadas, isto é, bom tempo de acesso e uso eficiente da memória.

Para as aplicações, o ideal é que os dados estejam sempre nos níveis de *cache* os mais próximos possíveis, de modo que seu uso nas computações seja mais eficiente. Diante disso, compilação com conhecimento da hierarquia (FATAHALIAN et al., 2006) se vale do fato de que frequentemente problemas podem ser divididos em problemas menores de tamanho variável, e ajustar esses tamanhos à capacidade das *caches* torna o uso delas mais efetivo, pois todos os dados usados nessas partes

menores da computação caberão nelas. Ainda, quando é possível haver vários níveis de subdivisão do problema, formando também uma espécie de hierarquia de subdivisões, os tamanhos das partes em diferentes níveis podem ser ajustados aos níveis de *cache* consecutivos. Isso pode ser visualizado com facilidade no exemplo de multiplicação de grandes matrizes presente no artigo referenciado.

A velocidade de níveis de *cache* mais próximos também beneficia a comunicação. Isso pode ser visto no uso de *Message Passing Interface* (MPI), um padrão utilizado no desenvolvimento de programas paralelos que seguem o modelo de passagem de mensagens. Em conjunto com dados sobre os padrões de comunicação entre processos, as informações sobre compartilhamento de *caches* podem ser usadas para definir um posicionamento de processos MPI que favoreça as comunicações (BROQUEDIS et al., 2010a). Outra otimização possível é o uso de metódos específicos do *hardware* para realizar comunicações dentro de um nó.

No contexto de arquiteturas NUMA, para diminuir o número de acessos a memórias remotas, há a possibilidade de mover os dados para outro nó ou as threads para outros núcleos. Uma combinação dessas opções foi desenvolvida no ForestGOMP (BROQUEDIS et al., 2010b), uma extensão de uma implementação de OpenMP, padrão utilizado no desenvolvimento de programas paralelos para sistemas com memória compartilhada. Seguindo o princípio de realizar essa combinação com base nos níveis da topologia, o posicionamento dinâmico de threads e dados desenvolvido no ForestGOMP se mostrou efetivo. Um cenário apresentado é a existência de vários conjuntos de threads e dados com grande afinidade, em que a migração de uma thread para outro núcleo só ocorreria se houvesse um nível de *cache* compartilhado, de modo a manter a thread próxima dos seus dados, enquanto em outros casos poderia haver a migração de todas as threads e dados relacionados.

Outro projeto que se vale do conhecimento da topologia é o HieSchella (HIERARCHICAL..., 2013), cujo objetivo é prover portabilidade de desempenho, característica presente quando se consegue que uma mesma aplicação rode em diferentes plataformas utilizando os núcleos com eficiência. Informações sobre os custos de comunicação

da plataforma são obtidos e disponibilizados para algoritmos de mapeamento de tarefas. Alguns algoritmos de balanceamento de carga desenvolvidos utilizando o modelo de topologia disponibilizado pelo HieSchella demonstraram desempenho superior ao de outros balanceadores de carga existentes (PILLA, 2014).

Esses exemplos ilustram como informações sobre a hierarquia podem efetivamente ser usadas para melhorar o desempenho de aplicações que seguem modelos ou estratégias em uso real, ou seja, os benchmarks utilizados possuem características encontradas na solução de problemas reais. Isso diz respeito a, por exemplo, padrões de comunicação ou distribuição de carga, que podem apresentar irregularidades e outras características presentes em aplicações científicas de diversas áreas.

1.2 OBJETIVOS

Este trabalho tem como objetivo o desenvolvimento de uma representação de topologias de máquina, compreendendo as estruturas de dados utilizadas e os métodos de acesso, que mantenha o compromisso necessário entre tempo de acesso e espaço ocupado na memória pelas estruturas de dados.

1.2.1 Objetivos Específicos

Os objetivos específicos são:

- Analisar fatores relevantes para a eficiência na representação de topologias
- Desenvolver representações (estruturas de dados e métodos de acesso)
- Testar as representações desenvolvidas, por meio de experimentos em diferentes máquinas, observando o uso da memória e o tempo de execução
- Disponibilizar bases para uma nova ferramenta para a representação de topologias de máquina

1.3 METODOLOGIA

- Estudar organização de computadores com foco na hierarquia de memória
- Estudar como topologias de máquina são representadas em trabalhos e ferramentas do estado da arte
- Entender o protótipo utilizado no Laboratório de Computação Embarcada - *Embedded Computing Lab* (ECL) até o momento
- Implementar novos métodos de armazenamento e acesso às informações
- Testar os novos métodos e estruturas de dados utilizando máquinas com topologias diferentes e avaliar os resultados

1.4 ORGANIZAÇÃO DO TRABALHO

As seções restantes estão organizadas da seguinte forma: No Capítulo 2, serão apresentados alguns conceitos fundamentais relevantes para o trabalho. No Capítulo 3, será fornecida uma visão geral do estado da arte em representação de topologias de máquina. No Capítulo 4, as estruturas e algoritmos desnvolvidos e implementados serão apresentados e analisados e, no Capítulo 5, os resultados dos testes de desempenho realizados com eles. Por fim, as conclusões e trabalhos futuros serão apresentados no Capítulo 6.

2 FUNDAMENTAÇÃO TEÓRICA

As hierarquias de memória possuem várias características que afetam o desempenho de aplicações e precisam, portanto, ser conhecidas ao se trabalhar com aplicações de alto desempenho. Nas seções a seguir são apresentados alguns conceitos importantes nesse contexto, e, ao final, alguns conceitos relacionados a estruturas de dados relevantes.

2.1 MEMÓRIAS *CACHE*

Caches são memórias com o próposito de diminuir o tempo médio de acesso aos dados. Mais especificamente, é comum haver dois ou três níveis de *cache* entre as unidades de processamento e a memória principal. Elas são construídas com tecnologias que as tornam mais rápidas que outros níveis abaixo (PATTERSON; HENNESSY, 2011). Porém, essa velocidade vem em troca de custo mais elevado. Por isso, elas têm espaço de armazenamento menor, além de que memórias maiores podem ter sua velocidade de acesso diminuída, o que também motiva a existência de vários níveis de *cache*.

O princípio de sua funcionalidade é manter à disposição dos programas, de forma rápida, aqueles dados dos quais eles precisam ou que estão usando no momento. Esses dados são disponibilizados conforme a capacidade de armazenamento da *cache*. Esta comumente é menor que o conjunto de todos os dados sobre os quais o programa opera, resultando na necessidade de remover alguns dados para acomodar outros.

2.1.1 Localidade Espacial

Quando um programa referencia determinada posição da memória, é comum que logo em seguida os dados nas posições de memória adjacentes sejam necessários também. Assim é definida a localidade espacial: endereços próximos tendem a ser referenciados um após o outro com pouco tempo de diferença (PATTERSON; HENNESSY, 2011). As *caches* levam isso em conta para beneficiar as aplicações, trazendo dos

níveis de memória abaixo não só o dado requisitado, mas também os dados que o rodeiam, constituindo um bloco. Assim, do ponto de vista da *cache*, a memória é uma sequência de blocos que, quando necessários, são carregados em algum espaço disponível, ou substituem um bloco carregado previamente se não houver espaços disponíveis.

Analisar as *caches* ajuda a entender por que um esquema que usasse estruturas muito espaçosas para reduzir a quantidade de operações e acessos poderia não funcionar bem. No pior caso, acessos consecutivos poderiam ser todos a blocos diferentes, ocasionando o custo de trazer cada um para a *cache* e resultando na poluição da *cache* das aplicações, ou seja, diversos blocos com dados das aplicações seriam substituídos, tornando maior o tempo para acessá-los na próxima vez.

2.2 *NON-UNIFORM MEMORY ACCESS*

Non-Uniform Memory Access (NUMA) é um tipo de multiprocessador com espaço de endereçamento único em que diferentes partes da memória estão mais próximas de alguns núcleos do que dos outros (PATTERSON; HENNESSY, 2011). Deste modo, o tempo de acesso depende do núcleo e da memória acessada. Sistemas desse tipo possuem boa escalabilidade, pois é possível adicionar nós sem prejudicar o tempo de acesso dos núcleos às memórias mais próximas. Estas características diferem das de outro tipo de multiprocessador existente, *Uniform Memory Access* (UMA), em que o tempo de acesso independe do núcleo e da memória, e não há a vantagem de memórias locais com tempo de acesso reduzido. Um problema com arquiteturas UMA é a contenção: vários núcleos disputam pelo barramento para acessar a memória, podendo fazer com que alguns fiquem ociosos enquanto esperam, diminuindo o desempenho. Além disso, a escalabilidade é reduzida – ao aumentar a quantidade de núcleos, o tempo de acesso à memória também cresce, seja qual for a região acessada, podendo se tornar inaceitável.

O modo como os nós NUMA são interconectados determina os custos de comunicação entre quaisquer dois núcleos em nós distintos. Além disso, ele define os níveis que existirão a mais na hierarquia,

ou seja, a sua profundidade. Essas informações são de grande valia e devem ser fornecidas por representações da topologia com precisão. Mesmo poucos níveis a mais na rede de interconexão podem resultar em diferenças de desempenho significativas (RASHTI et al., 2011).

2.3 ESTRUTURAS DE DADOS

Uma *árvore* é uma estrutura de dados que define uma hierarquia, logo, é conveniente para representar hierarquias de memória.

Árvores são compostas por nós e ligações que relacionam esses nós. Cada nó possui dados que dependem do que a árvore representa e de com que propósito ela será usada. Cada ligação relaciona dois nós, um com papel de *pai*, e o outro com papel de *filho*. Ou seja, uma ligação estabelece que um nó a é pai de outro nó b, e, equivalentemente, que o nó b é filho do nó a.

Em uma árvore, existe um único nó que não possui pai, o qual é chamado de *nó raiz* (ou simplesmente *raiz*). Todos os demais nós possuem exatamente um pai. Cada nó pode ter qualquer quantidade de filhos. Se não possui nenhum, é chamado de *nó folha* (ou *folha*). Os filhos de um nó são ordenados. Essa ordem não necessariamente reflete algum atributo ou característica daquilo que os nós estão representando.

Para se realizar operações sobre essa estrutura de árvore, é possível seguir as ligações para descobrir os nós adjacentes (pai e filhos). Visto que árvores geralmente são visualizadas com o nó raiz no topo, *subir* uma ligação significa obter o pai de algum nó, e *descer*, obter um dos filhos de um nó.

Todos os nós que podem ser obtidos subindo, em sequência, uma ou mais ligações a partir de um nó são chamados de *ancestrais* desse nó. Semelhantemente, nós que podem ser obtidos apenas descendo ligações são *descendentes*. Nenhum nó é ancestral de si mesmo.

Uma árvore é dividida em *níveis*, cada um composto por um ou mais nós. O nível 0 é composto pela raiz. O nível i é composto por todos os nós que podem ser obtidos descendo i ligações em sequência a partir da raiz. O nível em que um nó está é também chamado de sua

profundidade.

A quantidade de filhos que um nó possui é chamada de *grau do nó*. O maior entre os graus dos nós que compõe um nível é chamado de *grau do nível*.

3 ESTADO DA ARTE

No estado da arte quanto à representação de topologias de máquina encontra-se o projeto *Hardware Locality* (hwloc) (BROQUEDIS et al., 2010a), um pacote de *software* amplamente utilizado que possui grande portabilidade. Ele contém ferramentas de linha de comando, além de permitir que aplicações acessem as informações sobre a topologia por meio de uma API na linguagem C.

Entre os trabalhos relacionados ao tema, há também o projeto LIKWID (TREIBIG; HAGER; WELLEIN, 2010), um conjunto de utilidades para auxiliar no desenvolvimento de programas com foco no desempenho. Suas funcionalidades são acessadas em parte por uma API, mas principalmente pela linha de comando. É possível, entre outros, obter informações sobre a topologia das memórias *cache* e utilizar contadores de *hardware*. O LIWKID utiliza o hwloc, como pode ser visto no código fonte (PERFORMANCE..., 2017). Outro projeto com propósito semelhante é o Servet (GONZÁLEZ-DOMÍNGUEZ et al., 2010), que obtém as informações utilizando *benchmarks* para estimar as características da máquina.

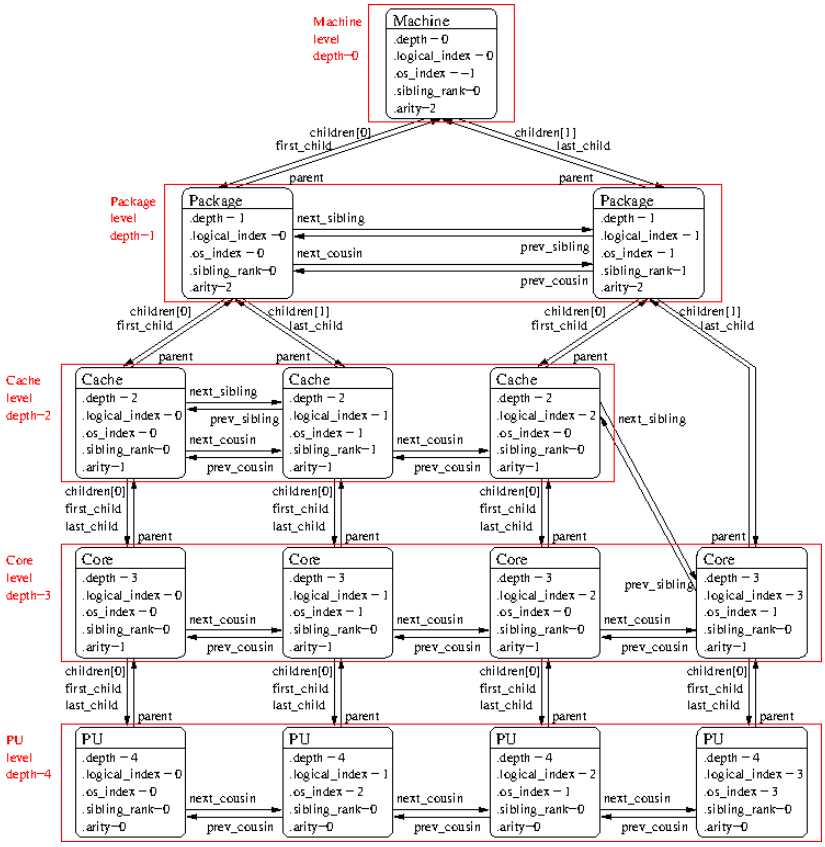
O hwloc foi analisado neste trabalho devido ao seu uso em diversos projetos e, em especial, no HieSchella (HIERARCHICAL..., 2013), o qual poderia ter ganhos significativos de desempenho com os resultados deste trabalho. As seções a seguir apresentam alguns aspectos relevantes do hwloc.

