Universidade Estadual de Campinas
Instituto de Computação

Guilherme Guaglianoni Piccoli

Técnicas de compilação para apoiar a migração de dados em sistemas NUMA

CAMPINAS
2016

Guilherme Guaglianoni Piccoli

Técnicas de compilação para apoiar a migração de dados em sistemas NUMA

Dissertação apresentada ao Instituto de Computação da Universidade Estadual de Campinas como parte dos requisitos para a obtenção do título de Mestre em Ciência da Computação.

Orientador: Prof. Dr. Edson Borin
Coorientador: Prof. Dr. Fernando Magno Quintão Pereira

Este exemplar corresponde à versão final da Dissertação defendida por Guilherme Guaglianoni Piccoli e orientada pelo Prof. Dr. Edson Borin.

CAMPINAS
2016

Agência(s) de fomento e nº(s) de processo(s): FAPESP, 2013/18794-3

Ficha catalográfica
Universidade Estadual de Campinas
Biblioteca do Instituto de Matemática, Estatística e Computação Científica
Maria Fabiana Bezerra Muller - CRB 8/6162

P581t

Piccoli, Guilherme Guaglianoni, 1988-
Técnicas de compilação para apoiar a migração de dados em sistemas NUMA / Guilherme Guaglianoni Piccoli. – Campinas, SP : [s.n.], 2016.

Orientador: Edson Borin.
Coorientador: Fernando Magno Quintão Pereira.
Dissertação (mestrado) – Universidade Estadual de Campinas, Instituto de Computação.

1. Compiladores (Computadores). 2. Sistemas de computação. 3. Gerenciamento de memória (Computação). I. Borin, Edson,1979-. II. Pereira, Fernando Magno Quintão. III. Universidade Estadual de Campinas. Instituto de Computação. IV. Título.

Informações para Biblioteca Digital

Título em outro idioma: Compilation techniques to support memory migration on NUMA systems
Palavras-chave em inglês:
Compiling (Electronic computers)
Computing systems
Memory management (Computer science)
Área de concentração: Ciência da Computação
Titulação: Mestre em Ciência da Computação
Banca examinadora:
Edson Borin [Orientador]
Philippe Olivier Alexandre Navaux
Lucas Francisco Wanner
Data de defesa: 18-04-2016
Programa de Pós-Graduação: Ciência da Computação

Universidade Estadual de Campinas
Instituto de Computação

Guilherme Guaglianoni Piccoli

Técnicas de compilação para apoiar a migração de dados em sistemas NUMA

Banca Examinadora:

- Prof. Dr. Edson Borin
  Universidade Estadual de Campinas
- Prof. Dr. Philippe Olivier Alexandre Navaux
  Universidade Federal do Rio Grande do Sul
- Prof. Dr. Lucas Francisco Wanner
  Universidade Estadual de Campinas

A ata da defesa com as respectivas assinaturas dos membros da banca encontra-se no processo de vida acadêmica do aluno.

Campinas, 18 de abril de 2016

Agradecimentos

Em primeiro lugar, gostaria de agradecer aos meus pais, Piccoli e Ivana, que permitiram que eu estudasse até agora, e deram total apoio e ajuda nos momentos de dificuldade. Sem eles, definitivamente eu não teria conseguido. Ainda, minha noiva Diliane, que ajudou tanto na leitura do texto quanto com uns "puxões de orelha" quando eu estava desanimado com o trabalho - sem o apoio dela, não teria conseguido concluir esse texto.

Ao meu orientador prof. Edson Borin gostaria de agradecer muito, pois sua compreensão e apoio foram fundamentais ao desenvolvimento do mestrado. Também, meu co-orientador prof. Fernando Magno Quintão Pereira com sua atitude positiva foi de fundamental importância no decorrer desse trabalho.

Aos amigos do laboratório Alisson (*aka* Anderson), Gilvan e Renan, cujo apoio foi de grande ajuda, gostaria de deixar meu agradecimento. Da Unicamp em geral, o Lucas Tadeu e o Raul Baldin foram grandes amigos em vários momentos, e a eles também agradeço pelo apoio. Da vida antes do mestrado, meus dois grandes amigos Juliano e Rafael, que sempre me apoiaram em tudo merecem aparecer aqui pelos grandes momentos de sempre. E da vida "pós" mestrado, gostaria de deixar o meu agradecimento a todos do *Linux Technology Center* (IBM), que têm feito meu dia-a-dia divertido e desafiador.

Durante esse trabalho tive uma colaboração imensa e fundamental do Henrique Santos, aluno do prof. Fernando na UFMG, e detentor de um conhecimento admirável do LLVM e de computação em geral - a ele, deixo um profundo agradecimento pela boa-vontade de ajudar sempre que precisei.

Às equipes dos laboratórios LabMec e LMCAD, bem como ao time da CPG do Instituto de Computação, agradeço pela ajuda quando precisei e pela infraestrutura oferecida para a execução desse trabalho. Ao prof. Philippe Devloo, agradeço pela generosa cessão da máquina utilizada nos experimentos.

Finalmente, mas não menos importante, gostaria de agradecer à FAPESP (processo 2013/18794-3) pelo auxílio financeiro, fundamental na execução desse trabalho, e também à Capes que forneceu bolsa essencial no início do mestrado.

Resumo

Com o avanço cada vez maior dos processadores de múltiplos núcleos, devido especialmente à barreira tecnológica imposta por limitações físicas no crescimento da frequência de operação dos processadores, arquiteturas de memória não-uniforme (NUMA) estão se difundindo como solução para a escalabilidade de projetos de alto desempenho computacional. Tais arquiteturas não são isentas de problemas, especialmente em se tratando do acesso à memória; má alocação de memória pode causar contenção no acesso aos dados e gerar redução significativa no desempenho de aplicações. Neste trabalho apresentamos a técnica *Selective Page Migration* (SPM), uma otimização no âmbito de compiladores que, através da análise de laços e suas propriedades, e dos vetores acessados dentro de tais laços, realiza a migração seletiva de páginas de memória segundo uma heurística. Seu objetivo é atenuar problemas de contenção e má alocação de memória em arquiteturas NUMA, sem que haja necessidade de se modificar código-fonte ou utilizar *hardwares* ou sistemas operacionais específicos. Para tanto, análises de compilação foram implementadas para instrumentação de código-fonte em busca de estruturas cujos dados sejam propícios à migração; ainda, uma heurística foi desenvolvida, capaz de avaliar se a migração de páginas se faz interessante ou se potencialmente prejudicaria o desempenho da aplicação. Obtivemos bons resultados, com ganho de desempenho de mais de 5x para alguns *benchmarks* - realizamos análises comparativas com outros dois mecanismos usados com o mesmo objetivo, e também apresentamos uma avaliação teórica de uma variedade de técnicas com o mesmo propósito de SPM.

Palavras-chave: *NUMA, Migração de páginas, Otimização de código, Compiladores, LLVM.*

Abstract

Multicore processors are increasingly common, especially due to technological barrier imposed by physical limitations in the growth of processors clock frequency. Non-uniform memory architecture (NUMA) are spreading as a solution to the scalability of high performance computing applications - such architectures, however, still exhibits problems, especially in the case of memory accesses. Bad data placement can cause contention on memory access leading to significant decrease in applications performance. We present Selective Page Migration (SPM), a compiler optimization that, by analyzing loops and their properties along with arrays accessed within such loops, can perform selective page migration according to a heuristic. The goal is to mitigate contention issues and bad data placement in NUMA architectures with no need to modify source code or using specific hardware or operating systems. To achieve this goal, compilation transformations have been implemented so we can perform source code instrumentation to find data structures that, once migrated, lead to an increase in program performance. Also, a heuristic has been developed so we're able to assess whether the migration of those data structures are likely to become profitable or will potentially impair the application performance. We've achieved good results, with speedup of more than 5x for some benchmarks. We have compared SPM with another two mechanisms used with the same goal, and also presented a theoretical evaluation of a variety of techniques with the same end of SPM.

Keywords: *NUMA, Page migration, Code optimization, Compilers, LLVM.*

Lista de Figuras

Lista de Tabelas

Lista de Abreviações e Siglas

SMP	*Symmetric multiprocessing* (pg. 12)
ccNUMA	*cache coherent Non-Uniform Memory Architecture* (pg. 13)
PTE	*Partition Table Entry* (pg. 19)
TLB	*Translation Lookaside Buffer* (pg. 19)
AutoNUMA	*Automatic NUMA balancing* (pg. 20)
SPM	*Selective Page Migration* (pg. 28)
IR	*Intermediate Representation* (pg. 30)
SSA	*Static Single Assignment* (pg. 30)
CFG	*Control Flow Graph* (pg. 36)
THP	*Transparent Huge Pages* (pg. 59)

Sumário

Capítulo 1

Introdução

A evolução dos microprocessadores nos últimos 25 anos se deu de modo impressionante; na década de 90 em média a cada 2 anos dobrava-se o número de transistores nos circuitos integrados, aumentando assim a frequência de operação dos processadores numa escala exponencial. Isso tudo sem um ônus muito claro, ou seja, uma situação de aparente ganho sem perda muito rara de se ver. Contudo, o custo de tamanha evolução chegou por volta do início do século XXI, em que esse aumento exponencial na frequência dos processadores foi barrado pela geração de calor ocasionada por limitações físicas - os processadores estavam atingindo temperaturas muito altas, inviabilizando assim o contínuo crescimento na frequência de operação [33]. Desse modo, um novo paradigma na construção de microprocessadores foi adotado: novos *chips* passaram a ser compostos por mais de um núcleo de processamento, isto é, num único *chip* encontram-se dois ou mais processadores de fato. Tal microprocessador é denominado *multicore*, e cada unidade de processamento é chamada *core* ou núcleo.

O uso de processadores *multicore* atrelado à difusão do paradigma da computação distribuída impulsionou pesquisas nas áreas de programação paralela e computação de alto desempenho - fez-se necessário buscar algoritmos paralelos e otimizações nos códigos existentes para explorar completamente o poder computacional oferecido por sistemas computacionais distribuídos e processadores *multicore*. No entanto, algumas dificuldades na obtenção de um bom desempenho em tais arquiteturas surgiram (ou foram acentuadas), como é o caso da disparidade entre poder computacional dos processadores e desempenho do acesso à memória [22]. Tal problema sempre esteve presente em menor escala, visto que a evolução das memórias se deu de modo mais lento que dos processadores, todavia com o avanço dos *multicores* essa situação se agravou.

Um dos grandes problemas relacionados ao acesso à memória em sistemas dotados de processadores *multicore* é a contenção do barramento. Se houver vários núcleos de processamento e apenas um controlador de memória, um processo de serialização no acesso à memória tende a ocorrer, visto que um único barramento provê suporte serial para a execução de requisições à memória. Um arranjo contendo apenas um único controlador de memória (e portanto um único barramento) está presente nas arquiteturas denominadas SMP (*Symmetric multiprocessing*) [30] - nos casos de apenas um processador com poucos núcleos, o possível gargalo ao se usar arquiteturas SMP pode ser aceitável, dada sua simplicidade e também devido aos mecanismos de memórias *cache* presentes nos pro-

cessadores, que podem atenuar possíveis perdas de desempenho. Além disso, algumas vantagens de arquiteturas SMP são o tempo constante de acesso à memória, visto que todos os processadores usam o mesmo "caminho" até os bancos de memória, e a disponibilidade de toda memória para todos os programas, de modo transparente, facilitando assim a alocação de dados por parte dos aplicativos, sem que seja necessário a preocupação sobre o banco no qual alocar memória, por exemplo. No entanto, quando há necessidade de maior poder computacional e um grande número de núcleos de processamento, arquiteturas SMP não são escaláveis. Uma alternativa nesse caso são as ditas arquiteturas ccNUMA (*cache coherent Non-Uniform Memory Architecture*), que provêm acesso integral e transparente à memória, assim como no caso SMP, mas contemplam vários controladores de memória [36].

Nas arquiteturas ccNUMA, um ou mais processadores (ou núcleos) estão conectados a um controlador e a alguns bancos de memória - tais arranjos são denominados *nodes* ou nós de processamento. Usualmente há vários nós e, portanto, vários controladores de memória num sistema desse tipo. Um acesso a um banco de memória conectado a um determinado controlador partindo de um processador do mesmo nó recebe o nome de acesso local; analogamente, se um acesso à memória parte de um processador conectado a outro nó, esse é denominado acesso remoto. Acessos locais possuem um tempo menor do que acessos remotos, pois para acessar um dado num controlador de memória em outro nó, uma rede de interconexão é usada; a razão entre os tempos de acesso a um dado local e remoto é chamada *NUMA factor* [29] e pode variar muito dependendo da distribuição dos controladores de memória e do número de nós e processadores por nó. Ainda que exista uma disparidade nesses tempos de acesso, arquiteturas ccNUMA são muito escaláveis e têm se tornado bastante populares, especialmente em ambientes de computação de alto desempenho. Assim como num arranjo SMP, uma arquitetura ccNUMA provê acesso integral à memória de modo transparente ao programador - mecanismos de *hardware* determinam se uma palavra está num banco de memória local ou remoto.

Tais arquiteturas não são, entretanto, isentas de problemas; em muitos casos, pode-se desenvolver o problema de contenção no acesso à memória, devido a uma alocação de memória ineficiente nos programas, por exemplo. Se um aplicativo realiza a alocação integral de sua memória num único nó, todos os futuros acessos efetuados, inclusive por *threads* em outros nós, serão direcionados para os bancos de memória associados ao controlador no qual a memória foi alocada, culminando em contenção de memória naquele barramento. A Figura 1.1 apresenta 2 casos, sendo o primeiro de contenção e o segundo de alocação eficiente. Apenas 2 nós são exibidos - as linhas pontilhadas representam em quais processadores os dados estão sendo requisitados, e a linha espessa entre os controladores de memória é a rede de interconexão, responsável pela transmissão de dados entre nós distintos. Na primeira porção da figura (à esquerda), todas as requisições de acesso à memória estão sendo efetuadas ao controlador 1, de modo que tal controlador fica sobrecarregado ao passo que o controlador 0 permanece ocioso, gerando uma situação de perda de desempenho na execução do programa. Na segunda metade da figura (à direita), há equilíbrio no uso dos controladores.

Tal problema é considerado grave, em especial nas aplicações que exigem alto poder computacional; se extrapolarmos o exemplo de contenção da Figura 1.1 para um sistema

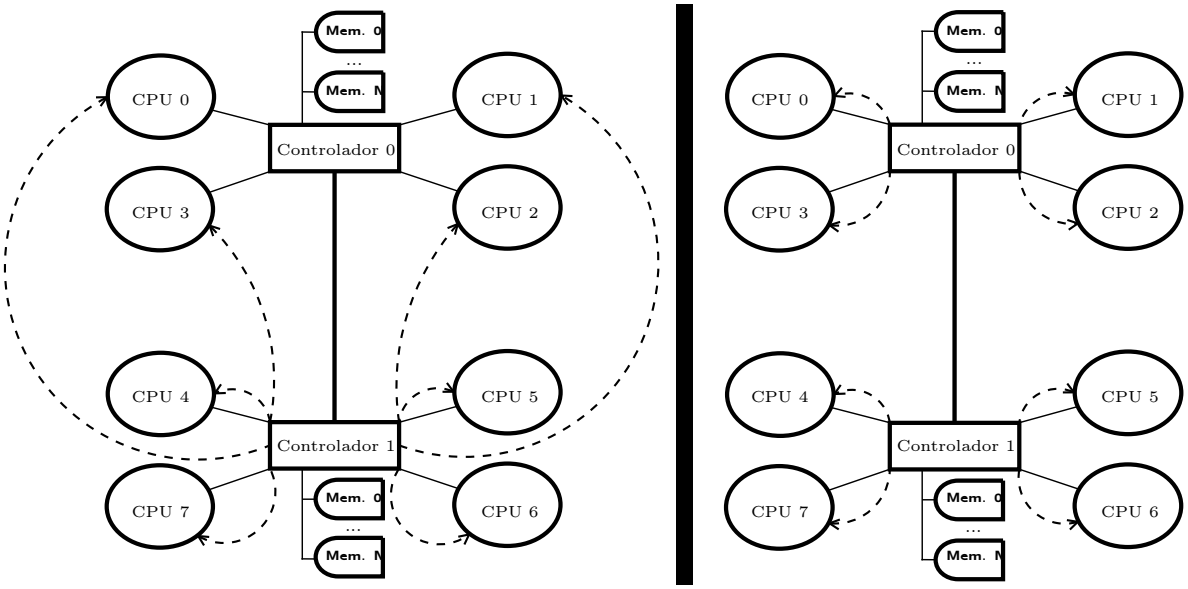


Figura 1.1: Na porção esquerda da figura temos um caso de contenção de memória; à direita vemos um caso de alocação eficiente.

de múltiplos nós executando um aplicativo do tipo *memory-bound*, a perda de desempenho será significativa [16]. A contenção é um problema bem mais prejudicial ao desempenho de aplicativos do que a disparidade nos tempos de acesso à memória, que tem se tornado menor nas arquiteturas modernas do tipo ccNUMA; nos primeiros sistemas o *NUMA factor* podia chegar a 10x, sendo que hoje não ultrapassa os 30% - num experimento comparando a queda de desempenho por acessos remotos ou contenção, Dashti *et al.* [26] obtiveram perdas de menos de 20% por acessos remotos e mais de 2x por contenção.

