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 máquinas 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 ??? 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. No nível mais alto dessas hierarquias estão as unidades de processamento, acessando e operando sobre os dados em memória. Quanto mais próximo for o nível de memória em que esse 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 na 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 **numa!** (**numa!**), em que mais níveis são acrescentados à hierarquia, compondo a rede de interconexão. Esta rede, por si só, também 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. Suas topologias são as mais variadas, visando atender às necessidades de várias classes de aplicações com características e comportamentos distintos. 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 a memória é composta de várias partes que estão diretamente ligadas a nós distintos, de modo que diferentes regiões da memória possuem diferentes tempos de acesso, 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 **mpi!** (**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.

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 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 **ecl!** (**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 **??** constam o progresso atual do trabalho e o planejamento do seu prosseguimento.

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.

2.1 MEMÓRIAS CACHE

Caches são memórias com o próposito de diminuir o tempo médio de acesso a outros níveis de memória. 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*

numa! (**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, **uma!** (**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.

O modo como os nós 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).

O hwloc é um pacote de software amplamente utilizado para a representação de topologias de máquina com 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.

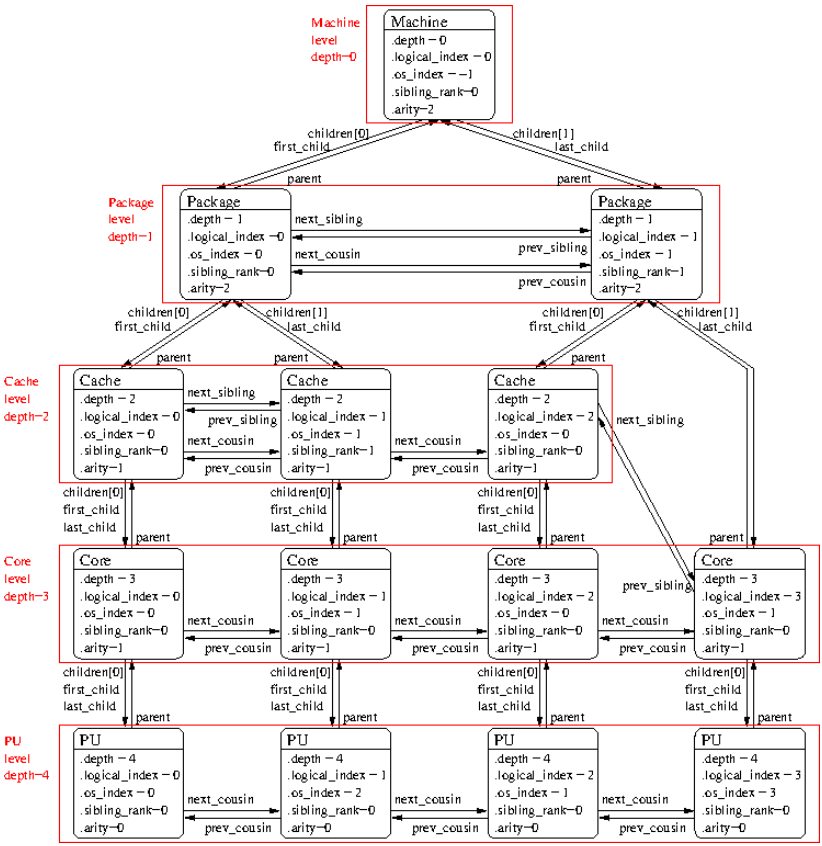
3.1 OBJETOS

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

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

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



profundidade não é exatamente o usado no contexto de estruturas de árvore, sendo possível irmãos terem profundidades diferentes.

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 array de arrays de objetos, semelhante a uma matriz, mas com arrays

internos de tamanho variável. Cada um dos arrays internos contém os objetos de um nível, e eles são organizados no array 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

Diversas funções podem ser utilizadas para acessar os objetos da hierarquia e informações sobre eles. Alguns exemplos de funções são:

- `hwloc_get_obj_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 dado objeto que seja de um determinado tipo.

- `hwloc_get_ancestor_obj_by_depth`: Semelhante à função anterior, procurando por nível em vez de tipo.

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 como o projeto hwloc foi analisado, as considerações feitas com relação ao desempenho e as implementações resultantes. Todo o código foi desenvolvido na linguagem C++.

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. !!! Listar funções !!!

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 ($O(altura)$), que é o mesmo que $O(\log N)$, onde N é a quantidade de nós da árvore. Além disso, foi analisado um projeto de código aberto que utiliza o hwloc, 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. 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.

Diante dessas considerações, 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. Existem algumas situações especiais. 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 utiliza apenas a estrutura de árvore em si, sem adição de outras informações, será referido como método simples. 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. Um ponto que pode afetar o desempenho deste algoritmo 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 ACMP do hwloc) !!!

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 do simples. Podemos considerar a seguinte ideia para encontrar outra maneira de implementar a função de ancestral comum: 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, essa função ACMP_{IDs} deveria ser *processada* em tempo constante. Ou seja, é necessário encontrar um algoritmo que seja executado em tempo constante nos processadores modernos. 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. !!! (? Isso pode ser visto... (exemplo à frente com matriz?)) !!!