3.1 OBJETOS

Objetos são uma abstração usada para representar todos os elementos presentes na topologia, tanto memória quanto núcleos. A hierarquia de memória é modelada como uma árvore de objetos, onde cada nível contém objetos de apenas um tipo. A topologia por completo é representada de forma mais detalhada por meio de várias ligações (ponteiros) entre objetos, conforme as suas relações na hierarquia. Essas relações são definidas como:

- Pai (*parent*): nó pai na estrutura de árvore
- Filhos (*children*): nós filhos na estrutura de árvore
- Irmãos (*siblings*): nós com o mesmo pai
- Primos (*cousins*): Objetos numa mesma profundidade (e, portanto, de mesmo tipo); todos os objetos que compõe um determinado nível (irmãos ou não) são primos

Figura 1 – Relações entre objetos em topologias do hwloc. Fonte: (PORTABLE. . . , 2016)



A Figura 1 apresenta um exemplo dessas relações em uma dada topologia. Ela ilustra, ainda, no ramo mais a direita, como hierarquias assimétricas (que possuem quantidades diferentes de objetos em ramos partindo do mesmo nó) são tratadas: podem surgir "buracos" devido a objetos de um mesmo tipo serem agrupados no mesmo nível. Portanto, objetos irmãos não estarão necessariamente no mesmo nível, e o conceito de profundidade não é exatamente o usado no contexto de estruturas de árvore, sendo possível irmãos terem profundidades diferentes.

Esse exemplo apenas ilustra o tratamento da assimetria. Em casos reais, hierarquias assimétricas geralmente surgem devido ao acréscimo de objetos (chamados de grupos) na hierarquia para representar melhor a afinidade de dispositivos de entrada e saida, deixando esses dispositivos mais próximos de algumas unidades de processamento do que das demais na árvore.

Essas relações acrescentadas à estrutura de árvore facilitam a navegação entre objetos, em troca do espaço adicional ocupado por cada ponteiro nos objetos. Por exemplo, como indicado na figura, a partir de qualquer objeto, é possível navegar para o próximo primo (pelo ponteiro `next_cousin`), ou para o próximo irmão (ponteiro `next_sibling`), se existirem. Além disso, todos os objetos são armazenados numa estrutura de arranjo de arranjos de objetos, semelhante a uma matriz, mas com arranjos internos de tamanho variável. Cada um dos arranjos internos contém os objetos de um nível, e eles são organizados no arranjo externo na ordem dos níveis, de modo que é possível acessar o enésimo objeto de um dado nível diretamente usando essa estrutura.

Todos os objetos que compõe a hierarquia e outras estruturas de dados utilizadas na representação da topologia são armazenados em uma estrutura (`struct hwloc_topology`) que é utilizada pelas funções que acessam dados da topologia.

3.2 FUNÇÕES E ATRIBUTOS

Existem funções que podem ser utilizadas para acessar os objetos da hierarquia e informações sobre eles. Entre elas, o hwloc possui

diversas funções de percorrimento. Estas permitem acessar nós da árvore que representa a hierarquia de forma absoluta ou relativa a outros nós. Por exemplo, é possível encontrar o nó com um determinado índice dentro de um dado nível, ou, a partir de algum nó, o próximo no mesmo nível. Alguns exemplos de funções são:

- `hwloc_get_nbobjs_by_type`: Informa quantos objetos de um determinado tipo existem, por exemplo, a quantidade de núcleos.
- `hwloc_get_cache_covering_cpuset`: Encontra a primeira memória *cache* que abrange um dado conjunto de CPUs.
- `hwloc_get_ancestor_obj_by_type`: Encontra o ancestral de um objeto que seja de um determinado tipo.
- `hwloc_get_ancestor_obj_by_depth`: Semelhante à função anterior, procurando por nível em vez de tipo.
- `hwloc_get_common_ancestor_obj`: Encontra o ancestral comum mais próximo (ACMP) entre dois objetos, isto é, o nó de maior profundidade que é ancestral de ambos.

Essas funções foram analisadas quanto à complexidade com o objetivo de identificar pontos que poderiam ser melhorados do ponto de vista do desempenho. Essa análise revelou que, em geral, elas têm tempo constante ($O(1)$) ou linear na altura da árvore ($O(\textit{altura})$), que é o mesmo que $O(\log N)$, onde N é a quantidade de nós da árvore. Além disso, foi analisado o projeto HieSchella, o qual possui código aberto e utiliza o hwloc (HIERARCHICAL. . . , 2013). Foram identificadas as chamadas mais importantes a funções do hwloc no HieSchella para se ter uma referência de quais funções são mais relevantes para o desempenho dentro de um projeto real. O Capítulo 4 apresenta o que foi feito em decorrimento dessas análises.

Os objetos ainda têm atributos que podem guardar diversas informações, como detalhes do sistema operacional ou da máquina, que podem ser coletadas automaticamente durante o descobrimento da topologia ou adicionadas manualmente. Além disso, há várias informações

específicas de *caches*, como associatividade ou tamanho. Também é possível associar estruturas arbitrárias aos objetos, conforme for necessário, usando dados de usuário (ponteiro `userdata`).

3.3 CONJUNTOS DE CPUS

Cada objeto pode ter um *cpuset* (conjunto de CPUs), que é um mapeamento dos núcleos existentes para *bits* (*bitmap*), usado para determinar quais núcleos estão sob o objeto na hierarquia, implementado como uma sequência de variáveis de 32 *bits*, tantas quantas forem necessárias. A implementação dos *bitmaps* poderia ser otimizada para diminuir a quantidade de variáveis utilizadas em casos em que existam grandes quantidades de núcleos. Algo nesse sentido é citado no respectivo arquivo fonte em um comentário sobre otimizações que poderiam ser realizadas (HARDWARE. . . , 2016).

4 IMPLEMENTAÇÃO

Esta seção apresenta as análises realizadas com foco no desempenho e os algoritmos e implementações resultantes. Todo o código foi desenvolvido na linguagem C++.

Diante das considerações feitas sobre o projeto hwloc e o seu uso no projeto HieSchella, a função que encontra o ancestral comum mais próximo (ACMP) entre dois nós foi escolhida como alvo de otimizações. Ela é uma das funções implementadas no hwloc com complexidade O(*altura*) (ou O($\log N$)) e está entre as de uso mais significativo no HieSchella. Na seção a seguir, será discutido como essa função poderia ser implementada de forma mais eficiente e quais as implicações de diferentes abordagens.

4.1 IMPLEMENTAÇÕES DA FUNÇÃO ACMP

A função que encontra o ancestral comum mais próximo recebe dois nós como entrada (e possivelmente algumas estruturas adicionais, se houver necessidade) e retorna um nó (o ancestral) como saída. Cada par de nós em uma árvore tem exatamente um ancestral comum mais próximo. Em geral, o caso em que os dois nós são o mesmo (ACMP(a, a)) não será tratado. Se um nó a for ancestral de outro nó b, ACMP(a, b) terá como resultado a.

4.1.1 Abordagem inicial

A maneira provavelmente mais intuitiva de se descobrir o ACMP é "subir" pela árvore, isto é, a partir dos dois nós dos quais se deseja encontrar o ACMP, seguir as ligações em direção aos pais até se chegar ao mesmo nó. Por ser um método que funciona subindo as ligações, será referido como método AscendACMP (Algoritmo 1). Sua complexidade é O($\log N$). Para que esse método funcione, é necessário subir de forma sincronizada – a cada passo, devem-se comparar ancestrais dos dois nós iniciais que estejam no mesmo nível. No caso do hwloc, o algoritmo se torna um pouco mais complicado, pois ele trata hierarquias assimétricas

(pode haver ramos sem nó em algum nível). Neste caso, mesmo que se tenham dois nós no mesmo nível, seus pais podem estar em níveis diferentes. Ao fim de cada iteração, um dos nós pode estar "mais alto" (nível menor), portanto, este método usado pelo hwloc será chamado de OndulACMP (Algoritmo 2), em analogia a uma superfície ondulada. Um ponto que pode afetar o desempenho destes algoritmos é o fato de que é necessário acessar cada nó (os nós iniciais e todos os seus ancestrais até o ACMP), e os nós estão espalhados pela memória.

Algoritmo 1: AscendACMP – ACMP subindo as ligações

Entrada: Dois nós a e b
Saída: O ACMP entre a e b

```
// Inicialmente, encontra ancestrais de a e b no
   mesmo nível
```
enquanto $\text{nível}_a > \text{nível}_b$ **faça**
  $a \leftarrow \text{pai}(a)$
enquanto $\text{nível}_b > \text{nível}_a$ **faça**
  $b \leftarrow \text{pai}(b)$
enquanto $a \neq b$ **faça**
  $a \leftarrow \text{pai}(a)$
  $b \leftarrow \text{pai}(b)$
retorna a

Algoritmo 2: OndulACMP – ACMP implementado no hwloc

Entrada: Dois nós a e b
Saída: O ACMP entre a e b

enquanto $a \neq b$ **faça**
  enquanto $\text{nível}_a > \text{nível}_b$ **faça**
    $a \leftarrow \text{pai}(a)$
  enquanto $\text{nível}_b > \text{nível}_a$ **faça**
    $b \leftarrow \text{pai}(b)$
  se $a \neq b$ *e* $\text{nível}_a = \text{nível}_b$ **então**
    $a \leftarrow \text{pai}(a)$
    $b \leftarrow \text{pai}(b)$
retorna a

Considerando a estrutura de árvore apenas, a única informação que relaciona um nó aos seus ancestrais são as ligações. Isso indica que outras estruturas associadas aos nós ou à árvore como um todo se fazem necessárias para ser possível encontrar o ACMP com algum método além dos mencionados. Podemos considerar a seguinte ideia para encontrar outra maneira de implementar a função ACMP: Para uma dada árvore que representa uma topologia,

1. Atribuir um valor (chamado de ID) a cada nó da árvore;
2. Definir uma função ACMP_{IDs} que receba o ID de dois nós distintos e tenha como resultado o nó ACMP.

Esses IDs (em conjunto com outras informações associadas a cada nó individualmente ou à árvore como um todo conforme necessário) podem estabelecer alguma relação entre um nó e seus ancestrais além da que já existe por meio das ligações da árvore.

Idealmente, para implementar essa função ACMP_{IDs}, deveria ser encontrado um algoritmo de complexidade constante. No entanto, é preciso lembrar que, mesmo que a quantidade de instruções executadas pelo processador seja constante, a maneira como a memória é acessada pode aumentar o tempo de execução, especialmente quando há outras tarefas fazendo uso da memória, o que deve acontecer em cenários reais.

Com isso em mente, podemos analisar diferentes formas de definir tal função para descobrir qual seria mais adequada.

4.1.2 Matriz

Uma possibilidade é relacionar cada par de nós ao seu ACMP por meio de uma matriz em que cada linha representa um nó da árvore, assim como cada coluna, e o cruzamento contém o ACMP entre o nó da linha e o nó da coluna. Para isso, pode-se atribuir a cada um dos N nós da árvore um ID único entre 0 e $N-1$ e usar esses IDs como índices na matriz, que terá, na posição $A(i,j)$, o ACMP entre o nó de ID i e o nó de ID j. No entanto, esta é uma estratégia ingênua, pois esse espaço $\text{O}(N^2)$ ocupado na memória resultaria em problemas como

sujar a *cache* da aplicação. A Tabela 1 sumariza a diferença entre a utilização de uma árvore e de uma matriz para a função ACMP. Visto que a matriz é simétrica, apenas cerca de metade dela precisa realmente ser armazenada. Esta otimização foi utilizada na implementação dos testes de desempenho, conforme o Algoritmo 3. Isso, no entanto, não altera a complexidade espacial.

Tabela 1 – Complexidade das representações com árvore e com matriz

	Árvore	Matriz
Acesso (encontrar ACMP)	$O(\log N)$	$O(1)$
Espaço ocupado na memória	$O(N)$	$O(N^2)$

Algoritmo 3: MatrizACMP – ACMP usando uma matriz

Entrada: Dois nós a e b
Uma matriz M que possui na linha i e coluna j ($M(i,j)$), $i > j$, o ACMP entre os nós de ID i e j
Saída: O ACMP entre a e b

$\text{linha} \leftarrow \max(\text{id}_a, \text{id}_b)$
$\text{coluna} \leftarrow \min(\text{id}_a, \text{id}_b)$
retorna $M(\text{linha}, \text{coluna})$

4.1.3 Função de espalhamento

Outra possibilidade foi idealizada, dividindo a função ACMP_{IDs} em dois passos:

1. dados os IDs de dois nós, descobrir o ID do ancestral;
2. encontrar o nó que possui esse ID.

Em linhas gerais, o funcionamento do primeiro passo se baseia no seguinte: O ID de um nó aparece na representação binária dos IDs de todos os seus descendentes. Desse modo, dados os IDs de dois

descendentes, é possível identificar o ID do ACMP. Usando apenas as operações que podem ser vistas nas linhas 1 – 4 do Algoritmo 4 mais adiante, para as quais existem instruções que tomam poucos ciclos nas arquiteturas atuais, pode-se descobrir o ID do ACMP. A quantidade de instruções é fixa, portanto, a complexidade é constante. Essa estratégia será chamada de CombinACMP pelo modo como o ID do ancestral é extraído da parte dos IDs em que eles combinam.

Para descrever como os IDs são formados, as seguintes definições são necessárias:

- id_a é o ID do nó a.
- $\mathrm{id}_a^{\mathrm{str}}$ é uma cadeia (*string*) de *bits* correspondente ao id_a em binário (com o *bit* menos significativo na última posição). O tamanho depende do nível de a, como será especificado adiante.