Variadas soluções são propostas na literatura para o problema de contenção de memória em arquiteturas ccNUMA [47]. Em primeiro lugar, uma solução elementar seria alterar os códigos-fonte para que realizassem uma alocação mais eficiente. Ainda que possível, tal solução é custosa do ponto de vista de tempo, visto que vários códigos são legados e uma varredura completa dos pontos de alocação de memória tende a ser não-trivial. Soluções mais automáticas no âmbito de *hardware*, sistemas operacionais, bibliotecas ou *runtime systems* foram propostas, cada uma das abordagens tendo suas vantagens e limitações. No capítulo 2 apresentamos um comparativo de várias técnicas que buscam lidar com problemas de perda de desempenho em tais arquiteturas. Na seção seguinte apresentamos detalhes do mecanismo de alocação do sistema operacional Linux e como se pode otimizar a alocação para evitar contenção de memória em sistemas ccNUMA.

1.1 Mecanismos de alocação de memória

Conforme descrito anteriormente, arquiteturas ccNUMA - embora escaláveis - podem sofrer de problemas relacionados à contenção de memória. Alguns mecanismos do sistema operacional Linux buscam otimizar a alocação de memória, mas no geral não levam em conta arquiteturas de memória não-uniforme. Por padrão, o sistema Linux usa uma po-

lítica de alocação efetiva de memória baseada no princípio de *lazy allocation*: páginas de memória são alocadas de fato apenas no momento em que são tocadas pela primeira vez. Tal política, denominada *first-touch* [31], funciona da seguinte forma: quando ocorre uma exceção do tipo *page-fault*, o sistema operacional aloca a página de memória num determinado banco de memória seguindo uma política de localização de dados pré-determinada. No caso padrão, a política de localização de dados é chamada *node local*, e contempla a alocação das páginas nos bancos de memória atrelados ao nó em que a solicitação foi efetuada. Isso significa, por exemplo, que se uma aplicação a ser executada num sistema de arquitetura ccNUMA possuir um procedimento que realiza todas as alocações e inicializações das estruturas de dados (isto é, as páginas são efetivamente tocadas) numa única *thread*, seguindo as políticas usuais do Linux haverá uma sobrecarga de apenas um controlador, provocando contenção e diminuindo o desempenho da aplicação.

Uma outra política de localização de dados, denominada *interleave*, permite a atenuação de tal problema: ao se utilizar tal política, a alocação efetiva ocorrerá de maneira distribuída em todos os nós, de modo circular, ou seja, as páginas serão alocadas começando do primeiro nó, com cada página sendo alocada no próximo nó, até que se volte ao primeiro, num estilo *round-robin*. Claramente essa política permite evitar a sobrecarga de um único nó devido ao espalhamento das páginas, e portanto a chance de haver problemas de contenção se reduz drasticamente. O sistema Linux utiliza a política *interleave* na sua inicialização, para que os módulos do *kernel* realizem uma alocação distribuída - no momento da execução do primeiro processo em modo usuário (denominado `init`), a política de localização de dados padrão é modificada para *node local* [36]. É possível executar qualquer aplicação utilizando a política *interleave*, bastando para tanto que se utilize a ferramenta `numactl` [34]: tal ferramenta consiste num pequeno utilitário a ser empregado na invocação de um aplicativo, permitindo que esse faça uso de determinada política de localização de dados. Pode-se optar pela política *interleave* bem como limitar a alocação a um subconjunto do total de nós. Em nossos testes, apresentados no capítulo 4, realizamos comparações de nossos resultados com aqueles obtidos mediante o uso do `numactl` - é importante ressaltar que apesar de evitar o problema de contenção, a política *interleave* pode espalhar os dados de modo que as *threads* executem em nós diferentes daqueles que possuem os dados utilizados por elas, gerando assim uma vasta quantidade de acessos remotos.

Uma estratégia para otimizar a alocação de memória e "aproximar" dados e *threads* que os utilizam é a migração de páginas de memória [35]. Tal mecanismo consiste em movimentar dados entre os bancos de memória fisicamente, isto é, os dados são levados de um banco de origem até um banco de destino. Para tanto, pode-se usar chamadas de sistema que permitem realizar a migração de maneira transparente aos aplicativos - tais *syscalls* realizam a movimentação de dados e atualizam a tabela de páginas da memória virtual, para que o mapeamento de endereços físicos e virtuais continue válido. Isso, contudo, é feito de modo a se preservar os endereços virtuais, daí a transparência do processo do ponto de vista dos programas. É importante distinguir a migração de páginas de memória das políticas discutidas acima: a migração ocorre após as páginas estarem alocadas nos bancos de memória, ao passo que as políticas de alocação de dados determinam o local em que os dados serão alocados, portanto sendo relevantes antes da

efetiva alocação desses.

Nosso trabalho, que rendeu uma publicação internacional [47], descreve uma solução para problemas de contenção em arquiteturas ccNUMA agnóstica de *hardware*, sistemas operacionais e *middlewares*, e que não demanda modificações nos códigos-fonte dos programas. Para tanto, nossa abordagem envolve o uso de modificações num compilador para que, durante a compilação de programas, avaliações de trechos quentes do código-fonte sejam realizadas, permitindo a inserção de testes comparativos simples que envolvem tamanho e reuso de estruturas de dados. Logo, em tempo de execução, através dos testes inseridos é determinado se é ou não vantajoso migrar as páginas dessas estruturas para bancos de memória pertencentes aos nós em que *threads* estão executando e fazem uso dos dados. No capítulo 2 apresentamos um comparativo de variadas técnicas propostas para resolver problemas de alocação em arquiteturas ccNUMA; no capítulo 3 descrevemos detalhadamente nossa abordagem, e no capítulo 4 mostramos os resultados experimentais obtidos. Por fim, apresentamos no capítulo 5 as conclusões e possíveis trabalhos futuros.

Capítulo 2

Trabalhos Relacionados

Neste capítulo, apresentamos um conjunto de trabalhos relacionados no que toca estratégias para otimização da localidade de dados em arquiteturas de memória não-uniforme - vários desses trabalhos apresentam também mecanismos que focam na afinidade de *threads*/dados, contudo deixamos de avaliar aqui os que lidam exclusivamente com escalonamento de tarefas, visto que nosso foco é na distribuição dos dados. Dividimos os trabalhos pelo seu âmbito de atuação: algumas abordagens se baseiam em mecanismos de *hardware*, enquanto parte das técnicas recai em modificações nos sistemas operacionais; há ainda uma porção de trabalhos que são implementados na forma de *middlewares* ou via anotações nos códigos-fonte das aplicações. Após a análise do conjunto de publicações, apresentamos na seção 2.5 uma tabela comparativa que sumariza as técnicas, agrupando-as por âmbito de implementação e indicando a estratégia de distribuição de memória utilizada (migração ou replicação de páginas) bem como suas restrições.

2.1 Abordagens implementadas em *hardware*

Nesta seção apresentamos abordagens baseadas na utilização do *hardware* para favorecer a alocação eficiente de dados nas arquiteturas de memória não-uniforme. Tais abordagens são transparentes para o usuário, não requerendo modificações nos programas. Contudo, padecem do ônus de serem especializadas demais, visto que apenas funcionam num único *hardware* específico, dificultando assim sua difusão nos meios mais orientados à produção.

O protótipo Sun Wildfire, avaliado por Noordergraaf e van der Pas [44] apresenta uma arquitetura ccNUMA não-ortodoxa, sendo composto por um diminuto número de grandes nós de arquitetura SMP - tal arranjo pode ser enxergado como um grande conjunto de sistemas SMP trabalhando de modo associado tal que se tenha a impressão de ser um único sistema ccNUMA. Devido à sua arquitetura não usual, possui alta latência de acesso à memória remota, mas apresenta mecanismos dinâmicos de replicação e migração de páginas. Contadores de *hardware* monitoram, na rede de interconexão, o número de vezes que uma linha de *cache* associada a uma página é lida de seu nó remoto sem modificações. Uma avaliação então é realizada com base em tais contadores: através de um *threshold* determina-se quando a página associada a essa linha é boa candidata para migração ou replicação - o *threshold* é derivado, entre outros fatores, pela quantidade de memória livre

por nó. Um serviço do sistema operacional então decide entre migração ou replicação de uma página, baseando-se em informações como quais páginas já foram replicadas em um ou mais nós ou recentemente migradas para outros nós. Bull e Johnson [21] avaliaram o desempenho do Wildfire utilizando o *benchmark* NPB [1]; a replicação se mostrou mais interessante que a migração - por exemplo, a execução do aplicativo CG (componente do *benchmark*) com replicação obteve tempo de 296 segundos, e com migração esse tempo foi de 699 segundos. Ambas abordagens se mostraram eficientes e nos testes em que estavam desativadas, o tempo de execução em geral foi maior.

Tikir e Hollingsworth [49] propõem um método dinâmico de migração de páginas no Sun Fire 6800, baseado no perfilamento das aplicações em tempo de execução com o uso de contadores de *hardware*. Para cada página de memória da aplicação, continuamente dados sobre qual processador a acessa mais frequentemente são coletados através de contadores, que monitoram a rede de interconexão. Tais contadores são capazes de prover informações do endereço físico requisitado e do tipo de transação efetuada. Em intervalos de tempo fixos durante a execução do programa, páginas são migradas para os nós que contêm os processadores que mais as acessam - páginas podem ser migradas mais de uma vez durante a mesma execução do programa, no entanto há mecanismos para evitar múltiplas migrações. Sendo os contadores implementados em *hardware*, as informações por eles fornecidas se referem a endereços físicos de memória; para que a técnica seja acurada, é necessário realizar um mapeamento desses para endereços virtuais, o que é feito via uma *syscall* do sistema operacional Solaris. Além disso, a migração de páginas ocorre também através de uma *syscall*, denominada `madvise`. Na avaliação desse trabalho foi utilizado o *benchmark* NPB [1], com resultados positivos: obteve-se até 90% de redução no número de acessos remotos e 16% de diminuição no tempo de execução das aplicações.

2.2 Técnicas baseadas em sistemas operacionais

Nesta seção avaliamos técnicas cujas abordagens são predominantemente baseadas em modificações nos sistemas operacionais. Mais uma vez, tais abordagens são transparentes para o usuário e agnósticas de *hardwares* especiais, contudo é preciso que se use as versões modificadas do sistema operacional para se obter os ganhos oferecidos por cada técnica.

Um dos trabalhos precursores em tal abordagem é o de Verghese *et al.* [51], em que é proposto um modelo de atribuição dinâmica de páginas (seja via migração ou replicação). Implementado no sistema operacional IRIX, tal modelo - batizado de DPP (*Dynamic Page Placement*) -, é baseado em quatro contadores limitantes (denominados *threshold counters*). O contador de disparo (*trigger*) contabiliza o número de *page misses* a partir do qual uma página é considerada quente, o de compartilhamento (*sharing*) acumula o número de *misses* de uma página em um processador remoto - indicando páginas boas para replicação -, o de escrita (*write*) armazena o número de escritas necessárias para a migração ser considerada medida eficiente, e finalmente o contador de migração (*migrate*) contabiliza o número de migrações a partir do qual uma página deixa de ser considerada boa para ser migrada. Há um *reset interval* usado para zerar os contadores acima. Tais contadores podem ser especificados de modo que o modelo fique flexível e adaptável a

cada conjunto de *workloads* em que se deseja testá-lo. Os resultados, mistos, foram em geral positivos (até 55% de redução no tempo de *stall* do processador), mas para cargas de banco de dados foram considerados ruins, não havendo melhora significativa. Uma possível causa foi o ajuste não otimizado dos contadores - Wilson e Aglietti [52] fizeram um novo estudo baseado na mesma técnica, que obteve ganhos de até 32% em *workloads* de um *benchmark* de banco de dados, através de ajustes mais precisos nos *thresholds*.

Goglin e Furmento [31] propõem um modelo misto de implementação em âmbito de sistemas operacionais e anotação de código. Sua técnica, batizada de política *Next-Touch*, consiste em adicionar um novo parâmetro à *syscall* `madvise`: tal chamada de sistema deve ser invocada em trechos do programa que presumivelmente vão seguir um novo padrão de acesso à memória, como por exemplo laços paralelizados com OpenMP; a *syscall* então anota as páginas e informa ao sistema operacional para que remova seus privilégios de leitura e escrita da PTE (*Partition Table Entry*), sendo que no próximo acesso, uma *page fault* é gerada e o gerenciador de memória dispara uma exceção. O tratador de exceções então checa se a página que gerou a exceção foi anotada pela *syscall*: em caso positivo, aloca uma nova página, copia o conteúdo da página geradora da exceção e deleta a original - uma vez que o sistema operacional segue a política *first-touch*, a nova página é alocada no exato nó em que de fato será usada. Em testes de desempenho usando a fatoração LU de matrizes, a política *Next-Touch* foi capaz de obter ganhos de até 129%.

Awasthi *et al.* [13] avaliam os impactos no desempenho das aplicações gerados por arquiteturas de memória não-uniforme e apresentam duas estratégias para mitigar os efeitos colaterais que podem surgir nesse caso. Primeiramente, a política *Adaptive First-Touch Page Placement* (AFT) de mapeamento de páginas é proposta: sempre que uma *thread* começa a executar num determinado núcleo, a cada *page fault* uma função-objetivo é minimizada, de modo a encontrar o nó no qual a página deve ser colocada para se atenuar a latência de acesso remoto. A seguir, contemplando o caso em que programas diminuem o acesso à páginas não alocadas e entram num ciclo de acesso aos dados já presentes na memória, um mecanismo de migração dinâmica de páginas é proposto: denominado *Dynamic Page Migration Policy*, tal política é baseada na avaliação dos controladores de memória, através do uso de contadores de *hardware* - mecanismos esses presentes na grande maioria dos processadores atualmente, em especial naqueles voltados para servidores. A avaliação é feita de maneira que controladores com demasiada carga sejam "doadores" e outros com menor taxa de uso sejam os "receptores" - assim, a cada intervalo definido de tempo um conjunto de páginas é migrado dos bancos atrelados ao doador para os bancos do receptor. O custo da migração, na forma de invalidação de entradas da *cache* e TLB (*Translation Lookaside Buffer*) é atenuado através de mecanismos internos de *lazy-evaluation* da política de migração. Além disso, a métrica usada para avaliação da carga nos controladores de memória, baseada em eventos no sistema de memória, gera algum grau de sobrecarga pelo uso constante de contadores de memória. Os resultados foram positivos, com ganhos de até 35 % e redução no consumo de energia de 14% (no caso da política AFT).