Com isso em mente, podemos definir tal função usando as operações básicas encontradas no conjunto de instruções das arquiteturas atuais, tais como as operações aritméticas e operações lógicas bit-a-bit.

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 (método simples) e de uma matriz (método matricial). 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 1. 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 1: MatrizACMP

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

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

4.1.3 Função de espalhamento

Outra possibilidade foi idealizada, dividindo a função $\mathrm{ACMP}_{\mathrm{IDs}}$ 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 é formado por um ou mais bits seguidos do ID do seu pai, de modo que, dados dois nós a e b, b descendente de a, os bits menos significativos de b são iguais ao ID de a. Desse modo, dados dois descendentes de um nó c, todos os bits menos significativos deles que coincidem (todos os que vêm antes do primeiro que difere)

são iguais ao ID de c (ignorando zeros à esquerda). Usando apenas as operações que podem ser vistas nas linhas 1 – 4 do Algoritmo 2 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.

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,

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

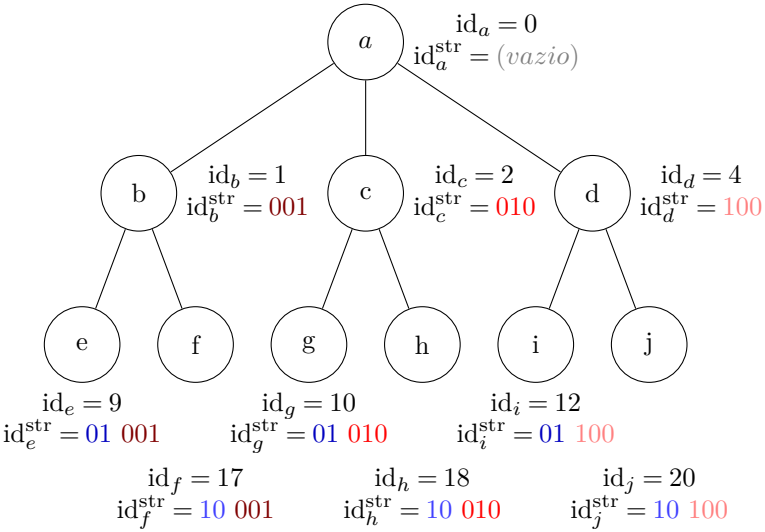
  onde $\parallel$ é a concatenação e x é uma cadeia cujo tamanho é o grau do nível de $\mathrm{pai}(a)$. Se a é o i-ésimo filho de seu pai, x possui 1 na i-ésima posição da direta para a esquerda e 0 nas demais.

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. O !!! Algoritmo X !!! mostra como o ID do ACMP é obtido a partir do ID de dois nós.

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. !!! Exemplo !!!

- 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. !!! Exemplo !!! Este foi o método escolhido.

As linhas 5 – 7 do Algoritmo 2 apresentam como este método funciona. Quanto à implementação da função BitParaNível, os processadores 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.