Os IDs, então, são definidos da seguinte forma:

- A raiz tem id 0, e $\mathrm{id}_{raiz}^{\mathrm{str}}$ é a cadeia vazia.
- Quanto aos demais, para cada nó a,

  $$\mathrm{id}_a^{\mathrm{str}} = x \parallel \mathrm{id}_{pai(a)}^{\mathrm{str}}$$

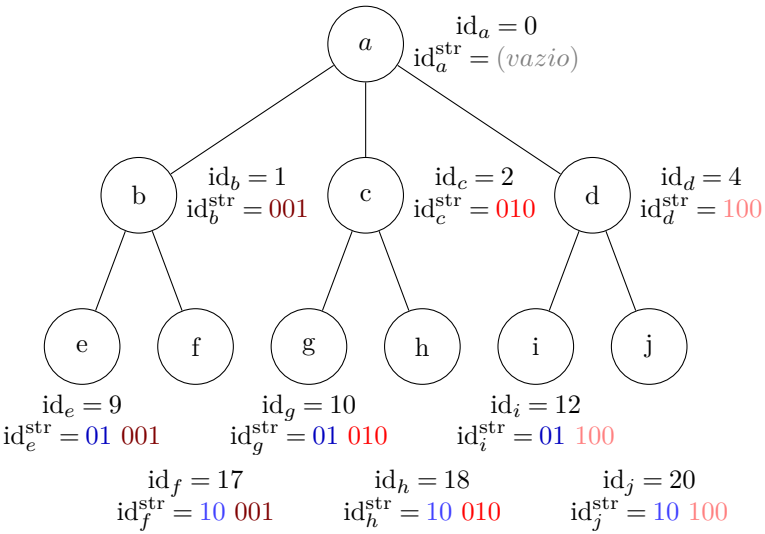
  onde $\parallel$ é a concatenação e x é uma cadeia cujo tamanho é o grau do nível de pai(a). Se a é o i-ésimo filho de seu pai, x possui 1 na i-ésima posição da direta para a esquerda e 0 nas demais.

Assim, as cadeias correspondentes aos IDs de todos os nós de um nível têm o mesmo tamanho, que é o somatório dos graus dos níveis anteriores. A Figura 2 apresenta os IDs atribuídos aos nós de uma árvore.

Falta, então, apenas o segundo passo, o de encontrar o nó a partir do ID. Algumas opções para isso seriam:

- Usar os IDs como índices em um arranjo: Seria simples, mas impraticável – poderiam ser necessários arranjos com milhões de posições (devido a como os IDs são formados) e apenas algumas centenas ocupadas.

Figura 2 – Exemplo de árvore com IDs atribuídos aos nós



- Usar uma função de espalhamento (*hash*): a maneira mais simples seria apenas aplicar a operação módulo com algum m (id mod m). No entanto, podem haver colisões (dois IDs diferentes podem ser congruentes módulo m). Isso pode ser tratado, mas acarretaria acessos adicionais à memória, o que não é desejável. Por exemplo, para os IDs do nível 1 da Figura 2, $(1,2,4)$, $m = 3$ resultaria em $(1,2,1)$, com dois nós mapeados para a posição 1.

- Buscar uma função que não cause colisões: o mesmo que a função de espalhamento, porém utilizando um valor de módulo que não cause colisões. Pode exigir arranjos cujo tamanho é algumas vezes maior que a quantidade de nós, mas aparenta compensar quando comparado a tratar colisões de outros modos. Por exemplo, com os mesmos IDs, $m = 4$ resultaria em $(1,2,0)$, sem colisões (e, coincidentemente, exigiria um arranjo de apenas três posições). Este foi o método escolhido.

As linhas 5 – 7 do Algoritmo 4 apresentam como este método funciona. Quanto à implementação da função BitParaNível, os pro-

Algoritmo 4: CombinACMP – ACMP usando as novas estruturas

Entrada: Dois nós a e b
Uma função BitParaNível, que retorna o nível do ancestral comum entre dois nós dado um número com apenas um *bit* 1, na primeira posição em que o ID dos nós difere
A função Espalhamento (Algoritmo 5) com dados da árvore em questão
Uma matriz (vetor de vetores) E, onde o vetor $E(i)$ possui os nós do nível i distribuídos pela função Espalhamento(i, id)

Saída: O ACMP entre a e b

```
// Bits que diferem
1 dif ← id_a OuEx_bit-a-bit id_b
// Bit 1 apenas na primeira posição em que os IDs diferem
2 bit ← dif E_bit-a-bit (−dif)
// Todos os bits antes do primeiro diferente
3 masc ← bit − 1
// Id do ancestral comum mais próximo
4 id ← id_a E_bit-a-bit masc
// Encontra o nó
5 nível ← BitParaNível(bit)
6 pos ← Espalhamento(nível, id)
7 retorna E(nível, pos)
```

cessadores atuais possuem uma instrução que encontra a posição do primeiro *bit* 1 em um número, a qual pode ser usada como índice em um pequeno arranjo contendo o devido resultado da função BitParaNível. Se os IDs têm até b *bits*, esse arranjo precisar ter b posições. A função Espalhamento (Algoritmo 5) utiliza dados específicos para a árvore, que devem ser descobertos previamente.

Testes feitos indicaram que a operação mais custosa no CombinACMP era o módulo, usado na função Espalhamento. No entanto, existem técnicas para realizar de maneira mais barata a divisão com denominador previamente conhecido (RECIPROCAL. . . ,

Algoritmo 5: ESPALHAMENTO – Função de espalhamento

Entrada: Um nível n ($n \in \mathbb{Z}$, $n \geq 0$)
O id de um nó

Saída: A posição em que o nó deve ficar no vetor do seu nível

Nota: Os valores ad_n, m_n, mult_n e desl_n dependem da árvore com a qual se usará esta função e do nível recebido (n).
x $\mathbf{Desl_{direita}}$ b é o valor x deslocado b *bits* para a direita, equivalente a $\lfloor x/(2^b) \rfloor$.

```
// Minimiza o maior resultado mod m_n para
   minimizar o tamanho do vetor
```

$\mathrm{v} \leftarrow \mathrm{id} + \mathrm{ad}_n$

```
// Equivalente a v mod m_n
```

retorna $\mathrm{v} - ((\mathrm{v} * \mathrm{mult}_n)$ $\mathbf{Desl_{direita}}$ $\mathrm{desl}_n) * m_n$

2002), e com o resultado da divisão pode-se calcular o módulo. Essas otimizações são usadas, por exemplo, por compiladores, sendo chamadas de Redução de Força. Aqui, no entanto, a ideia não é realizá-las em tempo de compilação, mas quando se está montando as estruturas para uma hierarquia específica. Isso continua sendo vantajoso pois a montagem ocorre apenas uma vez e esta operação de módulo com o mesmo valor será realizada uma grande quantidade de vezes. Portanto, é possível substituir a operação de módulo por outras operações que se mostraram mais baratas, a saber, duas multiplicações, um deslocamento e uma subtração.

Para se encontrarem valores apropriados para a função ESPALHAMENTO, são testados valores cada vez maiores para m, até que não haja colisões. Ao se encontrar um m válido, se busca um ad que minimize o maior resultado de $(\mathrm{id} + ad) \bmod m$, minimizando o tamanho do arranjo usado. Então, são descobertos os valores de *mult* e *desl*, usando a técnica descrita em Reciprocal... (2002), para evitar a operação de módulo.

A corretude das estruturas utilizadas na implementação do COMBINACMP foi testada. O programa de testes cria uma árvore

e compara o resultado do AscendACMP com o do CombinACMP para algumas buscas, além de verificar se todos os nós estão realmente na posição retornada pela função Espalhamento. Também é causada uma mutação em uma árvore e é verificado se o erro é detectado por esse código de verificação.

O CombinACMP, assim como o OndulACMP e o MatrizACMP, trata corretamente hierarquias assimétricas, diferentemente do AscendACMP. No entanto, ele possui uma limitação: A quantidade de *bits* do ID de um nó pode chegar até o somatório dos graus de todos os níveis. Se esta quantidade ultrapassar o tamanho de uma palavra (geralmente 32 ou 64 *bits*), o método terá problemas, limitando as árvores que podem ser usadas. Além disso, a técnica usada na função Espalhamento para otimizar o módulo inclui uma multiplicação cujo resultado pode ter até cerca do dobro da quantidade de *bits* do penúltimo nível (o último nível não possui espalhamento, pois seus nós nunca serão ACMP). Como uma solução parcial, níveis de grau um podem ser omitidos na definição dos IDs, visto que seus nós também nunca serão ACMP. Atribuindo a cada nó desses níveis e ao seu filho o mesmo ID, eles continuam podendo ser usados em buscas de ACMP. Deste modo, o tamanho máximo seria o somatório dos graus de todos os níveis de grau diferente de 1.

O método poderia ser adaptado para usar mais palavras se necessário, teoricamente deixando de ser $O(1)$ e se tornando $O(\log N)$, mas, na prática, nenhuma árvore deve ser tão grande que exija uma quantidade significativa.

O código implementado tem suporte à aplicação das operações `Ou` e `Ou-Exclusivo` ao ID no início da função Espalhamento, com valores que também precisariam ser descobertos previamente, o que poderia reduzir ainda mais o tamanho dos arranjos usados. No entanto, devido a restrições de tempo, a busca desses valores não foi implementada.

5 TESTES DE DESEMPENHO

Com o objetivo de avaliar o desempenho do algoritmo desenvolvido, foram realizados testes que permitiram comparar o desempenho das diferentes abordagens. Os dados foram obtidos medindo o tempo tomado pelos algoritmos ao se encontrar repetidamente o ACMP entre os nós folhas de uma árvore.

Um programa foi desenvolvido, também na linguagem C++, para realizar os testes de desempenho. Foi utilizada a funcionalidade de templates da linguagem para facilitar a definição equivalente dos testes para todos os algoritmos, mas ainda assim permitir que o compilador otimizasse as chamadas, evitando custos adicionais durante os testes devido à hierarquia de classes utilizada. O programa depende da biblioteca do hwloc. Os valores medidos são escritos em um arquivo no formato CSV (Valores Separados por Vírgula – *Comma-Separated Values*)

5.1 MÁQUINAS UTILIZADAS NOS TESTES

Os testes foram executados sobre duas máquinas (*notebooks*), que serão identificadas como Máquina A e Máquina B. A Tabela 2 apresenta detalhes de ambas. A Tabela 3 apresenta o tamanho das suas *caches*, que podem ter efeito nos resultados dos testes. Outras características da topologia das máquinas, no entanto, não são relevantes, pois os testes são totalmente sequenciais e as buscas de ACMP são realizadas sobre uma árvore passada como argumento ao executar o programa, representando uma topologia arbitrária, e não a da máquina em que os testes estão rodando.

5.2 CONFIGURAÇÕES

Em ambas as máquinas, os testes foram compilados usando a versão 5.3.0 do compilador GCC (*GNU Compiler Collection*) (GCC..., 2017), inclusa no projeto Cygwin (CYGWIN, 2017), o qual emula um sistema Unix em versões atuais do sistema operacional Windows. Esta

Tabela 2 – Características das máquinas utilizadas nos testes

	Máquina A	Máquina B
Sistema Operacional	Windows 10 Home 64 *bits*	
Processador	Intel® Core™ i5 5200U 2.20 GHz	Intel® Core™ i7 6600U 2.50 GHz
Memória Principal	DDR3 6 GB 798.7 MHz	DDR4 8 GB 1067 MHz

Tabela 3 – *Caches* das máquinas utilizadas nos testes

Cache	Máquina A	Máquina B
L1	32 KB	
L2	256 KB	
L3	3072 KB	4096 KB

versão do GCC era a única disponível no Cygwin durante o desenvolvimento do trabalho com a qual não foram encontrados problemas com funcionalidades da linguagem C++. Foram usadas as seguintes *flags* de compilação:

- `-std=c++11`: Utiliza funcionalidades do padrão C++11 da linguagem C++
- `-O3`: Ativa diversas otimizações
- *flags* obtidas com os comandos `pkg-config -cflags hwloc` e `pkg-config -libs hwloc` no Cygwin, que imprimem as *flags* necessárias para utilizar o hwloc

5.3 ESTRUTURA DOS TESTES

Os testes consistem em encontrar repetidamente o ACMP entre os nós folhas de uma dada árvore com os diferentes algoritmos. Cada vez

que as estruturas necessárias são criadas e os testes são executados sobre elas com um determinado algoritmo, obtém-se uma observação, que é o tempo que levou para realizar a quantidade especificada de repetições da função ACMP entre os nós folhas da árvore com o algoritmo. É usada uma grande quantidade de repetições pois não é possível medir corretamente o tempo de apenas uma chamada à função, o qual é menor que a resolução das chamadas de temporização no processador.

Para cada algoritmo, uma árvore de estrutura equivalente é criada (as estruturas de dados dependem do algoritmo) a partir dos graus fornecidos como entrada. Os graus são recebidos como uma lista de inteiros positivos. O i-ésimo será o grau do nível $i-1$. Deste modo, se os graus recebidos são $(g_1, g_2, ..., g_n)$, a raiz terá g_1 filhos, cada um com g_2 filhos, e assim por diante, até os nós do nível $n-1$, que terão g_n filhos cada.

Para obter uma observação de um algoritmo, uma lista contendo todos os pares possíveis de nós folhas (todas as combinações de duas folhas) é criada. Esta lista é embaralhada, usando uma semente fixa, de modo que a ordem pseudo-aleatória é a mesma para todas as execuções de todos os algoritmos sobre esta árvore. Isto é feito para evitar que o desempenho dos algoritmos seja beneficiado pelo acesso repetido dos mesmos nós, o que não corresponde a situações reais.

Uma observação é obtida por meio de uma etapa de aquecimento seguida de uma de medição, na qual o tempo total das repetições da função ACMP é medido. Ambas as etapas consistem em algum número de rodadas, o qual geralmente deve estar na casa de alguns milhares, dependendo do tamanho da árvore, para que o tempo medido seja significativo. Em cada rodada, a lista previamente embaralhada de pares de folhas é varrida e, para cada par, o ACMP é encontrado usando o algoritmo em questão.