Uma abordagem embasada na distribuição automática de dados e *threads* foi proposta por um conjunto de desenvolvedores do *kernel* do sistema operacional Linux, em princípio por Andrea Arcangeli [23] [32]. Inicialmente duas metodologias distintas foram apresentadas, uma buscando realizar otimizações antes da ocorrência de gargalos por má

alocação de memória, e a outra tendo como objetivo otimizar o código no momento da execução dos aplicativos, com enfoque em avaliações de perfilamento para determinação da localidade ótima de dados e tarefas. A segunda abordagem prevaleceu, sendo denominada *Automatic NUMA balancing* (AutoNUMA). Com base em análises do número de *page-faults* induzidos pelo sistema (através de páginas marcadas contra leitura e escrita), o mecanismo pode realizar, numa primeira etapa, a movimentação de tarefas para os nós apropriados, ou seja, os quais contêm dados usados pelas *threads*. Num segundo momento, dados acessados pelas *threads* mas localizados em nós remotos são migrados para os nós locais à elas; estatísticas sobre a localização de tarefas e dados são mantidas para que periodicamente migrações de *threads* e dados sejam realizadas para otimizar a localidade de dados/tarefas. Apresentamos no capítulo 4 análises comparativas do ganho de desempenho do mecanismo AutoNUMA quando comparado à nossa proposta, bem como uma descrição mais detalhada do mecanismo.

Dashti *et al.* [26] apresentam Carrefour, um mecanismo implementado no *kernel* do sistema operacional Linux com o objetivo de atenuar problemas de localidade de dados que geram contenção nos controladores de memória. Primeiramente, avaliou-se o impacto de acessos remotos e contenção nos controladores de memória - como esperado, o impacto da contenção é muito mais prejudicial ao desempenho dos programas do que acessos remotos em arquiteturas ccNUMA modernas. Três são as estratégias usadas por Carrefour para reduzir os problemas de contenção: replicação de páginas, migração de páginas e *interleaving*, i.e., dispersão de páginas pelos controladores de memória. O mecanismo faz intenso uso de contadores de *hardware* para definir sua estratégia de atuação: em princípio, baseado num contador, apenas aplicações que causam "congestionamento" de tráfego de dados ativam Carrefour. Uma vez ativado, ele avalia mais contadores para determinar se a solução mais efetiva é a replicação, migração ou *interleaving* - nem sempre a replicação é ativada, pois em casos de páginas com muitas escritas pode haver aumento considerável da sobrecarga do mecanismo, dada a necessidade de se manter um módulo de coerência entre as páginas replicadas. Por fim, uma avaliação sobre mecanismo de *hardware* benéficos para a implementação de Carrefour é realizada. Os resultados foram muito positivos, obtendo-se ganhos de até 3.6x e máxima sobrecarga de apenas 4% nos *benchmarks* avaliados.

Diener *et al.* [28] propõem um mecanismo de otimização de afinidade de dados e *threads* implementado no *kernel* do sistema operacional Linux. Denominado kMAF, tal mecanismo permite avaliar quais dados são acessados por quais *threads* através do sistema de memória virtual do Linux; juntamente com informações topológicas da máquina, pode-se inferir um grau otimizado de afinidade entre dados e *threads* e então migrar tanto páginas de memória quanto *threads*. Seu funcionamento pode ser dividido em 4 etapas: primeiro, informações sobre o padrão de acesso à memória são coletadas através da contagem de *page faults*; a seguir, tais informações são analisadas e armazenadas em uma lista, junto dos nós e *threads* que acessaram cada página. No próximo passo, uma avaliação por nó é realizada, para se inferir o quão vantajoso seria migrar páginas para outros nós; finalmente, um algoritmo de mapeamento de *threads* é executado periodicamente para determinar a necessidade de migrá-las. Com extensas avaliações em variados *benchmarks*, os resultados foram bastante positivos: obteve-se ganhos de até 35% e redução no consumo

de energia de até 34%.

Cruz *et al.* [25] descrevem LAPT (*Locality-Aware Page Table*), um mecanismo híbrido de mapeamento eficiente de dados e *threads* baseado em modificações no *hardware* e na tabela de páginas do sistema operacional. A abordagem de LAPT é dividida em duas etapas: primeiramente, através da avaliação de *misses* na TLB, um vetor por página é mantido na tabela de páginas, determinando quais *threads* as acessaram recentemente; também um vetor que registra a comunicação entre *threads* é gerado. Para a construção e acesso aos vetores descritos faz-se uso de estruturas de *hardware*, como registradores adicionais e multiplexadores, visto que os dados fornecidos a tais vetores são provenientes da TLB e abordagens de modificações de *hardware* tendem a apresentar menor sobrecarga. Na segunda etapa, um mapeamento de *threads* para núcleos de processamento é realizado, com base no vetor de comunicação de *threads*, e a migração de páginas ocorre caso seja vantajosa - o sistema operacional periodicamente percorre a tabela de páginas e avalia os vetores para checar se o mapeamento entre a localização dos dados e das *threads* que os acessam é eficiente, realizando a migração de páginas caso contrário. Os resultados, baseados no uso de um simulador (devido às modificações necessárias no *hardware* pela técnica), foram positivos, obtendo-se redução de até 35% no tempo de execução dos *benchmarks*.

2.3 Técnicas baseadas em *middlewares*

Nesta seção consideramos abordagens baseadas em *middlewares*, termo usado aqui com o sentido de sistemas computacionais que compreendem uma camada de *software* entre uma aplicação e o sistema operacional, de modo que *runtime systems*, máquinas virtuais e arcabouços de execução no geral pertencem a essa categoria. Uma vez que a responsabilidade por lidar com migrações de páginas e avaliações dinâmicas sobre localidade dos dados ou *threads* é do *middleware*, tais abordagens são independentes de modificações profundas nos códigos-fonte das aplicações - por vezes, pode-se requerer alguma alteração nos códigos, mas não comparável ao uso de bibliotecas, por exemplo, que pode demandar reestruturação extensa do código-fonte das aplicações. A necessidade da instalação do *middleware* e da manutenção de sua compatibilidade com o sistema operacional se impõem como pontos fracos na sua utilização, além do escopo reduzido, visto que as abordagens abaixo elencadas somente se aplicam a programas que executam sobre os *middlewares* específicos.

Nessa categoria, Nikolopoulos *et al.* [43] apresentam algoritmos que relacionam migração de páginas com o escalonamento de *threads*. De tempos em tempos, páginas são migradas tendo como base a correlação entre as referências a tais páginas e informações do escalonamento das *threads* que as referenciam - migrações são efetuadas quando é detectada uma quebra na afinidade entre as *threads* e os dados acessados por elas. A idéia é utilizar informações interceptadas no sistema operacional sobre preemptividade e alterações no escalonamento de *threads* como gatilho para migrar páginas de maneira eficaz. Dois algoritmos são propostos: o primeiro deles, chamado preditivo, foca em programas ditos paralelamente iterativos (em que a mesma computação paralela é repetida várias vezes) e portanto considera algumas premissas na execução do código; o segundo

algoritmo é generalista e não faz uso de nenhuma inferência sobre o comportamento dos programas. Ambos os algoritmos são implementados como modificações num *runtime system* para OpenMP, utilizando a interface de gerenciamento de memória do sistema operacional IRIX para obtenção de informações de escalonamento de *threads* e migração de páginas. Na avaliação de sua eficácia, os *benchmarks* analisados obtiveram resultados bastante positivos, com mais de 200% de ganho de desempenho em alguns casos.

Ogasawara [46] propõe um *garbage collector* (GC) capaz de otimizar a localidade dos dados num âmbito de máquinas virtuais para linguagens de alto-nível; o mecanismo foi implementado em uma máquina virtual Java. A idéia consiste em usar o conceito de *Dominant Thread* (DoT), que representa a *thread* que realiza a maior parte dos acessos a uma determinada página, para detectar em qual nó alocar tal página - informações sobre uso de páginas por *threads* são em geral disponibilizadas em *runtimes* de alto-nível naturalmente, não agregando sobrecarga nesse caso. A técnica primeiramente, durante a execução do programa, aloca novas páginas em nós preferenciais, ou seja, naqueles em que os dados serão mais utilizados. Também, devido aos mecanismos inerentes do GC para liberação de espaço e para que se evite fragmentação de memória, páginas são periodicamente realocadas em outros bancos de memória - aqui, a técnica proposta pelo autor novamente utiliza as informações sobre quais nós são preferenciais e realiza essa realocação de forma eficiente. Na implementação, optou-se por uma política de invocação do GC mais agressiva do que a usual em máquinas virtuais de Java - a sobrecarga contudo não foi afetada de modo intenso. Os resultados foram positivos, obtendo-se até 53% de ganho de desempenho nos *benchmarks*.

Broquedis *et al.* [19] [20] descrevem ForestGOMP, um *runtime system* para OpenMP capaz de escalonar *threads* eficientemente e migrar páginas de memória dinamicamente. Em primeiro lugar, é proposto um escalonador de *threads* que utiliza informações da topologia de *hardware* extraídas em tempo de execução para realizar o escalonamento eficiente - as *threads* são organizadas em entidades denominadas "bolhas" que permitem explorar propriedades de afinidade entre elas. Uma biblioteca de gerenciamento de memória é então usada para efetuar a migração dinâmica de páginas e prover variadas informações ao *runtime system* sobre o uso da memória, tais como espaço livre por nó e custo de mover páginas. Finalmente, o *runtime system* propriamente dito associa ambos os mecanismos descritos: usando as informações fornecidas pelo gerenciador de memória e analisando as "bolhas" formadas pelo escalonador, ForestGOMP é capaz de distribuir de modo eficiente as *threads* nos nós e migrar páginas de memória caso seja conveniente. É possível ainda anotar o código-fonte das aplicações com informações adicionais para otimizar a distribuição de *threads*. Testes feitos com o *benchmark* STREAM mostraram resultados positivos, obtendo-se ganhos na vazão de memória de até 70%.

Blagodurov *et al.* [15] avaliam escalonadores dinâmicos de *threads* e observam que em geral a abordagem proposta por eles não leva em conta arquiteturas de memória não-uniforme, piorando problemas de contenção em tais arquiteturas; mais ainda, os autores experimentalmente determinam que a principal causa na degradação de desempenho de *softwares* em arquiteturas ccNUMA é a contenção de memória, e não o número de acessos remotos. Os autores apresentam um algoritmo de escalonamento dinâmico de *threads* otimizado para arquiteturas ccNUMA, denominado DINO. Correlacionando *misses* da *cache*

com a localização de *threads* em nós, o algoritmo busca um escalonamento eficiente de *threads* juntamente de uma distribuição otimizada de seus dados, através de um mecanismo de migração de páginas: usando contadores de perfilamento do *hardware* e informações sobre acessos remotos das *threads*, dada uma constante K, têm-se $K/2$ páginas anteriores e posteriores à página acessada remotamente (incluindo ela mesma) migradas para o nó em que a *thread* está executando - tal estratégia é denominada *sequential-forward-backward*. Os módulos de migração e avaliação de *threads* foram implementados em nível de usuário no sistema Linux, na forma de *daemons*; vários experimentos foram realizados, inclusive variando o valor da constante K (valores mais altos apresentaram melhores resultados) e um mecanismo para evitar migrações excessivas de páginas foi implementado. Os resultados gerais foram positivos, tendo-se obtido ganhos de mais de 30% em algumas aplicações dos *benchmarks* SPEC CPU 2006 [2] e SPEC MPI 2007 [3].

Wittman e Hager [53] realizam um comparativo entre as bibliotecas Intel TBB [4] e OpenMP, além de apresentarem o conceito de fila de localidade, que pode ser entendido como um *pool* cujo escopo é um nó e do qual as *threads* retiram tarefas a serem executadas. As abordagens de escalonamento e distribuição de tarefas variam entre as bibliotecas - na TBB, o processo é bastante transparente e automático, ao passo que a biblioteca OpenMP possui alguns modos pré-definidos e permite algum controle por parte do programador; ainda assim, o escalonamento de tarefas por *threads* é por vezes realizado de modo arbitrário, sem levar em conta a localidade requerida para o bom desempenho em arquiteturas ccNUMA. A noção de fila de localidade é idealizada de modo a se garantir que *threads* obtenham tarefas para execução provenientes do nó em que estão alocadas. Primeiramente as páginas da aplicação são tocadas para se obter um mapeamento página-nó; em seguida, as *threads* são iniciadas seguindo tal mapeamento, e através de uma chamada de sistema, são "travadas" em tais nós. Passam então a consumir tarefas da fila de localidade existente em seu próprio nó - o escalonamento dinâmico continua ocorrendo livremente no escopo de um nó. A abordagem foi avaliada num *toy benchmark* e os resultados do escalonamento com fila de localidade foram superiores quando comparados à não utilização da mesma.

Majo e Gross [41] apresentam uma API que permite a distribuição da computação e dos dados de forma eficiente em arquiteturas ccNUMA. Primeiramente, um estudo detalhado do *benchmark* NPB é descrito, contemplando avaliações do padrão de acesso à memória de matrizes e modelagens da frequência de acesso à memória - foi utilizado um sistema composto por 2 processadores da família Nehalem da Intel, que apresenta arquitetura ccNUMA e mecanismos específicos de perfilamento implementados em *hardware*. Uma API é então proposta, provendo dois tipos de primitivas: a primeira é responsável pela distribuição de dados da aplicação, enquanto que a outra permite otimizar a distribuição de iterações de laços através de novas cláusulas de escalonamento, que são implementadas como novos atributos em diretivas `pragma` de OpenMP. A avaliação dos resultados foi bastante positiva, de modo que após a utilização da API no código obteve-se um ganho de desempenho máximo de até 3.3x (com média de 1.7x), devido à melhor distribuição dos dados na memória e escalonamento eficiente das iterações dos laços.

2.4 Técnicas baseadas em anotações no código-fonte

Apresentamos nesta seção técnicas implementadas na forma de anotações nos códigos-fonte das aplicações. Nessa categoria estão contempladas abordagens que envolvem alterações mais profundas nos códigos, seja através de chamadas de sistema ou implementações de rotinas especializadas. A principal diferença entre abordagens de *middleware* e de anotações nos códigos-fonte é que, enquanto nos *middlewares* os mecanismos de distribuição/migração de dados e tarefas são implementados na camada intermediária, aqui todo o arcabouço de avaliações de localidade dos dados e migrações de páginas é implementado no código-fonte da aplicação. Ao se optar pelo uso desta abordagem há naturalmente um maior comprometimento da portabilidade, e as modificações decorrentes de atualizações de bibliotecas, por exemplo, podem implicar na necessidade de se realizar novas alterações nos códigos-fonte dos programas.

Löf e Holmgren [40] apresentam uma releitura da política *first-touch* batizada de *affinity-on-next-touch* e implementada no sistema operacional Solaris. A idéia é anotar o código de modo a reinvocar a política *first-touch* em trechos avaliados como quentes no que toca o acesso à memória - o que caracteriza tais trechos como interessantes é a mudança no padrão de acesso à memória e alternância de fases durante a execução. Além disso, por vezes a alocação de memória ocorre integralmente na *thread* principal, induzindo a política *first-touch* a um comportamento errático - a alocação nesse caso ocorre apenas nos bancos de memória locais em relação à *thread* principal. Assim, a proposta dos autores é permitir a reinvocação da política *first-touch*, que então recalcula a afinidade dos dados a serem alocados com o estado atual das *threads*, favorecendo assim o balanceamento de novos dados alocados, e migrando páginas para que esse balanceamento seja atingido nos dados previamente alocados. Usando como *benchmark* um aplicativo de resolução de equações diferenciais, os resultados foram positivos, apresentando ganhos de desempenho de até 70% numa execução com 22 *threads*. Uma análise da sobrecarga gerada pela técnica foi descrita e revelou que o custo de recalcular afinidade e mover páginas é irrelevante se comparado com o custo de atualizar a TLB, de modo que páginas maiores melhoram ainda mais os resultados - o ganho de desempenho ultrapassou 160% quando foram usadas páginas de 64KB em detrimento das de 8KB.

Nordén *et al.* [45] investigam o impacto da localidade de dados num aplicativo de análise numérica. Eles apresentam o conceito de localidade geográfica, que está relacionado à coesão de dados e *threads* num mesmo nó, com intuito de favorecer uma execução balanceada e sem contenção nos controladores de memória. Para avaliar o impacto da localidade geográfica, o *solver* utilizado faz uso de uma técnica de malhas adaptativas, o que acarreta modificação dos dados a cada passo da execução, comprometendo a eficiência da política *first-touch*. Utilizando-se de uma máquina ccNUMA Sunfire 15000 com o sistema Solaris, a avaliação comparativa leva em conta execuções num único *core*, execuções espalhadas pelos núcleos sem migração de dados e por fim espalhadas com migração dinâmica. Uma diretiva para a *syscall* `madvise` é proposta - denominada `migrate-on-next-touch`, permite que páginas sejam marcadas para migração futura tendo como destino o nó em que determinada *thread* fará uso da página. Os resultados apontam ganhos de desempenho de até 40% ao se utilizar a migração de dados com execução espalhada pelos *cores*.