Algoritmo 2: 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 3) 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)
```

Se os IDs têm até b bits, esse arranjo precisar ter b posições. A função Espalhamento (Algoritmo 3) 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..., 2002), e com o resultado da divisão pode-se calcular o módulo. Esta otimizações são usadas, por exemplo, por compiladores. Aqui, no entanto, a ideia não é realizá-las em tempo de compilação, mas quando

Algoritmo 3: ESPALHAMENTO – Função de espalhamento

Entrada: Um nível (inteiro não negativo) n
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 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 $\mathrm{v} - ((\mathrm{v} * \mathrm{mult}_n)$ $\mathbf{Desl_{direita}}$ $\mathrm{desl}_n) * m_n$

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 id mod m, minimizando o tamanho necessário do arranjo. 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.

!!! Pode falhar se os dois nós de entrada são, na verdade, o mesmo nó. If resolve Funciona em casos de hierarquias assimétricas, simples não Limitação: Quantidade de bits necessária - Solução parcial: 'Esconder' níveis cujos nós têm só um filho (níveis de grau um). !!!

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. Foi medido o tempo tomado pelos algoritmos ao se encontrar repetidamente o ACMP entre os nós folhas de uma árvore. Com esses dados, analisados

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, para as quais o programa lstopo gera as representações de hierarquia apresentadas na Figura 3. Estas representações revelam o tamanho das memórias cache, que podem ter efeito nos resultados dos testes. A Tabela 2 apresenta mais detalhes de ambas.

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 versão do GCC era a única disponível no Cygwin durante o desenvolvimento do trabalho com a qual não foram encontrados problemas com

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

funcionalidades da linguagem C++. Foram usadas as seguintes *flags* de compilação:

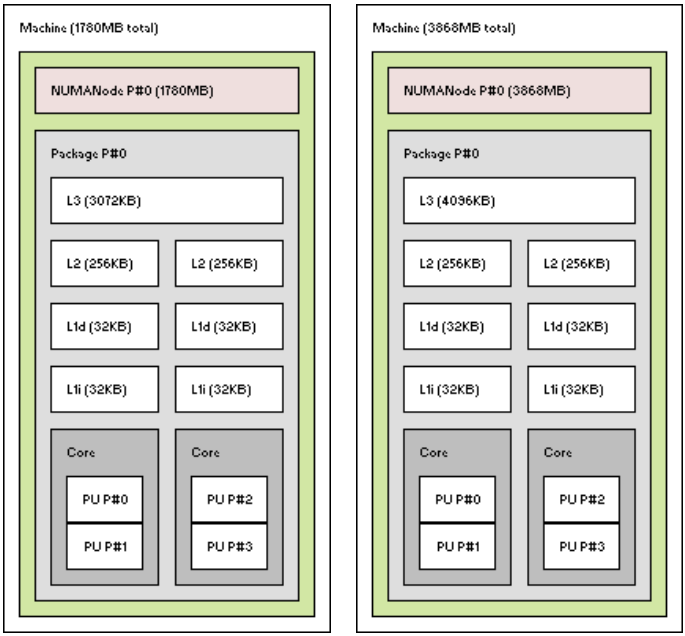
- `-std=c++11`: Utiliza funcionalidades do padrão C++11 da linguagem C++
- `-O3`: Ativa diversas otimizações
- *flags* obtidas com os comandos `pkg-config -cflags hwloc` e `pkg-config -libs hwloc` no Cygwin, que imprimem as *flags* necessárias para utilizar o hwloc

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 – o existente no hwloc e os implementados. 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 ACMPentre 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.

Figura 3 – Saída do programa lstopo sobre as máquinas utilizadas

(a) Máquina A (b) Máquina B



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. Os possíveis são o simples, o que utiliza as novas estruturas, o que utiliza uma matriz e o implementado no hwloc.

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.

!!! especificar parâmetros utilizados !!!

Notou-se a presença de diversos *outliers* de valor significativa-

mente 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.

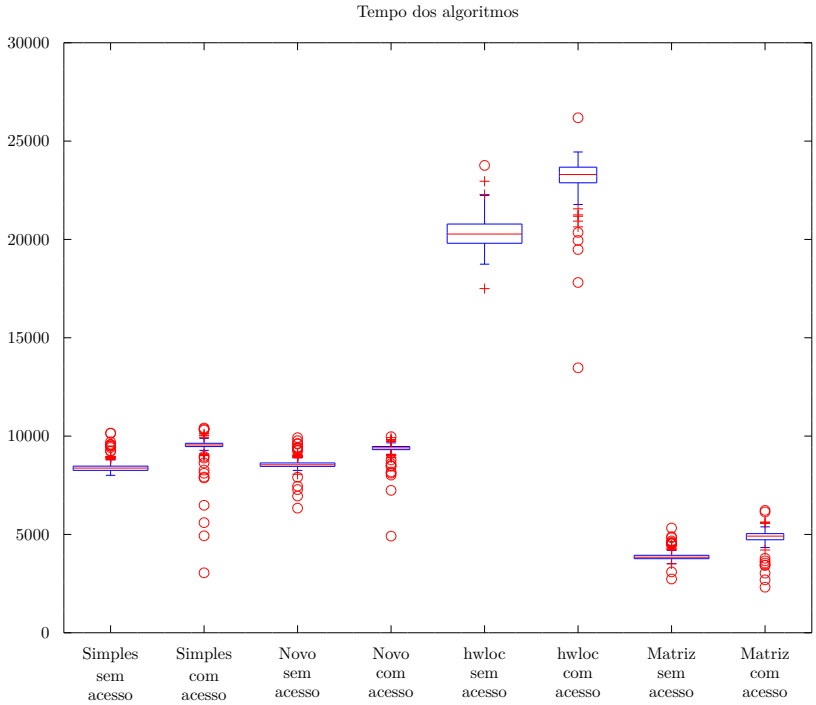
Inicialmente, os testes foram feitos sem acessar o ACMP obtido a cada execução, ou seja, para cada par de folhas, simplesmente se encontrava o ponteiro para o ancestral, mas nenhum dado do ancestral era obtido. No entanto, em cenários reais, os nós seriam buscados para se obter alguma informação sobre eles, portanto, foi adicionado um acesso a um valor qualquer de cada ACMP encontrado para simular isso. A Figura 4 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 3.

Tabela 3 – Aumento (%) do tempo mediano

	Simples	Novo	hwloc	Matriz
Mediana sem acesso	8373.99	8540.89	20274.45	3841.80
Mediana com acesso	9549.51	9401.76	23300.20	4914.60
Aumento (%)	14,04	10,08	14,92	27,92

!!! Outras análises: comparações entre as máquinas, comparar tempo médio por acesso do novo algoritmo entre topologias de tamanhos diferentes !!!

Figura 4 – 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.

REFERÊNCIAS

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>.

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

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.. ed. [S.l.]: Morgan Kaufmann, 2011.

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.