As observações dos diferentes algoritmos são realizadas de forma intercalada. O número de observações obtidas para cada algoritmo em uma execução do programa depende de dois parâmetros, número de iterações externas e de iterações internas. O número de iterações internas é a quantidade de observações que serão obtidas para um dos

algoritmos antes de passar para outro. A execução da quantidade de iterações internas para cada algorimo compõe uma iteração externa. Esta forma de especificar a quantidade de observações originou-se nas etapas iniciais dos testes, para facilitar a visualização dos resultados, mas foi mantida. No entanto, julga-se melhor usar poucas iterações internas e mais externas para evitar que eventuais condições temporárias da máquina, causadas por elementos externos ao programa, afetem diversas observações de apenas um dos algoritmos.

O programa também permite escolher quais algoritmos serão testados dentre os quatro analisados.

5.3.1 RESULTADOS

As árvores utilizadas nos testes correspondem à hierarquia de memória de máquinas reais, apresentadas no site do projeto hwloc, como representadas pelo programa lstopo (THE. . . , 2016). Assim, os resultados refletem a diferença dos algoritmos quando operando sobre hierarquias reais.

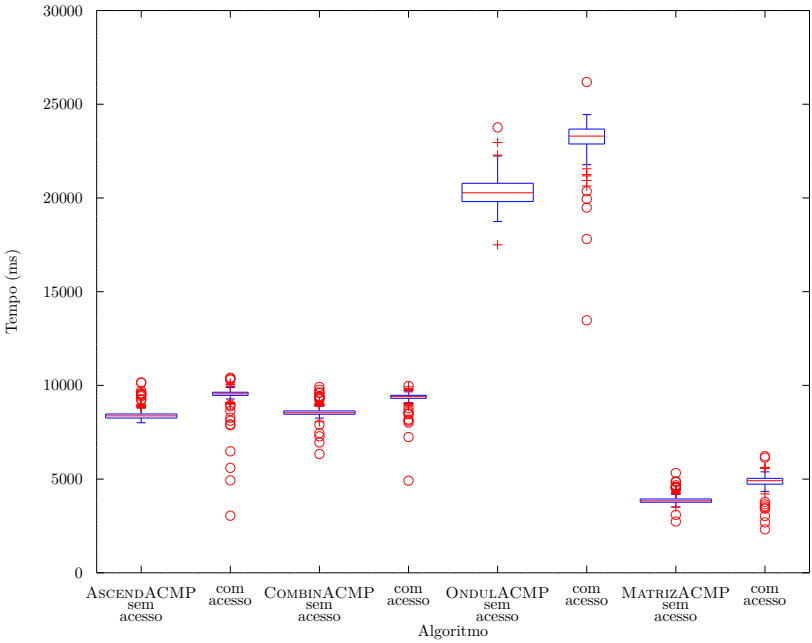
Foram usadas, em todas as execuções, mil rodadas na etapa de aquecimento e dez mil na etapa de medição. Foram usadas sempre três iterações internas.

Notou-se a presença de diversos *outliers* de valor significativamente mais baixo nas observações obtidas. Acredita-se que isso se deva a momentos em que os processos do sistema operacional em segundo plano em conjunto tenham coincidentemente requerido pouco processamento.

Inicialmente, os testes foram feitos sem acessar o ACMP obtido a cada execução, ou seja, para cada par de folhas, simplesmente se encontrava o ponteiro para o ancestral, mas nenhum dado do ancestral era obtido. No entanto, em cenários reais, os nós seriam buscados para se obter alguma informação sobre eles, portanto, foi adicionado um acesso a um valor qualquer de cada ACMP encontrado para simular isso. A Figura 3 compara o desempenho dos quatro algoritmos na Máquina A para a árvore com os graus $(1, 4, 1, 1, 9, 2, 1, 1, 4)$ antes e depois do acréscimo desse acesso. Sem o acesso foram feitas 300 observações

de cada algoritmo e, com o acesso, 150 observações. Como era esperado, os tempos aumentaram para todos os algoritmos, porém o algoritmo menos afetado foi o novo e o mais afetado foi o da matriz, considerando a porcentagem de aumento do valor mediano após acrescentar o acesso, conforme a Tabela 4.

Figura 3 – Tempos com e sem acesso ao ACMP



Cada retângulo representa o intervalo onde estão as 50% observações centrais. A linha vermelha dentro do retângulo indica a mediana. O símbolo "+" indica observações distantes da mediana em comparação com as demais (*outliers*), e o símbolo "o", observações ainda mais distantes.

Cinco árvores distintas foram usadas nos testes. Dados sobre elas são apresentados na Tabela 5. Para cada algoritmo e com cada árvore, foram feitas 150 observações na Máquina A e 450 observações na Máquina B, que teve menores tempos de execução, possibilitando esse maior número de observações. As Figuras 4 e 5 apresentam o tempo médio por chamada da função ACMP, calculado usando a mediana dos

Tabela 4 – Aumento (%) do tempo mediano

	AscendACMP	CombinACMP	OndulACMP	MatrizACMP
Mediana (ms) sem acesso	8373.99	8540.89	20274.45	3841.80
Mediana (ms) com acesso	9549.51	9401.76	23300.20	4914.60
Aumento (%)	14,04	10,08	14,92	27,92

dados, para cada uma das árvores. É possível ver que o CombinACMP foi o que teve a menor variação nos tempos diante do uso de árvores com características distintas, como tamanho (quantidade de nós) da árvore e quantidade média de ligações entre os nós usados nas buscas e seus respectivos ACMPs. O MatrizACMP, apesar da complexidade constante, com uma árvore com cerca de cinco vezes mais nós, teve tempos médios 73% maior na Máquina A e 52% maior na Máquina B.

Tabela 5 – Árvores utilizadas nos testes

Graus	Nº de nós	Nº de folhas
2 1 1 8 1 1 1 2	103	32
8 1 1 6 1 1 1	217	48
4 4 1 3 2 1 1 1	469	96
2 2 1 5 1 1 1 1 8	271	160
1 4 1 1 9 2 1 1 4	554	288

Figura 4 – Tempo médio por busca (ns) – Máquina A

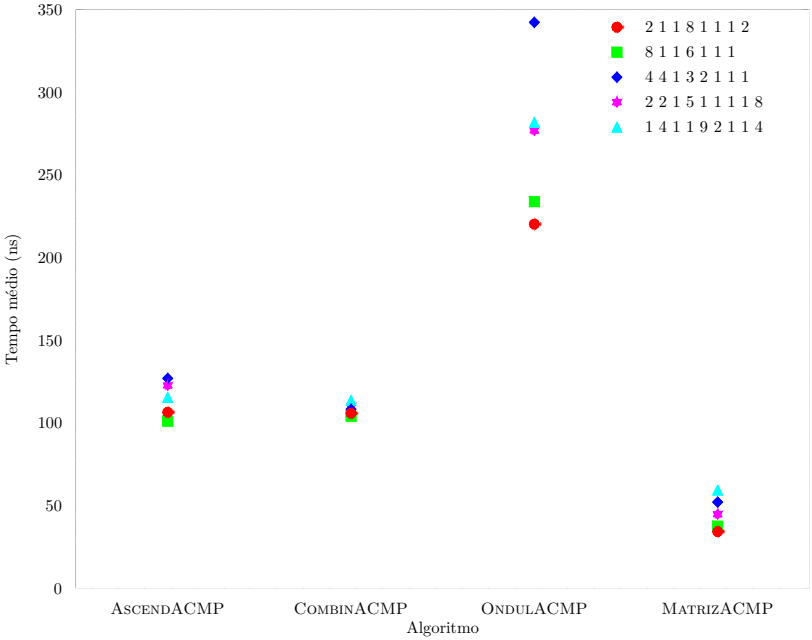
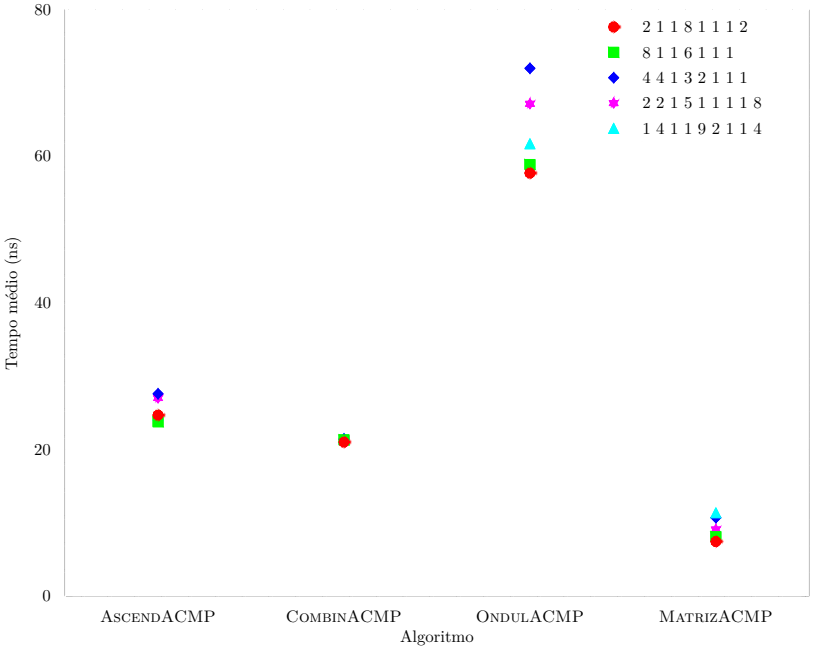


Figura 5 – Tempo médio por busca (ns) – Máquina B



6 CONCLUSÃO

O objetivo deste trabalho foi desenvolver formas otimizadas de representar topologias de máquina. O projeto hwloc, estado da arte em representação de topologias, foi analisado e a função que encontra o ACMP foi identificada como um ponto relevante a se otimizar. Com essa análise, novas estruturas foram desenvolvidas, além de um algoritmo utilizando-as, CombinACMP. Esse algoritmo e outras abordagens foram implementados e testados.

O CombinACMP mostrou características promissoras, como menor variação no tempo de execução e estruturas que devem originar acessos à memória menos esparsos do que nas outras abordagens. O MatrizACMP, apesar de ter tido tempos menores, mesmo sendo um algoritmo de complexidade constante, se mostrou sensível ao aumento do tamanho das árvores utilizadas. Não foram feitos testes com aplicações reais, mas a variação do tempo observada nos testes realizados é uma evidência de que seu uso da memória prejudicaria o desempenho. O desempenho do AscendACMP foi semelhante ao do CombinACMP, mas pode também estar sujeito à influência do padrão de acessos à memória, assim como o OndulACMP, utilizado pelo hwloc, que ainda necessita de um processamento maior.

Este algoritmo desenvolvido, CombinACMP, poderia ser utilizado no hwloc, realizando-se as devidas adaptações. Com base nesta otimização em conjunto com outras que possam ser realizadas, à medida que as estruturas e métodos usados se tornassem incompatíveis com a implementação do hwloc, uma nova ferramenta à parte poderia ser desenvolvida, tendo em foco considerações semelhantes às feitas neste trabalho em relação ao uso da memória.

Para isso, como trabalhos futuros, pode-se avaliar o desempenho dos algoritmos diante de aplicações reais e fazer outras análises, como acompanhar o uso da memória com mais detalhes. Se isto se mostrar relevante, outras técnicas podem ser estudadas para diminuir o tamanho das estruturas usadas no CombinACMP, incluindo a aplicação de máscaras `Ou` e `Ou-Exclusivo` aos IDs na função Espalhamento, para

o que há suporte no código desenvolvido, embora não tenha sido usado.

Quanto a outras funções presentes no hwloc, pode-se estudar a possibilidade e o impacto de otimizações na representação do conjunto de CPUs dos objetos, o qual é usado, por exemplo, para determinar se um objeto está sobre outro na hierarquia de memória. Além disso, outra função identificada como relevante no projeto HieSchella é a `hwloc_get_ancestor_obj_by_type`, que encontra o ancestral em determinado nível, também deixando espaço para se estudar se otimizações teriam efeito significativo no desempenho.

REFERÊNCIAS

BARROSO, L. et al. Attack of the killer microseconds. *Communications of the ACM*, ACM, v. 60, n. 4, p. 48–54, 2017.

BROQUEDIS, F. et al. hwloc: a Generic Framework for Managing Hardware Affinities in HPC Applications. In: *PDP 2010-The 18th Euromicro International Conference on Parallel, Distributed and Network-Based Computing*. [S.l.: s.n.], 2010. p. 180–186. ISSN 1066-6192.

BROQUEDIS, F. et al. ForestGOMP: an efficient OpenMP environment for NUMA architectures. *International Journal of Parallel Programming*, Springer, v. 38, n. 5-6, p. 418–439, 2010.

CYGWIN. 2017. Disponível em: <https://www.cygwin.com>. Acesso em: 31/05/2017.

FATAHALIAN, K. et al. Sequoia: Programming the Memory Hierarchy. In: *Proceedings of the 2006 ACM/IEEE Conference on Supercomputing*. New York, NY, USA: ACM, 2006. (SC '06). ISBN 0-7695-2700-0. Disponível em: <http://doi.acm.org/10.1145/1188455.1188543>.

FRANCESQUINI, E.; GOLDMAN, A.; MÉHAUT, J.-F. A numa-aware runtime environment for the actor model. In: IEEE. *Parallel Processing (ICPP), 2013 42nd International Conference on*. [S.l.], 2013. p. 250–259.

GCC, the GNU Compiler Collection. 2017. Disponível em: <https://gcc.gnu.org>. Acesso em: 31/05/2017.

GONZÁLEZ-DOMÍNGUEZ, J. et al. Servet: A benchmark suite for autotuning on multicore clusters. In: *24th IEEE International Symposium on Parallel and Distributed Processing, IPDPS 2010, Atlanta, Georgia, USA, 19-23 April 2010 - Conference Proceedings*. [S.l.]: IEEE, 2010. p. 1–9.

HARDWARE locality (hwloc). 2016. Disponível em: <https://github.com/open-mpi/hwloc/>. Acesso em: 16/07/2016.

HIERARCHICAL Scheduling for Large Scale Architectures. 2013. Disponível em: <http://forge.imag.fr/projects/hieschella/>. Acesso em: 14/07/2016.

PATTERSON, D. A.; HENNESSY, J. L. *Computer Organization and Design: The Hardware/Software Interface*. 4th. ed. [S.l.]: Morgan Kaufmann, 2011.

PERFORMANCE monitoring and benchmarking suite. 2017. Disponível em: <https://github.com/RRZE-HPC/likwid>. Acesso em: 09/07/2017.