Ribeiro *et al.* [48] propõem uma API - denominada Minas - capaz de alocar dados de forma eficiente, e um mecanismo de alteração automática de código-fonte para utilização de tal API. O conjunto de funções implementadas permite alocação distribuída de dados, gerenciamento de políticas de memória e obtenção de informações da topologia do sistema. Ainda que se possa alterar manualmente o código-fonte de uma aplicação para usar a API, um pré-processador é proposto, de modo que através de informações sintáticas do código-fonte, tal pré-processador efetua substituições no código para que automaticamente seja feito uso das funções descritas na API. Os ganhos de desempenho foram de 30% em média no *benchmark* NPB.

Marathe *et al.* [42] descrevem uma abordagem híbrida para favorecer a eficiente alocação de dados em aplicações executadas nas arquiteturas ccNUMA. Através de traços de memória, obtidos durante uma primeira execução do programa a ser otimizado, uma alocação eficiente é realizada na segunda rodada por funções específicas. Tais funções utilizam variáveis de ambiente para determinar se está ocorrendo a primeira execução da aplicação, em que os traços de memória são gerados, ou a segunda, em que eles são analisados e servem de guia para otimizações na alocação de dados, realizadas por *syscalls* que determinam afinidade entre dados e *cores*. Tal abordagem - que pode ser considerada um tipo de otimização guiada por perfilamento (*profile-guided optimization*) - necessita de anotações no código-fonte, para que as funções usuais de alocação de memória sejam substituídas por *wrappers* para as funções propostas pelos autores - ainda, o programador deve anotar trechos no código-fonte considerados estáveis do ponto de vista de mudança de fase, pois na etapa de coleta de traços esses trechos são os mais relevantes para que se tenha bons resultados com o uso da técnica. Além disso, por se basear na obtenção de traços de memória, o mecanismo depende do *hardware*, pois esse deve prover um conjunto específico de contadores e recursos que são necessários na primeira execução do arcabouço. Os resultados, positivos, possibilitaram redução de até 20% no tempo de execução das aplicações.

Borin *et al.* [17] realizaram um estudo com base na biblioteca NeoPZ [5], que faz uso do método dos elementos finitos: em uma máquina de arquitetura ccNUMA com 64 núcleos de processamento, 64 *threads* foram criadas para que cada uma realizasse decomposições Cholesky com matrizes replicadas nas mesmas. A análise demonstrou que a alocação das matrizes estava desbalanceada, de modo que apenas bancos de memória conectados a um determinado controlador estavam sendo usados para a alocação, pois essa era realizada integralmente na *thread* principal. Três técnicas foram avaliadas com o objetivo de otimizar a alocação de memória. A primeira delas foi o uso da ferramenta `numactl`, presente no pacote da `libnuma` [34], que permite invocar um programa com uma política de alocação específica; no caso a política `interleave` foi utilizada, distribuindo as páginas de memória num esquema *round-robin* entre os bancos de memória. A segunda técnica foi o uso da biblioteca `libnuma` em si, através de modificações no código-fonte - as matrizes foram completamente alocadas em nós separados (usando uma estratégia *round-robin*) e as *threads* foram travadas nos nós, podendo ser escalonadas em núcleos dentro de um mesmo nó. Finalmente, a técnica *copy-on-thread* foi proposta: tal técnica consiste em efetuar a cópia das matrizes dentro de cada *thread*, explorando a política *first-touch* do sistema operacional; assim, no momento da cópia de uma matriz no início

da *thread*, naturalmente os dados são alocados no banco mais próximo à ela. Em uma avaliação comparativa das 3 técnicas, obteve-se ganho de desempenho em todos os casos, sendo observado ganho de desempenho de 10.6x com o uso da ferramenta `numactl`, 30.9x com o uso da `libnuma` e 25.5x através da técnica *copy-on-thread*.

2.5 Tabela comparativa

A Tabela 2.1 apresenta uma síntese das técnicas discutidas nesse capítulo, agrupadas por seu âmbito; os grupos são delimitados pelas linhas mais espessas na tabela, seguindo a ordem descrita no capítulo: *hardware* (implementadas em *hardwares* especiais), sistemas operacionais, *middlewares* e anotações no código. A tabela relaciona as publicações com suas características relevantes, por exemplo se a técnica proposta faz uso de contadores de *hardware* ou se necessita de recompilação do código dos aplicativos. Nem todas as técnicas fazem uso de mecanismos de migração ou replicação de páginas - parte delas utiliza estratégias de alocação inicial eficiente, sem contemplar movimentação de dados durante a execução do programa.

Uma vez que a Tabela 2.1 apresenta vários campos, utilizamos siglas em prol da redução de espaço. A seguir apresentamos o significado de cada sigla:

- R. ou Replicação: a técnica faz uso de replicação de dados;
- M. ou Migração: a técnica realiza algum tipo de migração de dados, seja via chamada de sistema ou mecanismo de reinvocação da política *first-touch*;
- P. ou Perfilamento: a técnica faz uso de algum tipo de perfilamento em suas análises;
- C. *H.* ou Contadores de *Hardware*: a técnica faz uso de contadores de *hardware*;
- M. *H.* ou Modifica *Hardware*: a técnica exige modificação no *hardware* para seu funcionamento;
- R. *M.* ou Requer *Middleware*: a técnica funciona apenas em aplicações que executam sobre um determinado *middleware*;
- R. R. ou Requer Recompilação: a técnica requer recompilação do código da aplicação para ser utilizada, seja por necessidade de anotação no código-fonte ou ligação com biblioteca.

Nenhum dos trabalhos acima - apesar de relacionados ao tema do nosso projeto e com mesmo foco - faz uso de análises de reuso de dados em tempo de compilação para otimizar a distribuição de dados em tempo de execução nas arquiteturas ccNUMA. O *framework* Minas, já mencionado, apresenta um pré-processador capaz de substituir funções de alocação nos programas automaticamente, mas não leva em conta o reuso dos dados e suas relações com laços, além de não postergar análises para o tempo de execução, tendo assim sua generalidade reduzida. Não temos conhecimento, apesar de pesquisa exaustiva de trabalhos relacionados, de uma análise similar à nossa no âmbito de compiladores. Um trabalho que se baseia em compiladores mas com foco diferente é o de

Publicação	Nome	Ano	R.	M.	P.	C. *H.*	M. *H.*	R. *M.*	R. R.
Noordergraaf *et al.* [44]	-	1999	•	•	•	•			
Tikir e Hollingsworth [49]	-	2004		•	•	•			
Verghese *et al.* [51]	DPP	1996	•	•	•				
Goglin e Furmento [31]	*Next-Touch*	2009		•					•
Awasthi *et al.* [13]	AFT	2010		•	•	•			
Arcangeli *et al.* [23] [32]	AutoNUMA	2012		•	•				
Dashti *et al.* [26]	Carrefour	2013	•	•	•	•			
Diener *et al.* [28]	kMAF	2014		•					
Cruz *et al.* [25]	LAPT	2014		•			•		
Nikolopoulos *et al.* [43]	-	2000		•				•	
Ogasawara [46]	-	2009		•				•	
Broquedis *et al.* [19] [20]	ForestGOMP	2009		•				•	•
Blagodurov *et al.* [15]	DINO	2010		•	•	•			
Wittman e Hager [53]	-	2010						•	
Majo e Gross [41]	-	2012						•	•
Löf e Holmgren [40]	*affinity-on-next-touch*	2005		•					•
Nordén *et al.* [45]	-	2008		•					•
Ribeiro *et al.* [48]	Minas	2010							•
Marathe *et al.* [42]	-	2010			•	•			•
Borin *et al.* [17]	*copy-on-thread*	2015		•					•

Tabela 2.1: Sumário das técnicas descritas.

Li *et al.* [39], em que se utiliza um conjunto de análises em tempo de compilação para determinar o particionamento otimizado dos dados com relação ao conjunto de *threads*. Tal abordagem recai na avaliação de chamadas da função `malloc` nos programas, no momento da compilação: informações de "localização" adequada dos dados em relação às *threads* são capturadas e então um *hook* para a função `malloc` é usado para que a alocação seja feita de forma otimizada. O trabalho tem um foco teórico e foi testado em ambiente de simulação apenas - seu objetivo é melhorar a distribuição inicial dos dados. Nosso trabalho, por outro lado, tem enfoque na distribuição eficiente dos dados em tempo de execução, e faz uso de análises de reuso e tamanho das estruturas para tanto; nossa otimização pode ser usada em ambiente de produção, sem que seja necessário remapear funções de alocação, bastando que haja suporte do sistema operacional para obtenção de informações sobre o tamanho da *cache* e uma *syscall* de migração de páginas. No capítulo seguinte, apresentamos os detalhes de nossa otimização de compiladores.

Capítulo 3

Selective Page Migration (SPM)

Conforme apresentado no capítulo 1, arquiteturas ccNUMA podem apresentar problemas de contenção de memória; aplicações executadas em tais arquiteturas, dependendo do modo que realizam sua alocação de memória, podem sofrer queda de desempenho, ainda que executando em sistemas bastante poderosos e com múltiplos núcleos de processamento. No capítulo 2 apresentamos um conjunto de técnicas propostas para atenuar esse tipo de problema. No entanto, pudemos avaliar que todas as técnicas recaem em *hardwares* especializados, modificações no sistema operacional ou código-fonte das aplicações, ou ainda uso de *middlewares* específicos. Neste trabalho descrevemos uma técnica de otimização no âmbito de compiladores, que busca minimizar o problema de contenção em arquiteturas ccNUMA através de transformações no código e migrações automáticas de páginas de memória - tal técnica, denominada *Selective Page Migration* (SPM), consiste basicamente nos seguintes passos:

1. Consideramos laços como os trechos mais quentes dos programas; realizamos então uma estimativa do número de iterações dos laços com base em suas condições de controle. Não é possível, no caso geral, realizar essa estimativa em tempo de compilação, pois a condição de parada pode ser uma variável global ou valor a ser fornecido pelo usuário. Portanto montamos expressões numéricas contendo inferências sobre os laços, a serem completadas em tempo de execução.

2. De posse de uma expressão que representa o número estimado de iterações de um laço, inferimos a quantidade de acessos à memória para estruturas de dados contidas no laço, sejam esses acessos de escrita ou leitura. Ainda, utilizando informações das condições de controle do laço e com base na expressão de cálculo do endereço de acesso das estruturas de dados, determinamos o conjunto de páginas de memória pertencentes a ela.

3. A partir da estimativa de quantas vezes determinadas páginas são acessadas, inserimos chamadas para uma função de migração condicional de memória antes do laço no qual tais páginas serão acessadas, com o objetivo de migrá-las para o nó em que serão usadas em computações diversas. A função de migração é condicional pois leva em conta o reuso estimado dos dados e o tamanho total dos mesmos - se couberem

na *cache* ou forem pouco reutilizados, a sobrecarga imposta pela migração pode não trazer benefício, em alguns casos piorando o desempenho.

4. Tendo sido concluídos os passos acima, em tempo de execução o código inserido avalia se a migração é vantajosa ou não, e realiza a chamada de sistema responsável pela migração de páginas em caso positivo. Se os dados já estiverem nos bancos de memória locais aos processadores que os utilizarão, a *syscall* é inócua, ou seja, não realiza movimentação de páginas. Além disso, um mecanismo de travamento de *threads* em nós de processamento pode ser opcionalmente habilitado, favorecendo ainda mais a eficácia da técnica.

Um código-modelo que exprime nossa idéia pode ser visto na Figura 3.1: na parte (a) temos um código-fonte em C, que é executado em *threads* e tem desempenho inferior ao esperado, por problema de contenção; em (b) temos a versão do código modificada após a execução da otimização SPM. Nosso trabalho, que gerou uma publicação numa conferência internacional (*Compiler Support for Selective Page Migration in NUMA Architectures* [47]), apresentou resultados bastante positivos, a serem discutidos com maior profundidade no capítulo 4. Nas seções a seguir, as etapas de desenvolvimento e implementação da técnica *Selective Page Migration* serão detalhadas.

```
1  for (i=0; i<n; i++) {
2    num=0;
3
4    for (j=start; j<end-str_sz
      +1; j++) {
5      int success=1;
6
7      for (k=0; k<str_sz; k++)
8        if (array[j+k] !=
       pattern[k]) {
9          success=0;
10         break;
11       }
12
13     if (success)
14       num++;
15   } //for_j
16 } //for_i
```

(a)

```
1  R = (n+1)(end-str_sz+1-start)(
       str_sz+1);
2  spm_call(array,start,end-1,R);
3  for (i=0; i<n; i++) {
4    num=0;
5
6    for (j=start; j<end-str_sz
      +1; j++) {
7      int success=1;
8
9      for (k=0; k<str_sz; k++)
10       if (array[j+k] !=
       pattern[k]) {
11         success=0;
12         break;
13       }
14
15     if (success)
16       num++;
17   } //for_j
18 } //for_i
```

(b)

Figura 3.1: No bloco (a) temos um código em C, que após ser instrumentado pela otimização SPM, é convertido no código do bloco (b), tendo sido a linha em destaque adicionada pela otimização. Note que representamos o reuso como variável `R` apenas para melhor visualização.

3.1 Transformações implementadas no compilador

A implementação da técnica SPM pode ser dividida em 3 partes principais: o passo de otimização, a heurística de migração de páginas e o mecanismo de travamento de *threads*, adicionado ao projeto após a publicação do artigo. Nesta seção detalharemos a implementação das transformações no compilador. Para realização desta etapa do projeto, necessitamos realizar alterações num compilador - tal tarefa não é trivial, já que compiladores possuem em geral código extenso e complexo. Entretanto, nos últimos anos uma ferramenta bastante poderosa e de fácil manipulação vem sendo cada vez mais utilizada: a infraestrutura do LLVM[1] [38].O LLVM contempla um conjunto de ferramentas que associadas formam um compilador de várias linguagens para várias plataformas. Essa versatilidade advém de sua modularidade: desde o princípio o LLVM tem como pilar do seu desenvolvimento o espalhamento de suas funcionalidades em módulos, e o uso massivo de orientação a objetos em seu código - com isso obtém-se uma ferramenta facilmente adaptável. Além disso, a infraestrutura é completamente *open-source* e seu desenvolvimento é muito ativo tanto no meio acadêmico quanto por empresas, devido à facilidade em se usar o LLVM para gerar novos compiladores e otimizações. Um dos principais benefícios do LLVM é sua representação intermediária [37] (referida a seguir como IR, de *Intermediate Representation*). Ela possui uma sintaxe similar a uma linguagem de montagem, contudo representa o programa numa estrutura SSA (*Static Single Assignment*) [12] e apresenta conceitos mais sofisticados do que um típico *assembly*, como funções e alocação explícita de memória - além disso a IR do LLVM é muito bem documentada [6].

A estrutura de compilação do LLVM funciona conforme a Figura 3.2. Seguindo o princípio da modularidade, os arquivos de código-fonte são convertidos pelo *Frontend* na representação intermediária. Essa por sua vez é alvo de otimizações, que são implementadas na forma de bibliotecas chamadas pelo otimizador. Então a IR é convertida pelo *Backend* numa linguagem de montagem para a arquitetura-alvo desejada, gerando um arquivo-objeto; por fim, o *Linker* gera o código executável final a partir de um ou mais arquivos-objeto. Essa abordagem tem várias vantagens:

1. A independência entre os passos facilita tanto a análise como implementação de qualquer etapa do processo.

2. As otimizações são independentes de plataforma, e também por sua construção na forma de bibliotecas, são simples de modificar.

3. A IR se comporta de maneira "canônica" - toda transformação mantém a IR padronizada.

Optamos, portanto, pela utilização do LLVM nesse trabalho.