PILLA, L. L. *Topology-Aware Load Balancing for Performance Portability over Parallel High Performance Systems*. Tese (Doutorado) — Université de Grenoble; UFRGS, 2014.

PORTABLE Hardware Locality (hwloc) Documentation: v1.11.3. 2016. Disponível em: <https://www.open-mpi.org/projects/hwloc/doc/v1.11.3/a00002.php>. Acesso em: 16/07/2016.

RASHTI, M. J. et al. Multi-core and network aware mpi topology functions. *Recent Advances in the Message Passing Interface*, Springer, jan 2011. Disponível em: <http://dx.doi.org/10.1007/978-3-642-24449-0_8>.

RECIPROCAL Multiplication, a tutorial. 2002. Disponível em: <http://homepage.divms.uiowa.edu/~jones/bcd/divide.html>. Acesso em: 01/06/2017.

THE Best of lstopo. 2016. Disponível em: <https://www.open-mpi.org/projects/hwloc/lstopo/>. Acesso em: 29/05/2017.

TREIBIG, J.; HAGER, G.; WELLEIN, G. LIKWID: A lightweight performance-oriented tool suite for x86 multicore environments. In: *Proc. 39th Int. Conf. Parallel Processing Workshops*. [S.l.: s.n.], 2010. p. 207–216. ISSN 0190-3918.

APÊNDICE A – CÓDIGO DESENVOLVIDO

A.1 ESTRUTURAS (COMBINACMP)

arv/arv.h

```
1  #ifndef ARV_H
2  #define ARV_H
3
4  #include <iostream>
5  #include <bitset>
6  #include <climits>
7  #include <string>
8  #include <vector>
9
10 #include <algorithm>
11
12 #include "id.h"
13 #include "arv_mod.h"
14 #include "config_dbg.h"
15
16 struct estr_No;
17 typedef estr_No* No;
18
19 struct estr_No
20 {
21   // Id usado para encontrar ancestral mais próximo
22   ID id;
23
24   No pai;
25   int numFilhos;
26   No *filhos;
27
28   // Nível do nó: 0 -> raiz
29   int nivel;
30
31   void imprimir(int nBits = sizeof(id)*CHAR_BIT) { imprimir(0,
         nBits); }
32
33   friend inline std::ostream& operator<<(std::ostream &o, const No
         n) { o << n->id; return o; } // !!!
34
35   void imprimir(int ind, int nBits)
36   {
37     for (int i = 0; i < ind; i++)
38       std::cout << '|'; // Indentação
39     std::cout << "No " << idBin(nBits)
40       /*<< " (" << nivel << ' ' << numFilhos << ')'*/ << '\n';
41     ++ind;
```

```
42      for (int f = 0; f < numFilhos; f++)
43        filhos[f]->imprimir(ind, nBits);
44    }
45
46    std::string idBin(int bits)
47    {
48      return ::idBin(id, bits);
49    }
50
51    ~estr_No()
52    {
53      for (auto i = 0; i < numFilhos; i++)
54        delete filhos[i];
55      delete filhos;
56    }
57  };
58
59  class Arvore
60  {
61    static const bool DEBUG = ConfigDebug::Arvore::DEBUG;
62    static const int DEBUG_MAX_LINHAS_IDS =
          ConfigDebug::Arvore::DEBUG_MAX_LINHAS_IDS;
63
64    public:
65    No raiz;
66
67    // Quantidade de níveis
68    int numNiveis;
69
70    // Quantidade de nós e grau de cada nível
71    int *nosPorNivel, *grauPorNivel;
72
73    // Nós de cada nível
74    No **nosNiveis;
75
76    // Arranjos por nível com nós na posição resultante
77    No **imagemNiveis;
78    // NOTA: Níveis cujos nós têm um só filho poderiam ser totalmente
79    // omitidos na busca do ancestral mais próximo, desde o bit
          ocupado
80    // desnecessariamente no id até as estruturas (pois não são
          ancestral
81    // mais próximo de nenhum par de nós distintos), mas isso
          exigiria um
82    // tratamento todo especial...
83    // A não ser que os nós tenham um No *filhosEspecial.
84
85    // nivelDosBits[i]: Nível do ancestral mais próximo de a e b
          quando
86    // o primeiro bit que difere entre a->id e b->id é i
```

```
87  int *nivelDosBits;
88
89  // Função usada para cada nível da árvore
90  // Folhas não precisam
91  DadosFuncao *dadosFuncao;
92
93  Arvore(No r, int nNiveis)
94  {
95    raiz = r;
96    numNiveis = nNiveis;
97    dadosFuncao = new DadosFuncao[numNiveis];
98
99    montarEstruturas();
100 }
101
102 ~Arvore()
103 {
104   delete raiz;
105
106   delete[] nosPorNivel;
107   delete[] grauPorNivel;
108   for (int i = 0; i < numNiveis; i++)
109     delete[] nosNiveis[i];
110   delete[] nosNiveis;
111
112   for (int i = 0; i < numNiveis-1; i++)
113     delete[] imagemNiveis[i];
114   delete[] imagemNiveis;
115   delete[] nivelDosBits;
116   delete[] dadosFuncao;
117 }
118
119 // Descobrir quantos nós por nível
120 // Descobrir quantos níveis são realmente necessários
121 // Pegar todos os nós de um nível em um array e descobrir o m
122
123 int bitParaNivel(ID bit)
124 {
125   int pos = __builtin_ctz(bit); // Primeiro bit 1
126   return nivelDosBits[pos];
127 }
128
129 void montarEstruturas()
130 {
131   if (DEBUG)
132     std::cout << "<Arvore::montarEstruturas>\n";
133
134   // Descobre quantos nós há em cada nivel e o grau máximo
135   nosPorNivel = new int[numNiveis] (); // Zerado
136   grauPorNivel = new int[numNiveis] (); // Zerado
```

```
137     preencherNosGrauPorNivel(nosPorNivel, grauPorNivel, raiz);
138
139     // Preenche mapeamento (primeiro bit diferente -> nivel do
            ancestral)
140     // Se os graus (da raiz, filhos e netos) são [2, 3, 3]: [0, 0,
            1, 1, 1, 2, 2, 2]
141     int somaGraus = std::accumulate(grauPorNivel, grauPorNivel +
            numNiveis-1, 0);
142     nivelDosBits = new int[somaGraus];
143     int *bit = nivelDosBits, nivelAnc = 0;
144     while (grauPorNivel[nivelAnc] != 0)
145     {
146       for (int i = 0; i < grauPorNivel[nivelAnc]; i++)
147         *(bit++) = nivelAnc;
148       nivelAnc++;
149     }
150     if (DEBUG)
151     {
152       std::cout << "Nós por nível:";
153       for (int nos : std::vector<int>(nosPorNivel,
              nosPorNivel+numNiveis))
154         std::cout << ' ' << nos;
155       std::cout << '\n';
156       std::cout << "Grau por nível:";
157       for (int grau : std::vector<int>(grauPorNivel,
              grauPorNivel+numNiveis))
158         std::cout << ' ' << grau;
159       std::cout << '\n';
160       std::cout << "Somatório dos graus: " << somaGraus << '\n';
161       std::cout << "Nível dos bits:";
162       for (int nivel : std::vector<int>(nivelDosBits,
              nivelDosBits+somaGraus))
163         std::cout << ' ' << nivel;
164       std::cout << '\n';
165     }
166
167     // Se os nós ainda não têm ids, atribuir aqui... (agora têm-se
            os graus)
168     // (Árvores geradas com construirArvore() já têm os ids)
169     // TODO
170
171     // Cria um arranjo por nível com os ids e outro com os nós
172     // do nivel para descobrir os módulos e distribuir os nós
173     nosNiveis = new No*[numNiveis];
174     ID* idsNiveis[numNiveis];
175     for (int nivel = 0; nivel < numNiveis; nivel++)
176     {
177       nosNiveis[nivel] = new No[nosPorNivel[nivel]];
178       idsNiveis[nivel] = new ID[nosPorNivel[nivel]];
179     }
```

```
180     // Próxima posição a preencher de cada nível
181     int proxPos[numNiveis];
182     std::fill(proxPos, proxPos+numNiveis, 0);
183     // Pega os ids
184     preencherNosIdsNiveis(nosNiveis, idsNiveis, proxPos, raiz);
185     // Mostra ids
186     if (DEBUG)
187     {
188       int total = 0;
189       std::cout << "####### IDS DOS NÓS #######\n";
190       for (int i = 0; i < numNiveis; i++)
191       {
192         std::cout << "Nível " << i << '\n';
193         total += nosPorNivel[i];
194         if (total > DEBUG_MAX_LINHAS_IDS)
195         {
196           std::cout << "Grande demais\n";
197           break;
198         }
199         for (int j = 0; j < nosPorNivel[i]; j++)
200           std::cout << idBin(idsNiveis[i][j], somaGraus) << '\n';
201       }
202       std::cout << '\n';
203     }
204
205     // Descobrir o módulo para cada nível (menos último nível)
206     imagemNiveis = new No*[numNiveis-1];
207     for (int nivel = 0; nivel < numNiveis-1; nivel++)
208     {
209       // Níveis com grau 1 não precisam
210       if (grauPorNivel[nivel] == 1)
211       {
212         imagemNiveis[nivel] = nullptr;
213         continue;
214       }
215
216       int numNos = nosPorNivel[nivel];
217       No *nos = nosNiveis[nivel];
218       ID *ids = idsNiveis[nivel];
219       DadosFuncao *dadosNivel = &dadosFuncao[nivel];
220       // Encontra valores para a função que distribui os nós
221       buscarFuncaoIdeal(numNos, ids, dadosNivel);
222       // Coloca os nós nas posições resultantes da função
223       No *imagemNivel = new No[dadosNivel->tam] (); // Zerado
224       for (int no = 0; no < numNos; no++)
225         imagemNivel[dadosNivel->aplicar(ids[no])] = nos[no];
226       imagemNiveis[nivel] = imagemNivel;
227     }
228     // Mostra dados da função para cada nível (que tem)
229     if (DEBUG)
```

```
230     {
231       std::cout << "####### DADOS DAS FUNÇÕES #######\n";
232       for (int i = 0; i < numNiveis-1; i++)
233         if (grauPorNivel[i] > 1)
234           std::cout << "Nível " << i << '\n' << dadosFuncao[i] <<
                  '\n';
235         else
236           std::cout << "Nível " << i << "\nSEM DADOS\n";
237     }
238
239     // Libera
240     for (int nivel = 0; nivel < numNiveis; nivel++)
241       delete[] idsNiveis[nivel];
242
243     if (DEBUG)
244       std::cout << "<fim Arvore::montarEstruturas>\n";
245   }
246
247   void preencherNosGrauPorNivel(int *nosPorNivel, int
          *grauPorNivel, No no)
248   {
249     ++nosPorNivel[no->nivel];
250     grauPorNivel[no->nivel] = std::max(grauPorNivel[no->nivel],
            no->numFilhos);
251     for (int i = 0; i < no->numFilhos; i++)
252       preencherNosGrauPorNivel(nosPorNivel, grauPorNivel,
              no->filhos[i]);
253   }
254
255   void preencherNosIdsNiveis(No **nosNiveis, ID **idsNiveis, int
          *proxPos, No no)
256   {
257     // Coloca nó e id na próxima posição da linha do nível do nó
258     nosNiveis[no->nivel][proxPos[no->nivel]  ] = no;
259     idsNiveis[no->nivel][proxPos[no->nivel]++] = no->id;
260     for (int i = 0; i < no->numFilhos; i++)
261       preencherNosIdsNiveis(nosNiveis, idsNiveis, proxPos,
              no->filhos[i]);
262   }
263 };
264
265 #endif /* ARV_H */
```