Uma vez determinada a infraestrutura, nosso foco recai nas avaliações e transformações de código necessárias para a realização de nosso objetivo: inserir chamadas para uma função de migração de páginas antes de blocos quentes em que estruturas de dados são usadas e que, caso estejam aglomeradas num único nó ou ainda espalhadas de maneira

[1] LLVM no passado foi uma sigla para *Low-Level Virtual Machine*, sendo que hoje o nome LLVM não é mais considerado uma sigla.

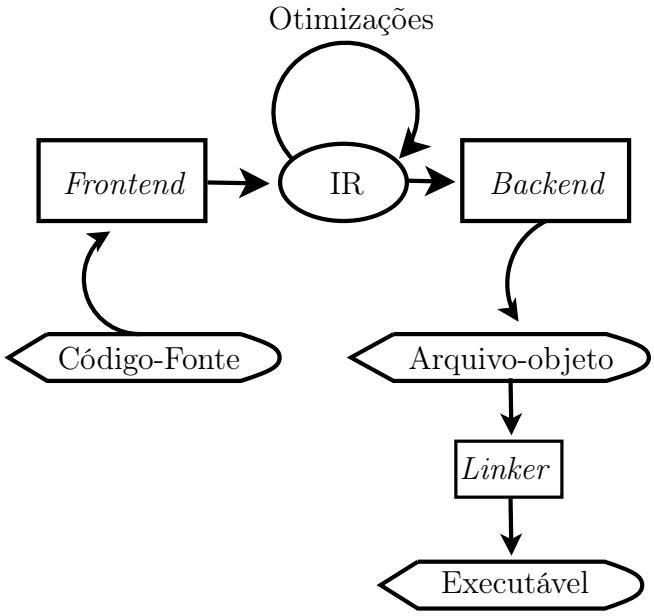


Figura 3.2: Etapas da compilação de um código-fonte no LLVM.

não otimizada, causarão impacto considerável no desempenho das aplicações. Logo, faz-se necessário avaliar na IR os acessos de escrita e leitura na memória (`loads/stores`), as condições de controle dos laços (visto que os consideramos trechos quentes) e também inferir a partir dessas avaliações o reuso e o tamanho de tais estruturas, bem como o conjunto de páginas nas quais elas estão mapeadas.

Uma idéia simplificada da otimização SPM é apresentada no diagrama da Figura 3.3 - os detalhes de cada etapa são apresentados nas subseções a seguir. Dado que o passo tem escopo de função (executa em todas as funções de uma unidade de compilação), temos no diagrama uma determinada função na qual a otimização será executada; tal função é denominada N no exemplo. O passo então tem início, testando primeiramente se a função N é a `main()`; em caso positivo, insere chamadas de inicialização e término da biblioteca `hwloc`. Após isso, o passo busca laços; ao encontrar um laço dentro da função N, todas suas instruções são percorridas em busca de `loads/stores` da IR do LLVM. A partir de então, inicia-se uma sequência de passos auxiliares, como `ReduceIndexation` para simplificar o `load/store` em análise, `RelativeExecutions` para determinar a relação do `load/store` com o(s) laço(s) em que está inserido; finalmente, o passo `RelativeMinMax` determina os índices máximo e mínimo de acesso à memória do `load/store` sendo avaliado. Supondo que todas as etapas tenham sido concluídas com êxito, uma chamada para a função de migração (`spm_call()`) é criada e guardada numa lista; quando o passo termina a análise de todos os `loads/stores` internos aos laços da função N, insere as chamadas de `spm_call()` em locais previamente calculados no código. Na subseção seguinte, todos os passos auxiliares são detalhados.

3.1.1 Detalhes dos passos auxiliares implementados no LLVM

A abordagem do otimizador do LLVM recai na carga de bibliotecas, que representam os módulos responsáveis por determinadas transformações/análises da IR. Poderíamos

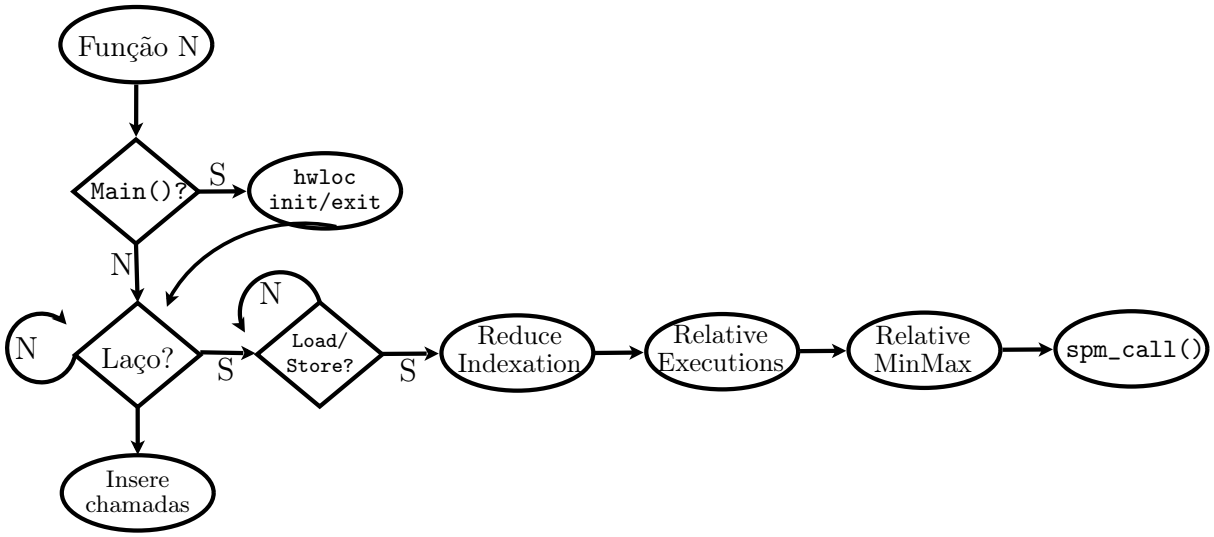


Figura 3.3: Diagrama de etapas da otimização SPM.

ter condensado todas as etapas implementadas no compilador num único módulo, mas levando em conta o foco em orientação a objetos do LLVM, e os benefícios da divisão organizada do código em entidades especializadas distintas, optamos pela construção de 8 bibliotecas: `SelectivePageMigration` (biblioteca principal), `GiNaCExpressionWrapper`, `SymPyInterface`, `LoopInfoExpression`, `RelativeExecutions`, `ReduceIndexation`, `GetWorkerFunctions` e `RelativeMinMax`. Detalharemos cada uma delas, a começar pelas bibliotecas auxiliares nesta subseção, culminando no funcionamento da transformação principal na subseção a seguir. Cabe aqui uma nota: todos os módulos possuem um parâmetro de invocação que habilita um modo de depuração verboso, com o intuito de facilitar as análises de causas de possíveis erros.

`GiNaCExpressionWrapper`: Uma vez que nossas análises recaem em avaliações de expressões numéricas, como produtos, somas e por vezes somatórios aninhados, necessitamos de ferramental de manipulação algébrica. Para tanto, com o objetivo de poder representar estruturas algébricas diretamente no código em C++, fazemos uso da biblioteca `GiNaC` [7], que possui tipos abstratos de dados capazes de prover operações algébricas simples de modo direto, como somas e produtos. A implementação do módulo `GiNaCExpressionWrapper` portanto representa uma camada de acesso ou *wrapper* para a biblioteca `GiNaC`. Tal camada contém principalmente métodos de checagem de operações e tipos, por exemplo podendo avaliar se determinada expressão é uma soma de números reais ou um produto de inteiros.

`SymPy`: Ainda que a biblioteca `GiNaC` permita manipulação direta de tipos algébricos no código, ela não cobre avaliações simbólicas mais sofisticadas, como resolução de somatórios; para esse fim contamos com a biblioteca de matemática simbólica `SymPy` [8]. A biblioteca `SymPy` é implementada em Python - nosso módulo `SymPyInterface` atua como interface Python, permitindo por exemplo a conversão de expressões descritas nos mode-

los da `GiNaC` em objetos de Python a serem avaliados pela `SymPy`. O uso concomitante dessas 2 bibliotecas garante o conjunto necessário de rotinas de manipulação algébrica para todos os outros módulos implementados.

`LoopInfoExpression:` Visto que nossa abordagem considera laços como trechos quentes dos programas, faz-se necessário prover um conjunto de avaliações desses - em especial pois é a partir de informações como os valores inicial e final da variável de indução e o passo (*step)* dessa que podemos inferir o tamanho e reuso das estruturas de dados contidas nos laços, informações essenciais para realizarmos migrações de páginas de memória. O LLVM possui uma classe específica para obtenção de dados sobre laços, a qual é utilizada vastamente nas bibliotecas implementadas. Contudo, foi necessária a implementação de uma extensão dessa biblioteca, denominada `LoopInfoExpression`, que apresenta métodos para se obter a variável de indução de um laço ou checar se determinado valor é de fato a variável de indução, por exemplo. Além desses, um dos mais importantes métodos implementados está presente nessa classe: `getLoopInfo`, que é capaz de determinar a variável de indução de um laço, bem como seus valores inicial e final, e seu passo (*step*). Tal método se baseia nas seguintes etapas:

1. Uma avaliação da instrução de salto na saída dos laços (*branch instruction*) é realizada, e então determina-se o operador de comparação da condição de controle avaliada - são válidos os operadores $\leq, <, >, \geq$ e $==$.

2. Analisa-se então a invariância[2] dos termos de tal condição de comparação em relação ao laço, para se determinar qual a variável de indução e qual o valor sendo comparado com essa - assumimos condições de controle do tipo `LHS` $*$ `RHS`, em que $*$ é um operador da lista exibida no item anterior, `LHS` ou `RHS` é a variável de indução e o termo restante é o valor final dessa, invariante ao laço.

3. Nesse ponto temos a variável de indução e sua condição final (o elemento invariante do item anterior). Avaliando-se o pré-cabeçalho do laço (*loop preheader*), o valor inicial da variável de indução é encontrado.

4. Finalmente uma avaliação do passo (*step*) do laço é realizada: através de um mecanismo de casamento de padrões, uma expressão genérica/arbitrária (*wildcard*) do tipo `indvar` $+ X$ (em que `indvar` é a variável de indução encontrada na etapa anterior e X um valor válido para o passo do laço, por exemplo um valor numérico) é testada contra a expressão de entrada do corpo do laço, usualmente um *phi node*. Se houver o casamento, têm-se o *step* do laço.

`RelativeExecutions:` No que toca avaliação de reuso dos dados, a biblioteca responsável pela etapa mais fundamental é denominada `RelativeExecutions`; é nela que se calcula o número estimado de execuções de um bloco básico interno a um laço. Note que nossa abordagem para atenuar o problema de contenção de memória em arquiteturas ccNUMA

[2]Entende-se aqui um valor invariante em relação a um laço como aquele que não é modificado em nenhuma etapa dentro do laço, inclusive no cabeçalho desse.

faz uso de migração de páginas de memória - operação custosa, que deve apenas ser realizada se for pertinente. Nossa heurística para estimar a eficácia da migração recai no reuso e tamanho de estruturas de dados; assim, a acurácia da biblioteca `RelativeExecutions` se faz essencial para que a heurística seja precisa. Tal biblioteca apresenta o método `getExecutionsRelativeTo`, que calcula uma estimativa para o número de iterações de um laço: tal estimativa pode ser um valor ou uma função do número de iterações do laço de nível mais alto, em caso de aninhamento.

Seja um laço com variável de indução i, tal que o valor mínimo possível para i é L, o máximo é U e o passo (*step*) de i é S. Consideramos então que o número estimado de iterações do laço, assumindo uma função de incremento do tipo $i = i \pm S$, é dado pelo somatório 3.1.

$$\left\lceil \frac{1}{S} \sum_{i=L}^{U} 1 \right\rceil \tag{3.1}$$

A idéia por trás do somatório 3.1 é simples: a soma representa o total de iterações de um laço cuja variável de indução é incrementada/decrementada em uma unidade; o fator $1/S$ tem como objetivo particionar o espaço de iterações de acordo com o tamanho do salto no incremento. Para $S = 2$ por exemplo, o fator $1/2$ reduz o espaço de iterações pela metade, já que o incremento é dobrado em relação à unidade. O módulo `RelativeExecutions` portanto faz uso de um somatório do tipo apresentado para estimar o número de iterações de laços. Avaliando o grau de aninhamento de um dado laço, o método já mencionado `getExecutionsRelativeTo` calcula os somatórios levando em conta as relações de tal laço com os de maior nível (quando houver aninhamento), obtendo no final uma expressão estimada que reflete a estrutura dos laços e cujos valores componentes podem estar disponíveis apenas em tempo de execução - tal método faz uso da biblioteca `SymPy` para o cálculo dos somatórios.

`ReduceIndexation`: A biblioteca `ReduceIndexation` lida com a simplificação das instruções de `load` e `store` da IR do LLVM, uma tarefa necessária para que nosso passo seja preciso em suas estimativas. Uma vez que nossa abordagem leva em conta o tamanho de estruturas de dados, e na migração se faz necessário determinar a página inicial e o deslocamento do acesso à memória para cálculo do número total de páginas a serem migradas, é essencial que possamos representar os acessos às estruturas de dados como `endereço inicial` + *`offset`*. Justamente essa é a funcionalidade do módulo `ReduceIndexation`, que avaliando as instruções de `load` e `store` e a instrução especial de cálculo de endereços `GetElementPtr`[3], representa os acessos na forma requerida, provendo essa informação para todos os outros módulos.

`GetWorkerFunctions`: O módulo `GetWorkerFunctions` realiza um trabalho simples, mas fundamental para o mecanismo de travamento de *threads* (a ser detalhado na seção 3.3).

[3] A instrução `GetElementPtr` realiza aritmética de endereços de memória na IR do LLVM, não realizando qualquer acesso à memória.

Tal módulo percorre todas as funções de um arquivo-objeto e determina quais delas são *worker functions*, isto é, funções invocadas pela chamada `pthread_create`. Assim, a biblioteca `GetWorkerFunctions` é responsável por permitir mais tarde que o mecanismo de travamento de *threads* tenha ciência se uma determinada função é executada numa *thread* disparada por uma chamada de `pthread_create`.

`RelativeMinMax:` Finalmente, o último dos módulos auxiliares é `RelativeMinMax`, que realiza um trabalho de certa complexidade, fortemente embasado na biblioteca GiNaC. Tal módulo avalia expressões em busca de seus valores mínimo e máximo - tal tarefa é fundamental na obtenção do tamanho de estruturas de dados, já que esse é derivado através da diferença entre os valores máximo e mínimo da expressão que determina o acesso à estrutura. Encontrar máximos e mínimos de expressões genéricas não é trivial, pois por vezes tais expressões são compostas por sub-expressões conectadas por operações de multiplicação e exponenciação, por exemplo. A idéia fundamental é que o máximo e mínimo de uma constante é a própria constante, e de 2 constantes podem ser calculados simplesmente por comparação; assim, expressões complexas devem ser decompostas em "átomos" numéricos (constantes).

Esse processo de decomposição varia segundo os operadores presentes na expressão e deve ser avaliado numa base caso-a-caso, com intuito de ser o mais generalista possível e avaliar o maior número de casos. Além dessa característica de individualidade de expressões, temos o fator de relativização da análise: no nosso caso, as análises devem ser relativas ao conjunto de símbolos não-resolvidos em tempo de compilação, como variáveis cujos valores são entradas do usuário ou resultado de outros cálculos no programa. Em especial, a variável de indução de um laço tende a aparecer em expressões de máximo e mínimo, já que no nosso caso avaliamos estruturas de dados internas a laços no geral.

Uma vez que todos os módulos auxiliares foram detalhados, na subseção a seguir trataremos do módulo principal e apresentaremos um exemplo da execução da análise, contemplando todas as suas etapas.

3.1.2 O módulo `SelectivePageMigration` e um exemplo de sua aplicação

O módulo principal de transformação é o único que de fato modifica a IR - até agora, todas as bibliotecas auxiliares citadas apenas realizam análises baseadas na IR. Tal módulo se comporta como um *Function Pass* do LLVM, ou seja, uma biblioteca cujo escopo de avaliação são as funções; o módulo é invocado uma vez por arquivo-objeto, e percorre todas as suas funções, uma por vez. Além do parâmetro de invocação para habilitar o modo de *debug* verboso, esse módulo possui mais 2 parâmetros de invocação: o `spm-pthread-function`, que limita o escopo do módulo para uma determinada função (unicamente), e o parâmetro `spm-thread-lock`, que habilita o mecanismo de travamento de *threads*.

A biblioteca `SelectivePageMigration` primeiramente avalia se a função sendo analisada é a `main` - em caso positivo, são inseridas duas chamadas especiais, de inicialização e encerramento da biblioteca `hwloc` [18], logo no início da função `main` e imediatamente

antes de sua instrução de `return`, respectivamente. O uso da biblioteca `hwloc` e o detalhamento de suas funções de inicialização e término são devidamente comentados na seção 3.2. Além disso, se o parâmetro `spm-thread-lock` for utilizado, uma chamada para uma função auxiliar de travamento de *threads* é inserida no início das funções que são executadas em *threads* - a identificação de quais são essas funções é realizada pelo passo auxiliar já descrito, `GetWorkerFunctions`. O conceito de travamento de *threads* será detalhado na seção 3.3.

Uma vez que esses casos particulares tenham sido tratados, o módulo avalia todos os blocos básicos da função sendo analisada em busca de cabeçalhos de laços (*loop headers*) - ao encontrar um laço, são avaliadas todas as instruções de todos os blocos básicos que o compõem, em busca de instruções do tipo `load` e `store`. Para cada instrução desse tipo, um método auxiliar - denominado `generateCallFor` - é invocado para realizar uma série de análises e gerar, se todas suas etapas forem bem-sucedidas, uma chamada para a função/heurística de migração relacionada à estrutura de dados acessada pelo `load` ou `store` em questão; tal método é bastante relevante para a otimização SPM e será detalhado a seguir. Finalmente, depois de todas as instruções de todos os laços da função terem sido percorridas, o módulo `SelectivePageMigration` insere todas as chamadas para a função de migração de uma única vez nas posições correspondentes, e segue para a próxima função do arquivo-objeto.

O método `generateCallFor` é realmente o componente central do passo de otimização, utilizando todas as bibliotecas auxiliares já comentadas conjuntamente numa sequência de análises baseadas nas seguintes etapas:

1. Em primeiro lugar, o método certifica-se de que a instrução a ser analisada é um `load` ou `store` - em caso positivo, a biblioteca auxiliar `ReduceIndexation` é invocada para reduzir tal instrução para a forma geral `endereço inicial` + ***`offset`***.

2. O próximo passo é estimar a quantidade de vezes que tal instrução é executada - para tanto o módulo `RelativeExecutions` é usado. Além disso, todo o conjunto de laços exteriores à instrução (mais de um nos casos de laços aninhados) é avaliado para que a decisão sobre o local de inserção da possível chamada à função de migração seja tomada posteriormente.

3. Uma primeira avaliação da dependência de instruções no grafo de controle (CFG, de *Control Flow Graph*) da função é então realizada - esse ponto é fundamental para a determinação das possíveis posições de inserção da chamada à função de migração, e será detalhado logo a seguir.

4. Através da biblioteca `RelativeMinMax` uma análise do intervalo de acesso à estrutura de dados (ou ainda, análise dos valores mínimo e máximo do ***`offset`*** de acesso à estrutura) é realizada; as expressões de mínimo e máximo são montadas e confrontadas com uma análise de dependência de instruções no CFG da função para mais uma vez permitir a posterior inserção da chamada à heurística num local adequado.

5. Finalmente, tendo sido todas as etapas anteriores bem-sucedidas, nesse ponto a chamada à função de migração é criada (na forma de uma instrução `call` da IR

do LLVM) e sua posição determinada; tal chamada é adicionada numa lista de instruções a serem inseridas na IR no momento de inserção de instruções.

Nota-se que as etapas acima estão sujeitas a uma série de resultados de execuções de módulos auxiliares, bem como a uma análise de dependência de instruções no CFG da função que se mostre favorável. Nesse sentido, podemos entender que as etapas foram bem-sucedidas se todas as análises auxiliares puderam ser completadas com êxito, obtendo retornos factíveis no âmbito de nossa otimização proposta. Por exemplo, a biblioteca `RelativeMinMax` possui algumas restrições, dada a generalidade de seu escopo (conforme já discutido); assim, se durante a execução do método `generateCallFor` tal biblioteca, ao ser invocada, esbarrar numa restrição e não concluir sua análise com sucesso, todo o método `generateCallFor` falha. Portanto, tal método é bastante correlacionado com todos os outros módulos apresentados e seguindo naturalmente uma lei de probabilidades independentes, o mais sujeito à falhas. No caso de uma falha nesse método, nenhum dano ao desempenho do programa sendo compilado é incorrido - simplesmente deixa-se de otimizar um trecho supostamente interessante para obtenção de ganho de desempenho na aplicação.

Outra situação bastante delicada no que toca a inserção de instruções numa IR é a dependência entre as instruções no CFG da função. Nossa proposta, delineada no exemplo da Figura 3.1, recai na inserção de uma chamada à uma função/heurística de migração num ponto anterior ao uso da estrutura de dados cujas páginas devem ser migradas. As implicações dessa inserção são severas do ponto de vista do CFG, em especial se levarmos em conta que consideramos laços como trechos quentes e estruturas internas aos laços como favoráveis à migração. Note que se a inserção da chamada ocorrer imediatamente antes do uso da estrutura de dados, não há problemas de dependências, portanto esse é nosso objetivo. Contudo, não podemos inserir chamadas logo antes de um `load` ou `store` contidos num laço, pois apesar de a *syscall* de migração de memória ser inócua caso as páginas já estejam no local-alvo, há uma pequena sobrecarga em sua chamada, que pode ser potencializada no caso de múltiplas chamadas.

Nesses casos, que são frequentes, nossa abordagem é realizar a inserção da chamada imediatamente antes do laço mais externo que contém o `load` ou `store` - daí a importância da segunda etapa do método `generateCallFor`. No entanto, nem sempre é possível utilizar-se desse expediente: no caso de aninhamento de vários níveis, pode-se chegar a uma situação em que a localização da inserção deveria ocorrer num ponto muito anterior ao uso da estrutura de dados, mas há inviabilidade de tal inserção devido à dependência de instruções.

Por exemplo, símbolos usados no cálculo do *`offset`* de acesso do `load` ou `store` podem ter sua declaração num ponto após o local escolhido para a inserção da chamada, de modo que para o escopo da chamada, esses símbolos não serão válidos - por exemplo, isso aconteceria se a variável `end` fosse declarada e tivesse seu valor alterado dentro do laço mais externo (cuja variável de indução é `i`) na Figura 3.1. Na ocorrência de situações desse tipo, o passo irá adicionar a chamada no ponto mais próximo do ideal em que não haja conflitos de dependência; voltando ao exemplo, se a variável `end` fosse declarada de fato dentro do laço mais externo, o local escolhido para a inserção da chamada à função

de migração seria imediatamente após tal declaração.

Finalmente, na geração da chamada da função de migração de páginas, o reuso é uma informação relevante. O conceito de reuso adotado nesse trabalho é simples: consideramos reuso de uma estrutura o número de vezes que a acessamos. Ou seja, reuso representa a quantidade de vezes que o `load` ou `store` para tal estrutura é executado. Tal informação, juntamente do tamanho estimado da estrutura de dados, compõem as variáveis utilizadas na heurística para determinação se a migração de páginas é vantajosa ou não, a ser detalhada na seção seguinte.

Exemplo de execução do módulo `SelectivePageMigration`: Uma vez que o funcionamento do passo de otimização foi detalhado, apresentaremos aqui um exemplo concreto da sua execução - nosso foco será no método `generateCallFor`. A Figura 3.4 apresenta um código real usado no *benchmark* `PartitionStringSearch`, detalhado na seção 4 - note que o código da Figura 3.1 é uma versão simplificada do código do *benchmark*.

```
1  //TID is the thread number and its range is 0..num_threads-1
2  const long MY_REGION = global_V_array_size/num_threads;
3  const long MY_START  = TID * MY_REGION;
4  const long MY_END    = (TID + 1) * MY_REGION;
5
6  //some code here initializing str_size and *pattern...
7  //*pattern is the pattern to be matched against
8  //str_size is the size of the subword tested
9  //glob_num_queries is the number of runs
10
11 for (i=0; i<glob_num_queries; i++) {
12   num_occurrences=0;
13
14   for (j=MY_START; j<MY_END-str_size+1; j++) {
15     int success=1;
16
17     for (k=0; k<str_size; k++)
18       if (global_V[j+k] != pattern[k]) {
19         success=0;
20         break;
21       }
22
23     if (success)
24       num_occurrences++;
25   } //for_j
26 } //for_i
```

Figura 3.4: Trecho crítico do código do *benchmark* `PartitionStringSearch`: o acesso ao *array* global `global_V` causa contenção e nossa otimização é capaz de resolver esse problema.

Tal *benchmark*, que implementa um algoritmo de busca de padrões em *strings*, sofre de contenção de memória. O acesso ao *array* `global_V` prejudica o desempenho da aplicação, pois esse é inicializado numa única *thread*, logo todas as suas páginas são exclusivamente alocadas nos bancos de memória conectados ao nó em que a *thread* estava sendo executada no momento da alocação. No nosso caso de teste o *array* foi inicializado de modo a ocupar

4GB de memória RAM.

Ao compilar o código do *benchmark* com nossa otimização habilitada, o módulo `SelectivePageMigration` percorre todas as suas funções, e quando encontra o `load` relacionado ao acesso `global_V[j+k]`, invoca o método `generateCallFor`. O trecho de código da IR que realiza esse acesso é dado na Figura 3.5.

```
1 %add26 = add nsw i64 %j, %k
2 %5 = load i8** @global_V, align 8
3 %arrayidx27 = getelementptr inbounds i8* %5, i64 %add26
4 %6 = load i8* %arrayidx27, align 1
```

Figura 3.5: Trecho da IR do LLVM referente ao `load` que realiza o acesso ao *array* `global_V`.

A instrução `%6` é justamente a que realiza o acesso em questão - descrevemos a seguir os resultados das etapas de execução para tal acesso:

1. Na primeira etapa, o módulo `ReduceIndexation` encontra os valores de endereço inicial=`%5` e `offset`=`%add26`, o que representa de fato o acesso ao *array* na forma esperada.

2. A seguir, a biblioteca `RelativeExecutions` encontra os seguintes valores inicial, final e *step* da variável de indução `k`: `0, str_size-1` e `+1`. Com base nesses valores e nos valores encontrados para os laços externos (omitidos aqui por clareza), temos o reuso: $(-$`str_size`$\hat{}2) *$ `glob_num_queries` $+$ `glob_num_queries` $*$ `str_size` $-$`glob_num_queries` $*$ `MY_START` $*$ `str_size` $+$`str_size` $*$ `glob_num_queries` $*$ `MY_END`.

3. A análise do CFG é realizada e tal procedimento encontra um problema: por ser uma estrutura global, todo acesso a `global_V` incorre num `load` imediatamente antes do acesso - assim, o módulo define a inserção desse `load` (que é idêntico à instrução `%5` na Figura 3.5) antes de nossa chamada para a função de migração. Essa funcionalidade permite realizar a migração de páginas de estruturas globais.

4. O módulo `RelativeMinMax` encontra os seguintes valores mínimo e máximo para o `offset` de acesso ao *array* `global_V`: `MY_START` e `MY_END-1`.

5. Finalmente a chamada para a função de migração é criada, e seu ponto de inserção é determinado como sendo no local da IR que corresponde ao ponto imediatamente antes da linha 11 da Figura 3.4. A chamada fica: `call void @spm_call(%load, %start, %end, %reuse)`, com `%load` sendo uma instrução idêntica à `%5` da Figura 3.5, `%start` e `%end` correspondendo respectivamente aos valores `MY_START` e `MY_END - 1` e `%reuse` sendo o valor apresentado no item 2 acima.

Todos os valores e resultados do exemplo acima apresentado foram baseados em análises com depurador e com uso dos modos de depuração verboso dos módulos, portanto salvo por simplificações na nomenclatura dos valores da IR, os valores são reais. Na subseção a seguir, detalhamos a função de migração e o critério de comparação.

3.2 Heurística de migração de páginas

No que toca migração de páginas de memória, a idéia de realizar sua movimentação transparente recai no uso de *syscalls* do sistema operacional; tal tarefa envolve atualização da tabela de páginas e outras operações sofisticadas, devendo ser realizada pelo *kernel* do sistema operacional. Duas chamadas de sistema para esse fim estão presentes no Linux: as funções `move_pages` e `mbind`. Ambas as chamadas de sistema efetuam movimentação de páginas: a *syscall* `move_pages` realiza a movimentação de um vetor de páginas para o nó selecionado, ao passo que a função `mbind` - originalmente concebida para atribuir uma política de alocação de memória para um conjunto de páginas - permite, através de uma *flag*, que a movimentação de um conjunto de páginas seja efetuada.

No entanto, ambas *syscalls* são diretamente atreladas a um único sistema operacional e portanto não apresentam qualquer grau de portabilidade; além disso, a sintaxe de ambas recai em alguma complexidade que adicionaria alguns passos a mais na geração da chamada à função de migração por parte do compilador. Alternativamente, optamos pelo uso da biblioteca de portabilidade de *hardware* denominada `hwloc` [18], cuja proposta principal é prover uma abstração no acesso à informações da topologia do sistema, como dados do processador, memória e detalhes da arquitetura. Tal biblioteca provê também abstrações para funções de migração de páginas de memória. No caso do Linux, a função de alto-nível da `hwloc` é mapeada para a *syscall* `mbind` - além disso, a biblioteca contempla funções para se determinar o nó em que o trecho de código sendo executado se encontra, permitindo assim a migração para o local correto. Uma vez que a biblioteca é portável, em outros sistemas operacionais sua implementação pode realizar um mapeamento da camada de migração de páginas para uma chamada de sistema específica; assim, temos o bônus da portabilidade ao usar `hwloc`. Não fizemos experimentos em outros sistemas operacionais, portanto não avaliamos a disponibilidade de migração de páginas em outros sistemas pela biblioteca; contudo, com base numa avaliação do código-fonte da biblioteca, acreditamos que o suporte para outros sistemas é por ora inexistente. Uma ressalva quanto ao uso da biblioteca: por fazer uso da *syscall* `mbind`, seu desempenho tende a ser levemente menor do que o de `move_pages` - num experimento que realizamos, obtivemos um desempenho em torno de 8% menor com `hwloc`.

Ainda assim, o uso da biblioteca se faz bastante interessante. Além da portabilidade e abstração de alto-nível oferecidas por `hwloc`, um importante aspecto da otimização foi coberto ao optarmos pelo uso de tal biblioteca. Dado que nossa proposta é avaliar se a migração faz-se ou não vantajosa através do reuso da estrutura de dados e do seu tamanho, precisamos de um meio para obter, em tempo de execução, o tamanho da memória *cache* dos processadores para efeito de comparação na heurística. A biblioteca `hwloc` possui exatamente essa funcionalidade, e através dela podemos obter, independentemente do *hardware* ou da arquitetura em que estamos, o tamanho das memórias *cache* dos processadores, tanto de nível L1 quanto subsequentes. Uma questão importante em se tratando da comparação do tamanho da estrutura de dados com a memória *cache* é se devemos somar o tamanho das *caches* de todos os níveis ou utilizar apenas o último nível, por exemplo.

Tal discussão recai em primeiro lugar no foco da heurística: se optarmos por um

grau maior de agressividade na otimização, a tendência é migrar mais coisas, incorrendo num risco de geração de sobrecarga por migração de estruturas não-propícias a serem relocadas. Por outro lado, se nosso enfoque é mais conservador, devemos migrar apenas o que garantimos ser vantajoso. Ainda, um fator importante nessa escolha é a política de inclusão das *caches*, num caso de *caches* multiníveis (algo padrão nas arquiteturas atuais). *Caches* inclusivas apresentam replicação de dados entre os níveis, isto é, se um dado está no nível L1, garantidamente está nos níveis abaixo (pode-se haver um grau parcial de inclusão, por exemplo no caso de um dado na L1 estar na L3, mas não na L2). *Caches* não-inclusivas (ou exclusivas) apresentam o comportamento contrário: dados que estão num nível não estão presentes nos outros níveis; nesse caso, o tamanho total da *cache* é maior. Optamos aqui por considerar a soma de todos os níveis de memória *cache* no critério de comparação da heurística - tal opção se deve ao fato de nosso sistema (a ser detalhado no capítulo 4) possuir *cache* não-inclusiva, e também ao nosso foco numa heurística com maior liberdade, incorrendo nos já mencionados riscos de sobrecarga por migrações inapropriadas.