arv/arv_mod.h

```
1  /*
2   * Funções para descobrir os valores de módulo ideais para uma á
        rvore.
3   */
```

```
4
5   #ifndef ARV_MOD_H
6   #define ARV_MOD_H
7
8   #include <algorithm>
9
10  #include "id.h"
11
12  /*
13   * Guarda dados da função para um nível da árvore e aplica a função.
14   */
15  struct DadosFuncao
16  {
17    /*
18     * Possivelmente, as operações OuEx e Ou em conjunto minimizam o
           m.
19     * Pode-se provar que E e Ou têm o mesmo efeito sob módulo m,
           seja qual
20     * for o m.
21     * ( Aplicar "& 0" após "| 1" é o mesmo que apenas "& 0".
22     *   Fazendo isso, subrai-se 2^bit de todo mundo, apenas
           "rodando" todos
23     *   os valores módulo m.
24     *   Portanto, "& 0" após "| 1" é o mesmo (módulo m) que apenas
           "| 1".
25     *   Logo, "& 0" tem o mesmo efeito que "| 1". )
26     * A ordem não importa: um bit pode estar ativo (1) em apenas uma
           das
27     * duas máscaras, pois não há sentido estar ativo nas duas.
28     *      (x | 1) ^ 1 == x & 0,  (x ^ 1) | 1 == x | 1
29     * Outras questões devem ser observadas com respeito ao Ou,
30     * pois a maioria das combinações de bits *garantem* colisões
           (devido
31     * a como os ids são formados).
32     * Pouco foi testado com respeito a isso ainda.
33     */
34    ID mascOuEx, mascOu, ad, m;
35    // Tamanho do arranjo necessário (pode ser < m)
36    unsigned tam;
37    // Dados para aplicar o módulo
38    // Outro método pode ser necessário caso a multiplicação
39    // seja muito cara.
40    // Usar __int128_t para multiplicações maiores?
41    ID mult, desl;
42
43    ID aplicar(ID id) const
44    {
45      ID v = ((id ^ mascOuEx | mascOu) + ad);
46      return v - (v * mult >> desl)*m; // v % m
47    }
```

```
48
49    ID testar(ID id) const
50    {
51      return (id ^ mascOuEx | mascOu) % m;
52    }
53
54    friend inline std::ostream& operator<<(std::ostream &o, const
          DadosFuncao d)
55    {
56      o << "OuEx: " << idBin(d.mascOuEx) << "\n";
57      o << "Ou:   " << idBin(d.mascOu)   << "\n";
58      o << "ad: "   << d.ad              << ", ";
59      o << "m: "    << d.m               << ", ";
60      o << "tam: "  << d.tam             << "\n";
61      o << "mult: " << d.mult            << ", ";
62      o << "desl: " << d.desl;
63
64      return o;
65    }
66  };
67
68  /*
69   * Verifica se os valores no array módulo m resultam todos em
         valores
70   * diferentes.
71   */
72  bool funcaoInjetora(int tam, ID valores[], const DadosFuncao &f)
73  {
74    // Valores já atingidos
75    bool imagem[f.m];
76    std::fill(imagem, imagem+f.m, false);
77
78    // Aplica a função a cada valor
79    for (ID *valor = valores; valor < valores+tam; valor++)
80    {
81      ID res = f.testar(*valor);
82      // Valor já foi atingido
83      if (imagem[res])
84        return false;
85      imagem[res] = true;
86    }
87
88    // Não houve colisões
89    return true;
90  }
91
92  void dadosModulo(int numNos, ID *ids, DadosFuncao *dadosFuncao);
93
94  // Buscar os valores ideais da função
95  void buscarFuncaoIdeal(int numNos, ID *ids, DadosFuncao
```

```
    *sai_dadosFuncao)
{
  // Busca função
  DadosFuncao dados;
  dados.mascOu = dados.mascOuEx = dados.ad = 0;
  dados.m = numNos-1;

  // Para fazer buscas mais elaboradas (verificar vários dados)
  bool continuar = true;
  sai_dadosFuncao->tam = ~0u; // Maior possível
  while (continuar)
  {
    // Próxima tentativa
    dados.m++;

    // A função não satisfaz
    while (!funcaoInjetora(numNos, ids, dados))
      dados.m++;

    // Imagem do módulo
    ID imagem[numNos];
    for (int i = 0; i < numNos; i++)
      imagem[i] = dados.testar(ids[i]);
    // Analisa a imagem para descobrir o melhor ad (que resulta em
        menor tam)
    // Ideia: encontrar o maior "buraco" na imagem do módulo
        ordenada e
    // "empurrá-lo" para cima, de modo que o maior valor da imagem
    // seja o menor possível
    std::sort(imagem, imagem+numNos);
    ID difMax = (dados.m+imagem[0]) - (imagem[numNos-1]);
    dados.ad = dados.m-imagem[0];
    dados.tam = imagem[numNos-1] - imagem[0] + 1;
    for (int i = 1; i < numNos; i++)
    {
      ID dif = imagem[i] - imagem[i-1];
      if (dif > difMax) // Maior resulta em tam menor
      {
        dados.ad = dados.m - imagem[i];
        dados.tam = imagem[i-1] + dados.ad + 1;
        difMax = dif;
      }
    }

    // Conseguiu tamanho melhor
    if (dados.tam < sai_dadosFuncao->tam)
      *sai_dadosFuncao = dados;


    // TODO Encontrar condição melhor
```

```
143     continuar = dados.m < numNos*numNos;
144   }
145
146   dadosModulo(numNos, ids, sai_dadosFuncao);
147 }
148
149 /* Retorna dados para calcular o módulo sem usar divisão */
150 void dadosModulo(int numNos, ID *ids, DadosFuncao *dadosFuncao)
151 {
152   // Conta bits
153   ID id = *std::max_element(ids, ids+numNos);
154   int bits = 0;
155   while (id)
156   {
157     id >>= 1;
158     bits++;
159   }
160
161   // Expansão fracionária
162   // Mais precisão pode ser necessária?
163   ID m = dadosFuncao->m;
164   double r = 1.d/m;
165   // Zeros na frente
166   int z = 0;
167   while ((int) (2*r) == 0) // Próximo bit será 0
168   {
169     r *= 2;
170     z++;
171   }
172   // Pega (bits+1) bits significativos
173   ID mult = 0;
174   for (int i = 0; i < bits+1; i++)
175   {
176     r *= 2;
177     int bit = (int) r;
178     mult = (mult << 1) + bit;
179     r -= bit;
180   }
181
182   // Arredondamento
183   mult++;
184   // Deslocamento
185   int desl = z + bits + 1;
186
187   // Atribui valores
188   dadosFuncao->mult = mult;
189   dadosFuncao->desl = desl;
190
191   // Verifica se deu certo
192   for (auto i = 0; i < numNos; i++)
```

```
193   {
194     if (mult*ids[i] >> desl != ids[i]/m)
195     {
196       dadosFuncao->mult = dadosFuncao->desl = 0;
197       break;
198     }
199   }
200 }
201
202 #endif /* ARV_MOD_H */
```

arv/config_dbg.h

```
1  #ifndef CONFIG_DBG_H
2  #define CONFIG_DBG_H
3
4  struct ConfigDebug
5  {
6    struct Arvore
7    {
8      static const bool DEBUG = false;
9      static const int DEBUG_MAX_LINHAS_IDS = 100;
10   };
11 };
12
13 #endif /* CONFIG_DBG_H */
```

arv/constr.h

```
1  #ifndef CONSTR_H
2  #define CONSTR_H
3
4  #include <vector>
5
6  ////////////////////////////////////////
7  // Constrói árvore simétrica com niveis níveis abaixo da raiz
8  // e graus[i] filhos para cada nó no nível i (nível 0 = raiz).
9  // <raiz> deve ser um nó (até então folha) válido (pai, id e nivel
       válidos)
10 // numFilhos e filhos de raiz são atribuídos.
11 static void construirSubArvore(No raiz, int niveis, int *graus, int
       soma)
12 {
13   if (niveis == 0) // Nó folha
14   {
15     raiz->numFilhos = 0;
16     raiz->filhos = NULL;
17   }
```

```
18    else
19    {
20      int grau = *graus, nivelF = raiz->nivel + 1;
21      ++graus;
22      --niveis;
23      // Cria filhos
24      No *filhos = new No[grau];
25      for (int i = 0; i < grau; i++)
26      {
27        No f = new estr_No();
28        f->id = raiz->id | (1 << (soma+i));
29        f->pai = raiz;
30        f->nivel = nivelF;
31        filhos[i] = f;
32        construirSubArvore(f, niveis, graus, soma+grau);
33      }
34      raiz->numFilhos = grau;
35      raiz->filhos = filhos;
36    }
37  }
38
39  ////////////////////////////////////////
40  // Constrói árvore simétrica
41  // niveis: Níveis *abaixo* da raiz
42  No construirArvore(int niveis, int *graus)
43  {
44    No raiz = new estr_No();
45    raiz->id = 0;
46    raiz->pai = NULL;
47    raiz->nivel = 0;
48    construirSubArvore(raiz, niveis, graus, 0);
49    return raiz;
50  }
51
52  No construirArvore(std::vector<int> graus)
53  {
54    return construirArvore(graus.size(), graus.data());
55  }
56
57  #endif /* CONSTR_H */
```

arv/id.h

```
1  #ifndef ID_H
2  #define ID_H
3
4  #include <bitset>
5  #include <climits>
6  #include <string>
```

```
7
8  typedef unsigned long ID;
9
10 std::string idBin(ID id, int bits = sizeof(ID)*CHAR_BIT)
11 {
12   std::string repr =
         std::bitset<sizeof(ID)*CHAR_BIT>(id).to_string();
13   return repr.substr(repr.size() - bits);
14 }
15
16 #endif /* ID_H */
```

arv/percorr.h

```
1  #ifndef PERCORR_H
2  #define PERCORR_H
3
4  #include "arv.h"
5
6  ////////////////////////////////////////
7  /*
8   * Encontra o ancestral comum mais próximo "subindo" a árvore.
9   */
10 No ancestralSimples(No a, No b)
11 {
12   while (a->nivel > b->nivel)
13     a = a->pai;
14   while (b->nivel > a->nivel)
15     b = b->pai;
16   while (a != b)
17   {
18     a = a->pai;
19     b = b->pai;
20   }
21   return a;
22 }
23
24 ////////////////////////////////////////
25 No ancestral(Arvore* arv, No a, No b)
26 {
27   /* Opções:
28    *  mascaras[bsf(dif)] // bit scan forward (__builtin_ctz(x))
29    *  (signed) ((dif-1) ^ dif) >> 1
30    *  ((dif-1) | dif) ^ dif
31    */
32   ID dif, bit, masc, id;
33   // Bits que diferem
34   dif = a->id ^ b->id;
35   // Primeiro diferente (usar para encontrar o array do nível?)
```

```
36    bit = dif & (-dif);
37    masc = bit - 1; // Todos bits antes do primeiro diferente
38    //masc = (((dif-1) ^ dif) >> 1);
39    // Id do ancestral comum mais próximo
40    id = a->id & masc; // a->id ou b->id, tanto faz
41
42    // Encontra o nó
43    int nivel = arv->bitParaNivel(bit);
44    No *imagemNivel = arv->imagemNiveis[nivel];
45    return imagemNivel[arv->dadosFuncao[nivel].aplicar(id)];
46  }
47
48  #endif /* PERCORR_H */
```

arv/tst_estr.cpp

```
1   /*
2    * Testes com as estruturas para encontrar o ancestral comum.
3    *
4    * TODO: Testar as estruturas em si, construídas em
         Arvore::montarEstruturas()
5    * (apenas os resultados finais e a distribuição dos nós estão
         sendo testados)
6    */
7
8   #include <algorithm>
9   #include <vector>
10
11  #include "arv.h"
12  #include "constr.h"
13  #include "percorr.h"
14
15  using namespace std;
16
17  void testarDistribuicaoNos(Arvore*);
18  void testarAncestralComum(Arvore*, vector<int>, vector<int>,
        vector<int>);
19  No descendente(No, vector<int>);
20
21  void erro(string);
22  template <class T> void testarIgual(T, T, string);
23
24  void testes();
25  int main()
26  {
27    try { testes(); }
28    catch (string &erro) { cout << erro; }
29
30    return 0;
```

```
31 }
32
33 void testes()
34 {
35   vector<int> graus = {3, 2, 2, 5};
36   int niveis, nBits;
37   niveis = graus.size() + 1;
38   nBits = std::accumulate(graus.begin(), graus.end(), 0);
39
40   No raiz = construirArvore(graus);
41   //raiz->imprimir(nBits);
42   Arvore arv(raiz, niveis);
43
44   testarDistribuicaoNos(&arv);
45   try // Exceções...?
46   {
47     try
48     {
49       Arvore arvErro(construirArvore(graus), niveis);
50       arvErro.dadosFuncao[3].ad--; // Deve dar erro
51       testarDistribuicaoNos(&arvErro);
52     }
53     catch (std::string str) { throw 0; } // Deu erro: ok
54     erro("Mutação nas estruturas deveria causar erro"); // Não deu
           erro
55   }
56   catch (int i) { cout << "Erro (mutação nas estruturas)
         corretamente detectado\n"; }
57
58   // descendente() funcionando
59   cout << "descendente() funcionando: ";
60   cout << (raiz == descendente(raiz, {})) << ' ';
61   No a = raiz->filhos[0]->filhos[1]->filhos[0];
62   cout << (a == descendente(raiz, {1, 2, 1})) << ' ';
63   No b = raiz->filhos[0]->filhos[0]->filhos[0]->filhos[3];
64   cout << (b == descendente(raiz, {1, 1, 1, 4})) << '\n';
65
66   cout << "a e b: " << a->idBin(nBits) << ' ' << b->idBin(nBits) <<
         '\n';
67   cout << ancestralSimples(a, b)->idBin(nBits) << '\n';
68   cout << ancestral(&arv, a, b)->idBin(nBits) << '\n';
69
70   testarAncestralComum(&arv, {1}, {2, 1, 4}, {1, 1, 1});
71   testarAncestralComum(&arv, {3}, {2, 1, 4}, {1});
72   testarAncestralComum(&arv, {3, 2, 1}, {4}, {3});
73   testarAncestralComum(&arv, {}, {1}, {});
74   testarAncestralComum(&arv, {}, {1}, {2});
75 }
76
77 void testarDistribuicaoNos(Arvore *a, No no)
```

```
78  {
79    // Último nível (folhas) não tem
80    if (no->numFilhos == 0)
81      return;
82
83    int nivel = no->nivel;
84    No imagem = a->imagemNiveis[nivel][
          a->dadosFuncao[nivel].aplicar(no->id) ];
85    testarIgual(no, imagem, "testarDistribuicaoNos()");
86
87    for (No filho : vector<No>(no->filhos, no->filhos +
          no->numFilhos))
88      testarDistribuicaoNos(a, filho);
89  }
90
91  void testarDistribuicaoNos(Arvore *a)
92  {
93    testarDistribuicaoNos(a, a->raiz);
94    cout << "Distribuição dos nós correta\n";
95  }
96
97  /*
98   * Verifica se as funções de busca de ancestral retornam o nó
         correto.
99   * camAnc: Caminho até o ancestral
100  * cam1, cam2: Caminho do ancestral até os descendentes
101  * cam1[0] != cam2[0]
102  */
103 void testarAncestralComum(Arvore *arv, vector<int> camAnc,
        vector<int> cam1, vector<int> cam2)
104 {
105   if (cam1.size() > 0 && cam2.size() > 0 && cam1[0] == cam2[0])
106     erro("testarAncestralComum(): caminhos dos filhos devem começar
           com valores diferentes");
107
108   No anc, desc1, desc2;
109   anc = descendente(arv->raiz, camAnc);
110   desc1 = descendente(anc, cam1);
111   desc2 = descendente(anc, cam2);
112
113   No ancSimples, ancNovo;
114   ancSimples = ancestralSimples(desc1, desc2);
115   ancNovo = ancestral(arv, desc1, desc2);
116
117   cout << "-------\n";
118   cout << "ancestral simples ok: " << (ancSimples == anc) << '\n';
119   cout << "ancestral novo    ok: " << (ancNovo == anc) << '\n';
120   cout << "-------\n";
121 }
122
```

```
123 /*
124  * descendente(raiz, {1, 2, 2}) -> 2º filho do 2º filho do 1º filho
         de raiz
125  * (1, e não 0, indica o primeiro filho).
126  */
127 No descendente(No no, vector<int> caminho)
128 {
129   for (int f : caminho)
130     if (f > no->numFilhos)
131       erro("descendente(): Caminho inválido");
132     else
133       no = no->filhos[f-1];
134   return no;
135 }
136
137
138 void erro(string msg)
139 {
140   throw (string("### ERRO ###\n") + msg +
          string("\n#############\n"));
141 }
142
143 template <class T> void testarIgual(T a, T b, string msg)
144 {
145   if (a != b)
146     erro("Teste de igualdade falhou: " + msg);
147 }
```