Ainda em se tratando da biblioteca `hwloc`, deve-se realizar procedimentos de inicialização e finalização de uma estrutura de dados opaca que define a topologia do sistema em uso. Tais procedimentos já foram citados na seção anterior: quando ocorre a avaliação da função `main` por parte do compilador, uma chamada para a função `spm_init` é inserida em seu início, assim como uma chamada para o procedimento `spm_end` é inserida em seu término, logo antes da instrução de retorno. Tais funções realizam a inicialização e destruição da estrutura de topologia da `hwloc` - tal estrutura é declarada como uma variável global e é utilizada na chamada de migração da biblioteca `hwloc`. Ainda, é na função `spm_init` que se realiza a análise da memória *cache*, obtendo-se seu tamanho e disponibilizando esse dado na forma de uma variável global, além da inicialização de uma máscara com todos os processadores presentes no sistema, a ser utilizada no mecanismo de travamento de *threads*, descrito na seção a seguir.

Uma vez definidos os métodos de migração de páginas e obtenção do tamanho da memória *cache*, derivamos a função de migração, que avalia um critério para realizar a migração apenas em casos vantajosos. Nossa abordagem foi a de embutir tal função de migração num arquivo-objeto (bem como as funções `spm_init` e `spm_end`), de modo que o passo de otimização, no momento da compilação, desempenha suas análises e insere o código de chamada para tais funções, devendo tal arquivo-objeto, que contém as definições das funções, ser ligado ao programa pelo *linker*. Uma vantagem dessa abordagem é a maior simplicidade de se inserir código na representação intermediária através de um passo de transformação no compilador. Além disso, é trivial realizar modificações no código-fonte em linguagem C que dá origem à função de migração (e ao arquivo-objeto), ao passo que seria bastante trabalhoso lidar com um caso de inserção do corpo completo de funções diretamente na IR. Uma desvantagem dessa abordagem é a necessidade de se ligar o código-fonte compilado ao nosso arquivo-objeto, sendo também necessária a instrumentação da função `main` para que a otimização SPM possa funcionar.

Nosso critério de comparação para decidir se a migração de páginas é ou não vantajosa compara o tamanho das estruturas com um *threshold* envolvendo o tamanho da *cache*, e seu reuso com outro *threshold* - o código apresentado na Figura 3.6 demonstra todo o

processo de migração (com alguns detalhes de implementação omitidos em prol da clareza na exposição).

```
1  void spm_call (void *array, long start_address,
2                 long end_address, long reuse) {
3
4    if (reuse > reuse_threshold &&
5        end_address - start_address >
6        cache_threshold * cache_size) {
7
8        node = hwloc_Get_Node_Running();
9        hwloc_Migrate_Pages(array + start_address,
10                           end_address - start_address, node);
11
12   }
13 }
```

Figura 3.6: Trecho de código que exibe a função/heurística de migração: critérios de comparação do tamanho das estruturas de dados e de seu reuso são utilizados.

Na Figura 3.6, os valores `array`, `start_address`, `end_address` e `reuse` são provenientes do passo de otimização, que através das já comentadas avaliações deriva tais dados. O valor `cache_size` é uma variável global obtida através da biblioteca `hwloc`, cujo cálculo é realizado uma única vez, na função `spm_init`, conforme já explicado. Finalmente, `cache_threshold` e `reuse_threshold` são os limitantes/*thresholds* usados em nosso critério de comparação, para heuristicamente determinar a viabilidade de se realizar a migração das páginas de memória da estrutura apontada por `array`, com perspectiva de obtenção de ganho de desempenho geral no programa. Tais *thresholds* são calculados empiricamente através de *microbenchmarks* - basicamente rodamos uma série de experimentos e ajustamos seus valores até um ponto em que obtivemos bom desempenho com o critério de comparação, ou seja, migrações sendo efetuadas quando trazem bom desempenho, e deixando de ser realizadas quando do contrário. Por ser um critério experimental, baseado em rodadas de um número finito de *benchmarks*, acreditamos que não é possível contemplar todos os casos de programas a serem otimizados por *Selective Page Migration* - uma discussão sobre possíveis alternativas aos *thresholds* calculados empiricamente ocorre na seção de trabalhos futuros, no capítulo 5.

Nesse ponto, temos um passo de otimização que instrumenta códigos-fonte, inserindo chamadas para uma função/heurística de migração, e um arquivo-objeto que contempla a definição dessa, bem como de funções auxiliares e variáveis globais necessárias para o correto funcionamento da otimização. Contudo, nem sempre apenas migrar páginas de memória resolve o problema de contenção, pois sistemas operacionais modernos migram *threads* entre núcleos de processamento, podendo incorrer em queda de desempenho mesmo após nossa otimização migrar páginas de modo eficiente. Na seção a seguir apresentamos um mecanismo que busca contornar essa situação, através do travamento de *threads* em nós de processamento.

3.3 Mecanismo de travamento de *threads*

Nos sistemas operacionais modernos, um traço bastante consolidado é o processamento multitarefa, em que variados processos (e *threads*) executam "ao mesmo tempo", isto é, a impressão é de simultaneidade, mas na verdade os processos são escalonados sequencialmente para execução em janelas temporais. Esse método de execução de processos e *threads* possui como principal vantagem a impressão para o usuário final de que a execução de múltiplos programas ocorre simultaneamente; de fato, com o avanço dos *multicores* um certo conjunto de processos pode executar ao mesmo tempo. Contudo, numa situação de execução de programas de computação de alto desempenho, com inúmeras *threads*, há de fato a interrupção na execução tanto do processo em questão quanto de vários processos do próprio sistema operacional, como *daemons* e aplicativos de interface gráfica, por exemplo.

Tal interrupção, do ponto de vista do usuário, é imperceptível no geral. Contudo, do ponto de vista do desempenho de aplicações científicas, em especial em arquiteturas ccNUMA, seu impacto pode ser bastante desastroso. Isso porque muitas vezes uma *thread* é interrompida e volta a executar a seguir em outro nó. Como nesse trabalho migrações de páginas são propostas para atenuar problemas de alocação de memória, uma vez que uma *thread* salta para outro nó, ela se afasta dos dados que, via nosso passo de otimização por exemplo, foram situados em bancos de memória locais à ela. Assim, o escalonamento de *threads* pode prejudicar o desempenho de aplicações, em especial quando essas dependem bastante de dados na memória - tais aplicações são conhecidas como *memory bound*. Um experimento realizado com o *benchmark* `Cholesky` (a ser detalhado no capítulo 4), que implementa um algoritmo baseado na decomposição Cholesky de matrizes, pode ser visto na Figura 3.7.

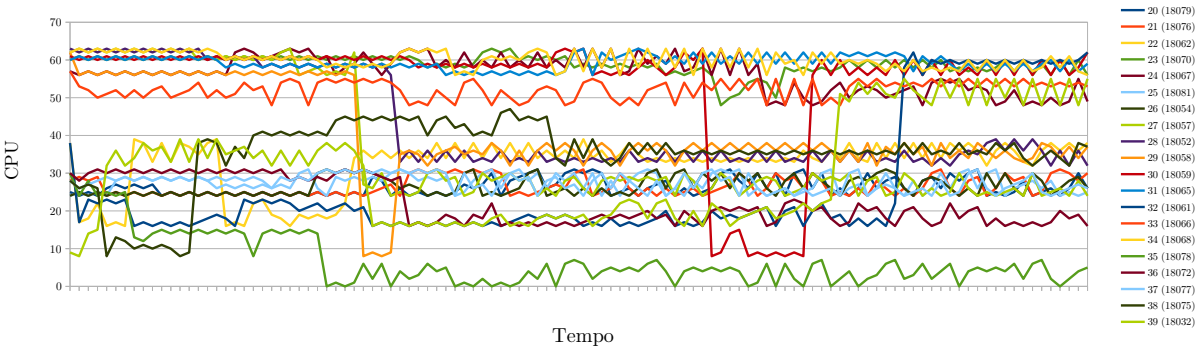


Figura 3.7: Resultado do experimento que busca detectar saltos para CPUs distantes no escalonamento de *threads*.

Para a realização do experimento cujos resultados estão apresentados na Figura 3.7, implementamos um pequeno utilitário que monitora em qual CPU (ou núcleo de processamento) uma determinada *thread* (ou conjunto delas) está executando, e toda vez que há mudança de CPU, o aplicativo adiciona a última CPU em que *thread* esteve executando numa lista. Esse processo é realizado continuamente durante a execução de um programa

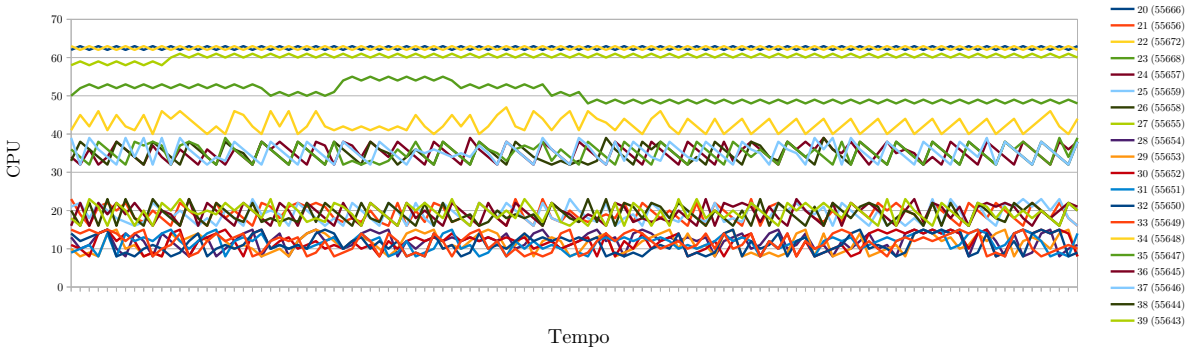


Figura 3.8: Resultado do experimento que avalia o impacto na movimentação de *threads* entre nós, aqui com o mecanismo de travamento ativado.

selecionado, e o resultado plotado na Figura 3.7 demonstra as movimentações entre núcleos de 20 *threads* do *benchmark* `Cholesky` - no gráfico, cada *thread* é representada por uma linha, e os *thread IDs* (TIDs) estão entre parênteses ao lado dos números de cada uma das 20 *threads*. Observa-se claramente que várias *threads* migram para núcleos "distantes", isto é, migram entre nós, prejudicando assim nossa otimização. Quanto maior o número de saltos nas linhas do gráfico, mais migrações entre nós foram efetuadas pelas *threads*.

Haja vista o problema de saltos das *threads* entre nós, implementamos uma modificação no passo que permite ativar ou desativar um mecanismo de travamento de *threads* em seus nós - tal mecanismo, denominado `SPM thread lock`, é provido novamente por funções da biblioteca `hwloc`. Para tanto, uma vez que o mecanismo esteja ativado, 2 instruções de chamada são inseridas nas funções que executam em *threads* (*worker functions*). Primeiramente, a chamada para a função auxiliar de travamento de *threads* é inserida como primeira instrução do bloco de entrada da *worker function*, determinando que a partir daquele ponto a *worker function* somente poderá ser escalonada para os processadores pertencentes a um determinado nó - o nó é escolhido via um algoritmo baseado em *round-robin*, de modo a manter um equilíbrio na distribuição de *threads* sem que haja um desbalanceamento, que poderia ocasionar uma situação com alguns nós tendo mais *threads* escalonadas do que outros. Em seguida, uma função auxiliar de liberação de *threads* é inserida logo antes da instrução de retorno da *worker function*, para que a partir dali, qualquer nova *thread* disparada pelo processo não fique presa ao escalonamento escolhido anteriormente - com isso, buscamos não interferir no agendamento de tarefas em novas fases do programa.

Uma vez com o mecanismo ativado, as *threads* ainda possuem liberdade para que sejam escalonadas num conjunto de núcleos, mas são travadas de modo que não saiam do nó, ou seja, são escalonadas apenas no subconjunto de processadores pertencentes a um único nó. A Figura 3.8 demonstra a mudança de comportamento das *threads* no caso do *benchmark* `Cholesky` - novamente, o experimento foi realizado com 20 *threads*, e nesse caso nota-se que apesar de haver saltos nas linhas do gráfico, esses são bastante pequenos, indicando escalonamento das *threads* em núcleos próximos, dentro de um mesmo nó. No

capítulo seguinte, os resultados experimentais demonstram o grande impacto positivo de tal mecanismo de travamento de *threads* na otimização SPM, bem como algumas de suas limitações.

Capítulo 4

Resultados Experimentais

Neste capítulo apresentamos os resultados experimentais obtidos no desenvolvimento da otimização SPM. Primeiramente, descrevemos com maiores detalhes 2 mecanismos de otimização de desempenho em arquiteturas ccNUMA que já foram brevemente explicados nos capítulos anteriores, e serão novamente expostos aqui pois os utilizamos nos experimentos como "concorrentes" de nossa abordagem, para efeito comparativo.

A seguir, descrevemos o principal conjunto de *benchmarks* utilizado, denominado *EasyBench*. São aplicativos cujo código-fonte é bastante similar, contudo o *kernel* de cada um reflete operações matriciais diferentes: temos uma versão de adição de matrizes, fatoração LU, decomposição Cholesky e uma versão "ingênua" de multiplicação matricial[1]. Finalmente, outros 2 *benchmarks* usados em nossas análises são apresentados: *partitionStringSearch*, uma aplicação que realiza busca de expressões em cadeias de caracteres, e *BucketSort*, uma implementação do conhecido algoritmo de ordenação *bucket sort*.

Apresentamos então a metodologia usada para os experimentos, bem como detalhes do sistema em que os executamos, e finalmente os resultados experimentais e suas análises.

4.1 Mecanismos avaliados comparativamente

No capítulo 2 descrevemos uma vasta gama de técnicas de otimização de múltiplos âmbitos, as quais têm como foco solucionar problemas de má distribuição de dados na memória em arquiteturas ccNUMA, no intuito de evitar queda no desempenho das aplicações. Tais abordagens apresentam graus variados de complexidade de implementação e utilização. Para o escopo desse projeto, buscamos abordagens factíveis, que não dependessem de *hardwares* especiais e pudessem ser executadas no sistema operacional Linux. Além disso, optamos por mecanismos mais atuais, por serem em geral tanto dotados de maior taxa de compatibilidade com aplicações genéricas quanto mais fáceis de se utilizar.

No artigo publicado [47] com resultados preliminares da otimização SPM, realizamos comparações com o *framework Minas* [48], já comentado no capítulo 2. Optamos por não realizar a comparação com esse *framework* novamente por dois motivos: primeiramente, ele exige modificações nos códigos-fonte e acreditamos ser mais justa uma comparação

[1] Optamos por realizar os experimentos com uma versão simples do algoritmo de multiplicação matricial pois queremos aqui avaliar algoritmos *memory-bound*, de modo que algoritmos mais sofisticados de produto de matrizes já contemplam melhorias e não se beneficiariam de nossa otimização.

da otimização SPM com mecanismos de caráter mais automático; também, ainda que no artigo nossa otimização estivesse num estágio um pouco menos finalizado do que agora, nossos resultados foram superiores aos coletados com uso da técnica Minas. O ganho de desempenho máximo encontrado por nossa otimização no artigo foi de até 4x, ao passo que Minas não obteve ganhos maiores do que 50%. Portanto, buscamos aqui novos competidores.

Nossa opção se deu por 2 mecanismos já comentados tanto na Introdução quanto no capítulo de Trabalhos Relacionados: a ferramenta `numactl` e a otimização disponível no sistema operacional Linux denominada *Automatic* NUMA *balancing*. A primeira abordagem, com o uso da ferramenta `numactl` consiste basicamente em se invocar o executável de uma dada aplicação através do `numactl`, de modo que sua memória seja alocada de forma distribuída através da modificação de sua política da alocação.