A.2 ESTRUTURAS (MATRIZACMP)

matriz/arv.h

```
1  #ifndef MATRIZ_H
2  #define MATRIZ_H
3
4  #include <iostream>
5  #include <bitset>
6  #include <climits>
7  #include <string>
8  #include <vector>
9
10 #include <algorithm>
11
12 #include "config_dbg.h"
13
14 namespace matriz
15 {
```

```
16  struct estr_No;
17  typedef estr_No* No;
18
19  using ID = unsigned;
20
21  struct estr_No
22  {
23    // Id usado para encontrar ancestral mais próximo
24    ID id;
25
26    No pai;
27    int numFilhos;
28    No *filhos;
29
30    // Nível do nó: 0 -> raiz
31    int nivel;
32
33    void imprimir() { imprimir(0); }
34
35    friend inline std::ostream& operator<<(std::ostream &o, const
          No n) { o << n->id; return o; } // !!!
36
37    void imprimir(int ind)
38    {
39      for (int i = 0; i < ind; i++)
40        std::cout << '|'; // Indentação
41      std::cout << "No " << id
42        /*<< " (" << nivel << ' ' << numFilhos << ')'*/ << '\n';
43      ++ind;
44      for (int f = 0; f < numFilhos; f++)
45        filhos[f]->imprimir(ind);
46    }
47
48    ~estr_No()
49    {
50      for (auto i = 0; i < numFilhos; i++)
51        delete filhos[i];
52      delete filhos;
53    }
54  };
55
56  No ancestralSimples(No a, No b);
57
58  class Arvore
59  {
60    static const bool DEBUG = ConfigDebug::Arvore::DEBUG;
61    static const int DEBUG_MAX_LINHAS_IDS =
          ConfigDebug::Arvore::DEBUG_MAX_LINHAS_IDS;
62
63    public:
```

```
64      No raiz;
65
66      // Matriz de ancestral comum
67      // ancestral[i][j], i > j
68      No **ancestral;
69
70      // Quantidade de nós da árvore inteira
71      int numNos;
72
73      Arvore(No r)
74      {
75        raiz = r;
76        montarEstruturas();
77      }
78
79      ~Arvore()
80      {
81        for (int i = 1; i < numNos; i++)
82          delete[] ancestral[i];
83        delete[] ancestral;
84      }
85
86      void montarEstruturas()
87      {
88        if (DEBUG)
89          std::cout << "<Arvore::montarEstruturas>\n";
90
91        // Lista nós
92        std::vector<No> nos;
93        pegarNos(raiz, nos);
94        numNos = nos.size();
95
96        if (DEBUG)
97          std::cout << "Número de nós: " << numNos << '\n';
98
99        // Cria a (meia) matriz
100       ancestral = new No*[numNos] ();
101       for (int i = 1; i < numNos; i++)
102         ancestral[i] = new No[i];
103
104       // Preenche a matriz
105       for (int i = 0; i < numNos; i++)
106       {
107         for (int j = i+1; j < numNos; j++)
108         {
109           No a, b;
110           a = nos[i];
111           b = nos[j];
112           ancestral[std::max(a->id, b->id)][std::min(a->id, b->id)]
                  = ancestralSimples(a, b);
```

```
113         }
114       }
115     }
116
117     void pegarNos(No no, std::vector<No> &nos)
118     {
119       nos.push_back(no);
120       for (No filho : std::vector<No>(no->filhos,
              no->filhos+no->numFilhos))
121         pegarNos(filho, nos);
122     }
123   };
124 }
125
126 #endif /* MATRIZ_H */
```

matriz/config_dbg.h

```
1  #ifndef MATRIZ_CONFIG_DBG_H
2  #define MATRIZ_CONFIG_DBG_H
3
4  namespace matriz
5  {
6    struct ConfigDebug
7    {
8      struct Arvore
9      {
10       static const bool DEBUG = false;
11       static const int DEBUG_MAX_LINHAS_IDS = 0;
12     };
13   };
14 }
15
16 #endif /* MATRIZ_CONFIG_DBG_H */
```

matriz/constr.h

```
1  #ifndef MATRIZ_CONSTR_H
2  #define MATRIZ_CONSTR_H
3
4  #include <vector>
5
6  namespace matriz
7  {
8    ////////////////////////////////////////
9    // Constrói árvore simétrica com niveis níveis abaixo da raiz
10   // e graus[i] filhos para cada nó no nível i (nível 0 = raiz).
11   // <raiz> deve ser um nó (até então folha) válido (pai, id e
         nivel válidos)
```

```
12    // numFilhos e filhos de raiz são atribuídos.
13    static void construirSubArvore(No raiz, int niveis, int *graus,
          unsigned &proxID)
14    {
15      if (niveis == 0) // Nó folha
16      {
17        raiz->numFilhos = 0;
18        raiz->filhos = nullptr;
19      }
20      else
21      {
22        int grau = *graus, nivelF = raiz->nivel + 1;
23        ++graus;
24        --niveis;
25        // Cria filhos
26        No *filhos = new No[grau];
27        for (int i = 0; i < grau; i++)
28        {
29          No f = new estr_No();
30          f->id = proxID++;
31          f->pai = raiz;
32          f->nivel = nivelF;
33          filhos[i] = f;
34          construirSubArvore(f, niveis, graus, proxID);
35        }
36        raiz->numFilhos = grau;
37        raiz->filhos = filhos;
38      }
39    }
40
41    ////////////////////////////////////////
42    // Constrói árvore simétrica
43    // niveis: Níveis *abaixo* da raiz
44    No construirArvore(int niveis, int *graus)
45    {
46      unsigned proxID = 0;
47
48      No raiz = new estr_No();
49      raiz->id = proxID++;
50      raiz->pai = NULL;
51      raiz->nivel = 0;
52      construirSubArvore(raiz, niveis, graus, proxID);
53      return raiz;
54    }
55
56    No construirArvore(std::vector<int> graus)
57    {
58      return construirArvore(graus.size(), graus.data());
59    }
60  }
```

```
61
62 #endif /* MATRIZ_CONSTR_H */
```

matriz/percorr.h

```
1  #ifndef MATRIZ_PERCORR_H
2  #define MATRIZ_PERCORR_H
3
4  #include "arv.h"
5
6  #include <algorithm>
7
8  namespace matriz
9  {
10   ////////////////////////////////////////
11   /*
12    * Encontra o ancestral comum mais próximo "subindo" a árvore.
13    */
14   No ancestralSimples(No a, No b)
15   {
16     while (a->nivel > b->nivel)
17       a = a->pai;
18     while (b->nivel > a->nivel)
19       b = b->pai;
20     while (a != b)
21     {
22       a = a->pai;
23       b = b->pai;
24     }
25     return a;
26   }
27
28   ////////////////////////////////////////
29   No ancestral(Arvore* arv, No a, No b)
30   {
31     return arv->ancestral[std::max(a->id, b->id)][std::min(a->id,
         b->id)];
32   }
33 }
34
35 #endif /* MATRIZ_PERCORR_H */
```

matriz/tst_estr.cpp

```
1  /*
2   * Testes com a matriz para encontrar o ancestral comum.
3   */
4
```

```
5  #include <algorithm>
6  #include <vector>
7
8  #include "arv.h"
9  #include "constr.h"
10 #include "percorr.h"
11
12 using namespace std;
13 using matriz::Arvore;
14 using matriz::No;
15
16 void testarAncestralComum(Arvore*, vector<int>, vector<int>,
       vector<int>);
17 No descendente(No, vector<int>);
18
19 void erro(string);
20 template <class T> void testarIgual(T, T, string);
21
22 void testes();
23 int main()
24 {
25   try { testes(); }
26   catch (string &erro) { cout << erro; }
27
28   return 0;
29 }
30
31 void testes()
32 {
33   vector<int> graus = {3, 2, 2, 5};
34   int niveis;
35   niveis = graus.size() + 1;
36
37   No raiz = matriz::construirArvore(graus);
38   //raiz->imprimir();
39   Arvore arv(raiz);
40
41   // descendente() funcionando
42   cout << "descendente() funcionando: ";
43   cout << (raiz == descendente(raiz, {})) << ' ';
44   No a = raiz->filhos[0]->filhos[1]->filhos[0];
45   cout << (a == descendente(raiz, {1, 2, 1})) << ' ';
46   No b = raiz->filhos[0]->filhos[0]->filhos[0]->filhos[3];
47   cout << (b == descendente(raiz, {1, 1, 1, 4})) << '\n';
48
49   cout << "a e b: " << a->id << ' ' << b->id << '\n';
50   cout << ancestralSimples(a, b)->id << '\n';
51   cout << ancestral(&arv, a, b)->id << '\n';
52
53   testarAncestralComum(&arv, {1}, {2, 1, 4}, {1, 1, 1});
```

```
54    testarAncestralComum(&arv, {3}, {2, 1, 4}, {1});
55    testarAncestralComum(&arv, {3, 2, 1}, {4}, {3});
56    testarAncestralComum(&arv, {}, {1}, {});
57    testarAncestralComum(&arv, {}, {1}, {2});
58  }
59
60  /*
61   * Verifica se as funções de busca de ancestral retornam o nó
         correto.
62   * camAnc: Caminho até o ancestral
63   * cam1, cam2: Caminho do ancestral até os descendentes
64   * cam1[0] != cam2[0]
65   */
66  void testarAncestralComum(Arvore *arv, vector<int> camAnc,
       vector<int> cam1, vector<int> cam2)
67  {
68    if (cam1.size() > 0 && cam2.size() > 0 && cam1[0] == cam2[0])
69      erro("testarAncestralComum(): caminhos dos filhos devem começar
           com valores diferentes");
70
71    No anc, desc1, desc2;
72    anc = descendente(arv->raiz, camAnc);
73    desc1 = descendente(anc, cam1);
74    desc2 = descendente(anc, cam2);
75
76    No ancSimples, ancNovo;
77    ancSimples = ancestralSimples(desc1, desc2);
78    ancNovo = ancestral(arv, desc1, desc2);
79
80    cout << "-------\n";
81    cout << "ancestral simples ok: " << (ancSimples == anc) << '\n';
82    cout << "ancestral matriz  ok: " << (ancNovo == anc) << '\n';
83    cout << "-------\n";
84  }
85
86  /*
87   * descendente(raiz, {1, 2, 2}) -> 2º filho do 2º filho do 1º filho
         de raiz
88   * (1, e não 0, indica o primeiro filho).
89   */
90  No descendente(No no, vector<int> caminho)
91  {
92    for (int f : caminho)
93      if (f > no->numFilhos)
94        erro("descendente(): Caminho inválido");
95      else
96        no = no->filhos[f-1];
97    return no;
98  }
99
```

```
100
101 void erro(string msg)
102 {
103   throw (string("### ERRO ###\n") + msg +
          string("\n#############\n"));
104 }
105
106 template <class T> void testarIgual(T a, T b, string msg)
107 {
108   if (a != b)
109     erro("Teste de igualdade falhou: " + msg);
110 }
```

A.3 TESTES DE DESEMPENHO

tst_tmp/tst_tmp.cpp

```
1  /*
2   * Testes de desempenho temporal das funções de ancestral comum
3   */
4
5  #include <cstdlib>
6  #include <fstream>
7  #include <iostream>
8  #include <string>
9  #include <vector>
10
11 #include "tst_tmp.h"
12 #include "tst_tmp_arv.h"
13 #include "tst_tmp_hwloc.h"
14 #include "tst_tmp_matriz.h"
15
16 using namespace std;
17
18 using matriz::TesteAncestralComumMatriz;
19
20 void executarTestes(
21   const vector<int> &graus,
22   int iter, int iterAquec,
23   int vezesFora = 1, int vezesDentro = 1);
24
25 template <class A, class N>
26 void testar(BaseTesteAncestralComum<A, N> &tst, const vector<int>
       &graus, int vezes = 1);
27
28 const int ALG_SIMPLES  = 0,
29           ALG_NOVO     = 1,
```

```
30            ALG_HWLOC    = 2,
31            ALG_MATRIZ   = 3,
32            ALG_OVERHEAD = 4,
33            NUM_ALGS     = 5;
34  bool alg_ativo[NUM_ALGS] = {};
35  vector<int> algs_ativos;
36  string nomes[NUM_ALGS] =
37  {
38    "Simples",
39    "Novo",
40    "Hwloc",
41    "Matriz",
42    "Overhead"
43  };
44  int tamMaxNome = max_element(nomes, nomes+NUM_ALGS,
45    [](string &a, string &b) { return a.size() < b.size(); })->size();
46  vector<double> tempos[NUM_ALGS];
47
48  int main(int argc, char *argv[])
49  {
50    // Graus de https://www.open-mpi.org/
          projects/hwloc/lstopo/images/16XeonX7400.v1.11.png
51    vector<int> graus({4, 4, 1, 3, 2, 1, 1, 1});
52    string maquina;
53    bool suprimir = false;
54    int vezesFora = 2, vezesDentro = 3, iteracoes = 10000,
          aquecimento = 1000;
55    alg_ativo[ALG_SIMPLES] = true;
56    alg_ativo[ALG_NOVO   ] = true;
57    alg_ativo[ALG_HWLOC  ] = true;
58
59    int i = 1;
60    string arg;
61    while (i < argc)
62    {
63      arg = argv[i++];
64      string resto = arg.substr(1);
65      switch (arg.front())
66      {
67        case 's': suprimir = true;           break; // Suprimir saída
              para arquivo
68        case 'M': maquina = resto;           break; // Identificador
              da máquina rodando os testes
69        case 'i': iteracoes   = stoi(resto); break; // Iterações
70        case 'a': aquecimento = stoi(resto); break; // Aquecimento
71        case 'r': vezesDentro = stoi(resto); break; // Repetições de
              cada algoritmo
72        case 'R': vezesFora   = stoi(resto); break; // Repetições de
              tudo
73        case 'A': // Algoritmos
```

```
74          for (bool &b : alg_ativo) b = false;
75          for (char &alg : resto)
76            switch (alg)
77            {
78              case 's': alg_ativo[ALG_SIMPLES ] = true; break; //
                    Simples
79              case 'n': alg_ativo[ALG_NOVO    ] = true; break; // Novo
80              case 'h': alg_ativo[ALG_HWLOC   ] = true; break; //
                    Hwloc
81              case 'm': alg_ativo[ALG_MATRIZ  ] = true; break; //
                    Matriz
82              case 'o': alg_ativo[ALG_OVERHEAD] = true; break; //
                    Overhead
83              case '*': for (bool &b : alg_ativo) b = true; break; //
                    Todos
84              default: cerr << "Algoritmo não reconhecido: " << alg
                    << '\n';
85            }
86          break;
87        case '-': // Graus
88          graus = vector<int>();
89          while (i < argc)
90            graus.push_back(stoi(argv[i++]));
91          break;
92        default:
93          cerr << "Argumento não reconhecido: " + arg << '\n';
94      }
95    }
96    for (int alg = 0; alg < NUM_ALGS; alg++)
97      if (alg_ativo[alg])
98        algs_ativos.push_back(alg);
99    if (algs_ativos.empty())
100   {
101     cerr << "Nenhum algoritmo especificado\n";
102     return 0;
103   }
104
105   executarTestes(graus, iteracoes, aquecimento, vezesFora,
          vezesDentro);
106
107   if (suprimir)
108     return 0;
109
110   // Escreve em um arquivo em uma pasta definida pela configuração
111   string sep, pasta, iniNome, iniCmd;
112   #ifdef _WIN32
113     sep = "\\";
114     iniCmd = "mkdir res";
115   #else
116     sep = "/";
```

```
117     iniCmd = "mkdir -p res";
118   #endif
119
120   // Define nome da pasta
121   if (!maquina.empty())
122     pasta += maquina + sep;
123   pasta += 'i' + to_string(iteracoes);
124   pasta += 'a' + to_string(aquecimento);
125   pasta += '-';
126   for (const int &g : graus)
127     pasta += to_string(g) + '.';
128   pasta.pop_back();
129
130   // Cria a pasta
131   system((iniCmd + sep + pasta).data());
132
133   // Descobre nome não usado
134   iniNome = "res" + sep + pasta + sep + "resultados";
135   int n = -1;
136   while (ifstream(iniNome + to_string(++n) + ".csv")); // Existe
137   // Cria o arquivo
138   ofstream saida(iniNome + to_string(n) + ".csv");
139   if (!saida)
140   {
141     cerr << "Impossível criar o arquivo\n";
142     return 0;
143   }
144
145   // Escreve no arquivo
146   int ult_alg = algs_ativos.back();
147   algs_ativos.pop_back();
148   // Cabeçalho
149   for (const int &alg : algs_ativos)
150     saida << nomes[alg] << ',';
151   saida << nomes[ult_alg] << '\n';
152   // Amostras
153   int numAmostras = vezesFora*vezesDentro;
154   for (int i = 0; i < numAmostras; i++)
155   {
156     for (const int &alg : algs_ativos)
157       saida << tempos[alg][i] << ',';
158     saida << tempos[ult_alg][i] << '\n';
159   }
160
161   cout << "Arquivo \"" << iniNome + to_string(n) + ".csv" << "\"
         criado\n";
162 }
163
164 template <class A, class N>
```

```
165 void testar(BaseTesteAncestralComum<A, N> &tst, int alg, const
        vector<int> &graus, int vezes)
166 {
167   if (!alg_ativo[alg])
168     return;
169
170   vector<double> resultados;
171   for (auto i = 0; i < vezes; i++)
172   {
173     double t = tst.executar(graus);
174     resultados.push_back(t);
175     tempos[alg].push_back(t);
176   }
177
178   string &nome = nomes[alg];
179   cout << nome << ": ";
180   for (int i = 0; i < tamMaxNome-nome.size(); i++)
181     cout << ' ';
182   for (auto t : resultados)
183     cout << " (" << t << " ms)";
184   cout << '\n';
185 }
186
187 void executarTestes(
188   const vector<int> &graus,
189   int iter, int iterAquec,
190   int vezesFora, int vezesDentro)
191 {
192   TesteAncestralComumSimples  simples (iter, iterAquec);
193   TesteAncestralComumNovo     novo    (iter, iterAquec);
194   TesteAncestralComumHwloc    hwloc   (iter, iterAquec);
195   TesteAncestralComumMatriz   matriz  (iter, iterAquec);
196   TesteAncestralComumOverhead overhead(iter, iterAquec);
197   for (auto i = 0; i < vezesFora; i++)
198   {
199     testar(simples , ALG_SIMPLES , graus, vezesDentro);
200     testar(novo    , ALG_NOVO    , graus, vezesDentro);
201     testar(hwloc   , ALG_HWLOC   , graus, vezesDentro);
202     testar(matriz  , ALG_MATRIZ  , graus, vezesDentro);
203     testar(overhead, ALG_OVERHEAD, graus, vezesDentro);
204     cout << '\n';
205   }
206 }
```

tst_tmp/tst_tmp.h

```
1 #ifndef TST_TMP_H
2 #define TST_TMP_H
3
```

```
4  #include <algorithm>
5  #include <chrono>
6  #include <random>
7  #include <vector>
8
9  template <class Arv, class No>
10 class BaseTesteAncestralComum
11 {
12   using Par = std::pair<No, No>;
13
14   int numIter, numIterAquec;
15
16   public:
17
18   BaseTesteAncestralComum(int nI, int nIA) : numIter(nI),
         numIterAquec(nIA) {}
19
20   virtual Arv criarArvore(std::vector<int> graus);
21   virtual std::vector<No> pegarFolhas(Arv a);
22   virtual No ancestralComum(Arv arv, No a, No b);
23   virtual int acessar(No n); // Acessa qualquer coisa no nó
24   virtual void destruirArvore(Arv a);
25
26   // Tempo em milissegundos
27   double executar(std::vector<int> graus)
28   {
29     Arv a = criarArvore(graus);
30
31     std::vector<No> fs = pegarFolhas(a);
32     std::vector<Par> pares = gerarPares(fs);
33     embaralhar(pares);
34
35     // Acumulador (evita otimizações)
36     int acumulador = 0;
37     volatile int saida;
38
39     // Aquecer
40     for (int i = 0; i < numIterAquec; i++)
41       for (Par par : pares)
42         acumulador += acessar(ancestralComum(a, par.first,
               par.second));
43
44     // Mede o tempo
45     using namespace std::chrono;
46     using relogio = high_resolution_clock;
47     using milisDouble = duration<double, std::milli>;
48     relogio::time_point t0 = relogio::now();
49
50     // Roda
51     for (int i = 0; i < numIter; i++)
```

```
52        for (Par par : pares)
53          acumulador += acessar(ancestralComum(a, par.first,
                par.second));
54
55      // Mede o tempo
56      relogio::time_point t1 = relogio::now();
57
58      saida = acumulador;
59      destruirArvore(a);
60
61      double tempo = duration_cast<milisDouble>(t1 - t0).count();
62      return tempo;
63    }
64
65    private:
66
67    std::vector<Par> gerarPares(std::vector<No> nos)
68    {
69      int tam = nos.size();
70      std::vector<Par> pares;
71      for (int i = 0; i < tam-1; i++)
72        for (int j = i+1; j < tam; j++)
73          pares.push_back({nos[i], nos[j]});
74      return pares;
75    }
76
77    void embaralhar(std::vector<Par> pares)
78    {
79      std::shuffle(pares.begin(), pares.end(),
              std::default_random_engine());
80    }
81  };
82
83  #endif /* TST_TMP_H */
```

tst_tmp/tst_tmp_arv.h

```
1   /*
2    * Classes para testes de tempo de ancestral comum usando as
         estruturas em arv.h
3    */
4
5   #ifndef TST_TMP_ARV_H
6   #define TST_TMP_ARV_H
7
8   #include "tst_tmp.h"
9   #include "../arv/arv.h"
10  #include "../arv/constr.h"
11  #include "../arv/percorr.h"
```

```
12
13 // Teste usando Arvore e No
14 class TesteAncestralComumArvore : public
       BaseTesteAncestralComum<Arvore*, No>
15 {
16   public:
17
18   TesteAncestralComumArvore(int nI, int nIA) :
         BaseTesteAncestralComum(nI, nIA) {};
19
20   Arvore* criarArvore(std::vector<int> graus)
21   {
22     return new Arvore(construirArvore(graus), graus.size()+1);
23   }
24
25   std::vector<No> pegarFolhas(Arvore *a)
26   {
27     int ultNivel = a->numNiveis - 1;
28     No *folhas = a->nosNiveis[ultNivel];
29     return std::vector<No>(folhas, folhas +
           a->nosPorNivel[ultNivel]);
30   }
31
32   int acessar(No n)
33   {
34     return int(n->id);
35   }
36
37   void destruirArvore(Arvore *a)
38   {
39     delete a;
40   }
41 };
42
43 // Teste usando ancestral comum simples
44 class TesteAncestralComumSimples : public TesteAncestralComumArvore
45 {
46   public:
47
48   TesteAncestralComumSimples(int nI, int nIA) :
         TesteAncestralComumArvore(nI, nIA) {};
49
50   No ancestralComum(Arvore *arv, No a, No b)
51   {
52     return ancestralSimples(a, b);
53   }
54 };
55
56 // Teste usando ancestral comum novo (usando as estruturas)
57 class TesteAncestralComumNovo : public TesteAncestralComumArvore
```

```
58 {
59   public:
60
61   TesteAncestralComumNovo(int nI, int nIA) :
         TesteAncestralComumArvore(nI, nIA) {};
62
63   No ancestralComum(Arvore *arv, No a, No b)
64   {
65     return ancestral(arv, a, b);
66   }
67 };
68
69 // Teste que não usa nada: Mede o overhead do loop
70 class TesteAncestralComumOverhead : public TesteAncestralComumArvore
71 {
72   public:
73
74   TesteAncestralComumOverhead(int nI, int nIA) :
         TesteAncestralComumArvore(nI, nIA) {};
75
76   No ancestralComum(Arvore *arv, No a, No b)
77   {
78     return nullptr;
79   }
80
81   int acessar(No n)
82   {
83     return 0;
84   }
85 };
86
87 #endif /* TST_TMP_ARV_H */
```

tst_tmp/tst_tmp_hwloc.h

```
1  /*
2   * Classe para teste de tempo de ancestral comum usando hwloc
3   */
4
5  #ifndef TST_TMP_HWLOC_H
6  #define TST_TMP_HWLOC_H
7
8  #include "tst_tmp.h"
9  #include <hwloc.h>
10 #include <string>
11
12 // Teste usando hwloc
13 class TesteAncestralComumHwloc : public
       BaseTesteAncestralComum<hwloc_topology_t, hwloc_obj_t>
```

```
14 {
15   public:
16
17   TesteAncestralComumHwloc(int nI, int nIA) :
         BaseTesteAncestralComum(nI, nIA) {};
18
19   hwloc_topology_t criarArvore(std::vector<int> graus)
20   {
21     // Descrição da topologia sintética
22     std::string s;
23     int ultimo = graus.back();
24     graus.pop_back();
25     for (int g : graus)
26       s += "ca:" + std::to_string(g) + ' '; // Caches evitam
             "compactação"
27     s += std::to_string(ultimo);
28
29     // Cria a topologia
30     hwloc_topology_t t;
31     hwloc_topology_init(&t);
32     hwloc_topology_set_synthetic(t, s.data());
33     hwloc_topology_load(t);
34
35     return t;
36   }
37
38   std::vector<hwloc_obj_t> pegarFolhas(hwloc_topology_t t)
39   {
40     int ultNivel = hwloc_topology_get_depth(t) - 1;
41     std::vector<hwloc_obj_t> folhas;
42     hwloc_obj_t folha = hwloc_get_obj_by_depth(t, ultNivel, 0);
43     while (folha != nullptr)
44     {
45       folhas.push_back(folha);
46       folha = hwloc_get_next_obj_by_depth(t, ultNivel, folha);
47     }
48     return folhas;
49   }
50
51   hwloc_obj_t ancestralComum(hwloc_topology_t t, hwloc_obj_t a,
         hwloc_obj_t b)
52   {
53     return hwloc_get_common_ancestor_obj(t, a, b);
54   }
55
56   int acessar(hwloc_obj_t n)
57   {
58     return int(n->depth);
59   }
60
```

```
61    void destruirArvore(hwloc_topology_t t)
62    {
63      hwloc_topology_destroy(t);
64    }
65  };
66
67  #endif /* TST_TMP_HWLOC_H */
```

tst_tmp/tst_tmp_matriz.h

```
1   /*
2    * Classes para testes de tempo de ancestral comum usando as
         estruturas em arv.h
3    */
4
5   #ifndef TST_TMP_MATRIZ_H
6   #define TST_TMP_MATRIZ_H
7
8   #include "tst_tmp.h"
9   #include "../matriz/arv.h"
10  #include "../matriz/constr.h"
11  #include "../matriz/percorr.h"
12
13  namespace matriz
14  {
15    // Teste usando matriz Arvore e No
16    class TesteAncestralComumMatriz : public
          BaseTesteAncestralComum<Arvore*, No>
17    {
18      public:
19
20      TesteAncestralComumMatriz(int nI, int nIA) :
            BaseTesteAncestralComum(nI, nIA) {};
21
22      Arvore* criarArvore(std::vector<int> graus)
23      {
24        return new Arvore(construirArvore(graus));
25      }
26
27      std::vector<No> pegarFolhas(Arvore *a)
28      {
29        std::vector<No> vetorFolhas;
30        pegarFolhas(a->raiz, vetorFolhas);
31        No *folhas = vetorFolhas.data();
32        return std::vector<No>(folhas, folhas + vetorFolhas.size());
33      }
34
35      void pegarFolhas(No no, std::vector<No> &nos)
36      {
```

```
37        if (no->numFilhos == 0)
38          nos.push_back(no);
39        else
40          for (No filho : std::vector<No>(no->filhos,
                no->filhos+no->numFilhos))
41            pegarFolhas(filho, nos);
42      }
43
44      No ancestralComum(Arvore *arv, No a, No b)
45      {
46        return ancestral(arv, a, b);
47      }
48
49      int acessar(No n)
50      {
51        return int(n->id);
52      }
53
54      void destruirArvore(Arvore *a)
55      {
56        delete a;
57      }
58    };
59  }
60
61  #endif /* TST_TMP_MATRIZ_H */
```