A política escolhida através do `numactl` foi do tipo *round-robin*, em que páginas de memória vão sendo alocadas de modo cíclico nó-a-nó, distribuindo totalmente o conjunto de dados do programa, e portanto aliviando drasticamente o problema de contenção. O grande porém dessa abordagem é que não há qualquer avaliação sobre o uso das páginas no momento da alocação, podendo tal mecanismo gerar uma situação em que as *threads* sempre acessam dados remotos, no pior caso. Dessa forma, esperamos que os resultados da nossa abordagem superem os encontrados com o uso da ferramenta `numactl`, já que realizamos uma análise para efetuar migrações de forma inteligente, buscando aliviar o problema de contenção no momento da execução dos programas.

O segundo "competidor" escolhido é mais recente do que a ferramenta `numactl` - denominado AutoNUMA, o mecanismo foi implementado no *kernel* do sistema operacional Linux no final de 2012 e início de 2013 [32], tendo se tornado mais popular a partir do final de 2013, com a adição de um parâmetro que permite ligar e desligar o AutoNUMA sem a necessidade de recompilar o *kernel*. Conforme já mencionado no capítulo 2, duas abordagens foram propostas como mecanismos de migração automática de páginas no Linux em arquiteturas ccNUMA, e experimentos com ambas foram realizados até que se optasse pela abordagem de Andrea Arcangeli. A subseção a seguir descreve alguns detalhes do funcionamento do AutoNUMA, e foi baseada numa apresentação do desenvolvedor de *kernel* Rik van Riel [50]. É importante notar que a abordagem do AutoNUMA atualmente está bastante consolidada e o mecanismo é referência em se tratando de otimização em âmbito de distribuição de mémoria em arquiteturas ccNUMA.

4.1.1 AutoNUMA: *Automatic* NUMA *balancing*

O mecanismo AutoNUMA funciona através de perfilamento dos acessos à memória por parte dos processos sendo executados. Periodicamente, conjuntos de páginas de memória associados aos processos têm suas permissões removidas, inclusive de leitura - em geral sequências relativamente grandes de páginas (256MB, ou 65536 páginas em arquiteturas x86 com páginas de 4KB) compõem esses conjuntos. Na tabela de páginas, elas ganham uma *flag* que indica, na ocorrência de um *page-fault* no acesso às páginas, que foram marcadas pelo mecanismo AutoNUMA. Uma vez que páginas estão sendo acessadas por *threads* em nós remotos, e levando-se em conta que a migração de páginas é um processo

custoso, elas são migradas apenas se forem acessadas 2x pela mesma tarefa, num mesmo nó remoto. Dados sobre o acesso às páginas remotas são mantidos por processo, de modo que um contador é incrementado toda vez que há *page-fault* causado por acesso a um dado em determinado nó, por parte de um processo - ou seja, há um contador por nó para cada processo.

Uma análise estatística é baseada nesses contadores: se um processo apresenta significativamente mais *page-faults* no exato nó em que está executando, então há uma situação ótima do ponto de vista do mecanismo AutoNUMA, e a migração de páginas não precisa ocorrer. Nesse caso, o perfilamento dos acessos à memória de tal processo diminui, de modo a diminuir a sobrecarga do mecanismo. Ainda, os contadores permitem avaliar se o número de *page-faults* ocorrendo num determinado nó são provenientes majoritariamente de um único processo (nesse caso temos *page-faults* privados), ou se múltiplos processos estão acessando o mesmo conjunto de dados na memória - esse último caso, de acesso compartilhado, é mais delicado de se otimizar, pois migrações podem ser injustas e otimizar apenas um subconjunto do total de processos acessando tais dados, prejudicando os demais.

O mecanismo AutoNUMA apresenta, além da migração de páginas de memória, o conceito de migração de tarefas: para um determinado processo P, avalia-se em qual nó a maior parte dos seus acessos à memória ocorre; a seguir, analisa-se todos os núcleos de tal nó em busca de um que esteja ocioso, ou ainda um núcleo cujo processo em execução, ao ser permutado de nó com o processo P, tenha um desempenho melhor (ou não muito pior) do que o atual. Nos casos em que a permutação de processos não leva a uma queda de desempenho, a troca desses processos entre os nós é realizada. A migração de tarefas apenas ocorre numa situação em que não causa desbalanço, do contrário poderia disparar um ciclo de migrações de tarefas e dados que acarretaria queda no desempenho do sistema.

Além do conceito de migração de processos/*threads*, o mecanismo AutoNUMA apresenta a idéia de *numa_groups*: processos que acessam um conjunto comum de páginas são marcados como agrupados, e tende-se a movê-los em conjunto e tentativamente mantê-los no mesmo nó de processamento - tal abordagem visa a conjunção de processos relacionados, ignorando por exemplos bibliotecas comuns do sistema (como a `glibc`), que naturalmente compartilham acessos com grande variedade de processos.

Finalmente, o mecanismo AutoNUMA faz uso de uma técnica denominada *pseudo-interleaving*, que atenua a queda de desempenho nos casos de aplicações com *memory footprint* muito grande que não cabem num único nó. Nesse caso, migrações de páginas com acessos compartilhados entre nós apenas ocorrem do nó que está mais sobrecarregado com destino aos outros nós. Numa situação de estabilidade, as migrações de páginas compartilhadas deixam de ser realizadas, pois um equilíbrio tende a ser atingido. Note que migrações de páginas privadas - ou seja, apenas acessadas por uma única *thread* por exemplo - podem ocorrer normalmente, sem influência do *pseudo-interleaving*.

Na seção a seguir, apresentamos nosso conjunto de *benchmarks* utilizado, e em seguida nossa metodologia experimental, bem como os resultados obtidos.

4.2 *Benchmarks* utilizados

Nesta seção apresentaremos os *benchmarks* utilizados nesse trabalho. Temos o conjunto de *benchmarks* denominado *EasyBench*, que recai no uso de *kernels* de álgebra linear; também realizamos experimentos com os *benchmarks partitionStringSearch* e *BucketSort*. Cabe aqui a ressalva sobre o uso de *benchmarks* considerados mais tradicionais, por vezes denominados *benchmarks* comerciais. Nosso trabalho não faz uso dessa categoria de *benchmarks* pelas razões expostas a seguir.

Primeiramente, não temos conhecimento de *benchmarks* tradicionais que têm como foco aplicações *multithreading* com alta taxa de uso de dados na memória e grande *memory footprint*. Temos ciência da existência de três *benchmarks* de aplicações *multithreading*: PARSEC e SPLASH-2x, ambos contidos na suíte PARSEC 3.0 [9], e NAS *Parallel Benchmark* (NPB) [1], que é implementado em OpenMP, portanto não passível de instrumentação através da otimização SPM (descrevemos com maiores detalhes as limitações do projeto no capítulo 5). A suíte do PARSEC 3.0 pode ser instrumentada pela nossa otimização, uma vez que sua vasta maioria de *benchmarks* faz uso de `pthreads` para disparar múltiplas *threads* de execução; tal suíte, contudo, não está atualizada de acordo com a categoria atual de microprocessadores - seus algoritmos e conjuntos de entradas não exploram adequadamente o potencial da geração mais moderna de processadores. Ainda, fizemos alguns experimentos com AutoNUMA nas suítes PARSEC e SPLASH-2x e percebemos que na vasta maioria dos *benchmarks* não há ganho de desempenho, e quando há ganho esse é bastante diminuto.

Nas subseções a seguir, apresentamos detalhes de nossos *benchmarks*.

4.2.1 *EasyBench*

O conjunto de *benchmarks* denominado *EasyBench* contempla 4 *kernels* de álgebra linear, implementados na forma de programas individuais que compartilham grande parte do código-fonte e das características de alocação de memória, por exemplo. Algumas características importantes comuns a todos os integrantes do conjunto são:

1. As matrizes a serem processadas são quadradas e sua dimensão é provida através de um parâmetro de invocação do programa; os elementos das matrizes são do tipo `double`.

2. Todas as matrizes são globais e alocadas/inicializadas (com valor 0) na função `main` - alguns *kernels* utilizam 2 matrizes, como a decomposição Cholesky, enquanto outros necessitam de 3 matrizes, como a fatoração LU.

3. O número de *threads* de execução a serem disparadas é fornecido através de um parâmetro de invocação dos programas.

4. Também através de um argumento na invocação das aplicações, determina-se o número de pares (ou ternas) de matrizes alocadas; desse modo estipula-se em média quantas matrizes serão processadas por *thread*.

Nos *benchmarks* do tipo *EasyBench*, *threads* são criadas e consomem matrizes do *pool* de matrizes globais gerado no início do programa. Daí o fato do argumento relacionado ao número de matrizes derivar a média de matrizes processadas por *threads* - note que pode ocorrer de uma *thread* acabar consumindo uma matriz a mais que outra, dependendo do escalonamento de *threads* realizado pelo sistema operacional. Dado que o escalonador seja justo, e o número de *threads* disparadas não exceda a quantidade de núcleos de processamento, é esperado que cada *thread* compute sobre o mesmo número de matrizes. A mesma computação é realizada em todas as matrizes, ou seja, não há particionamento de matrizes/computação.

Note que a estrutura de alocação dos *benchmarks* do conjunto *EasyBench* pode implicar alto uso de memória, pois temos um argumento para determinar o número de pares ou ternas de matrizes quadradas do tipo `double` a serem alocadas - dependendo dos parâmetros passados aos programas, podemos ter um *memory footprint* bastante grande. Em nosso experimentos, obtivemos até 32GB de memória alocada para as matrizes no maior caso, para um *kernel* que realiza processamento de três matrizes. Ainda, do modo como os *benchmarks* foram projetados, naturalmente exploramos o caso de contenção de memória, pois toda a memória é inicializada pela *thread* principal. Acreditamos ser esse modelo utilizado por parte das aplicações de computação científica, especialmente aquelas desenvolvidas num momento anterior ao uso massivo de arquiteturas ccNUMA.

Os algoritmos implementados por cada um dos 4 *benchmarks* de álgebra linear têm características únicas no que toca reuso dos elementos das matrizes e serão apresentados individualmente a seguir.

Fatoração LU: A fatoração (ou decomposição) LU (*Lower Upper*) é um algoritmo que tem como objetivo, a partir de uma matriz de entrada A, gerar duas novas matrizes triangulares[2] L e U, tal que $A = LU$, ou seja, o algoritmo decompõe uma matriz num produto de matrizes triangulares. O nome do algoritmo é baseado no fato de que a matriz L é triangular inferior (elementos abaixo da diagonal são nulos) e a matriz U é triangular superior (elementos acima da diagonal são nulos) - seu uso principal se dá na resolução de sistemas lineares.

Nosso algoritmo em linguagem C para a fatoração LU pode ser visualizado na Figura 4.1. Note que as matrizes A e B apresentam maior reuso, enquanto o grau de reutilização de C é reduzido, mas ainda significativo. Pelo fato de haver 3 laços aninhados, os acessos realizados dentro do laço mais interno ao aninhamento tendem a ditar o impacto da queda de desempenho da aplicação no caso de grande *memory footprint* aliado a uma má distribuição dos dados na memória. Para nossos experimentos, utilizamos uma matriz de entrada cujos elementos são todos iguais a 1.

Decomposição Cholesky: A decomposição Cholesky de matrizes também é uma operação do tipo fatoração: aqui, busca-se representar uma matriz A como o produto de uma matriz triangular inferior L e sua transposta, L^T - assim, após a decomposição Cholesky têm-se $A = LL^T$. Tal operação é bastante utilizada na resolução de sistemas lineares - em alguns casos tal decomposição torna a resolução dos sistemas lineares muito mais eficiente

[2]Matriz triangular é aquela cujos elementos acima ou abaixo de sua diagonal são nulos.

```
1  //LU decomposition of A
2  //B represents L and C represents U
3  //A, B and C are pointers to global matrices
4
5  int i,j,k;
6
7  for (i=0; i<N*N; ++i)
8    A[i] = 1.0;
9
10 for (i=0; i<N; ++i) {
11   B[i*N + i] = 1;
12
13   for (j=(i+1); j<N; ++j) {
14     B[j*N + i]= A[j*N + i]/A[i*N + i];
15
16     for (k=(i+1); k<N; ++k)
17       A[j*N + k] = A[j*N + k] - B[j*N + i]*A[i*N + k];
18   } //for_j
19
20   for (k=i; k<N; ++k)
21     C[i*N + k] = A[i*N + k];
22 } //for_i
```

Figura 4.1: Algoritmo em C da fatoração LU.

do que se fosse utilizada a já mencionada fatoração LU.

O algoritmo em C utilizado em nossos experimentos para realizar a decomposição Cholesky está apresentado na Figura 4.2 - nossa matriz de entrada é A, e ao final do algoritmo temos $B = L$. Não realizamos a transposição B, uma vez que tal operação não é computacionalmente relevante. Aqui novamente ambas as matrizes são interessantes para serem migradas, já que as duas são acessadas frequentemente no laço mais interno.

Adição matricial: A adição de duas matrizes é uma operação algébrica bastante simples; seu algoritmo apresenta complexidade computacional linear. Nosso objetivo ao avaliar esse *benchmark* é demonstrar o comportamento da otimização SPM num caso em que a migração não se faz necessária, e tende a piorar o desempenho se realizada. Na Figura 4.3 apresentamos o algoritmo utilizado para a adição de matrizes; tal algoritmo calcula a soma de A e B, de modo que ao final de sua execução temos $C = A + B$. Como pode ser observado, o acesso à memória é realizado uma vez para cada elemento de A e B, portanto não há reutilização de dados. Assim, nesse caso esperamos que a migração seja evitada pela heurística.

Multiplicação matricial: O produto matricial apresenta uma variedade de algoritmos otimizados, que buscam reduzir e otimizar o acesso à memória, de modo que ao utilizar um desses algoritmos não estaríamos expondo uma situação ideal de uso para a otimização SPM. Assim, optamos por implementar o algoritmo "ingênuo" de multiplicação matricial, de complexidade computacional cúbica e com alta taxa de acesso à memória. A Figura 4.4 descreve nosso algoritmo: nossas entradas são as matrizes A e B, de modo que ao final do algoritmo temos $C = A \times B$. Note que nesse caso, cada elemento da matriz C é

```
1  //Cholesky decomposition of A
2  //A and B are pointers to global matrices
3  //B is zero-initialized
4
5  double s;
6
7  for (i=0; i<N*N; i++)
8    A[i] = 1.0;
9
10 for (i=0; i<N; i++) {
11   for (j=0; j<(i+1); j++) {
12     s=0;
13
14     for (k=0; k<j; k++)
15       s += B[i * N + k] * B[j * N + k];
16
17       if (i == j)
18         B[i * N + j] = sqrt( A[i * N + i] - s );
19       else
20         B[i * N + j] = ( 1.0 / B[j * N + j] * (A[i * N + j] - s) );
21
22   } //for_j
23 } //for_i
```

Figura 4.2: Algoritmo em C da decomposição Cholesky.

```
1  //Matrix addition
2  //A, B and C are pointers to global matrices
3  //A, B and C are zero-initialized
4
5  int i,j;
6
7  for (i=0; i<N; ++i)
8    for (j=0; j<N; ++j)
9      C[i*N + j] = A[i*N + j] + B[i*N + j];
```

Figura 4.3: Algoritmo em C da adição matricial.

acessado uma única vez, portanto esperamos que a heurística não migre as páginas de C.

4.2.2 *partitionStringSearch*

O *benchmark partitionStringSearch* possui uma estrutura diferente dos *benchmarks Easy-Bench*. Seu código, escrito em C, implementa um algoritmo de busca de palavras em cadeias de caracteres. O algoritmo, apresentado na Figura 4.5, realiza um procedimento simples. Primeiramente, um vetor é particionado de modo que cada *thread* realize a computação numa porção da cadeia de caracteres. O algoritmo então é executado um determinado número de rodadas, definido através da variável `NUM_QUERIES`, que representa um argumento fornecido ao programa na linha de comando.

Em geral algoritmos de busca de caracteres em cadeias envolvem buscas de caracteres aleatórios ou providos pelo usuário, a serem localizados em cadeias também fornecidas de algum modo não-determinístico. No nosso caso, executamos o algoritmo de modo deter-

```
1  //Matrix product
2  //A, B and C are pointers to global matrices
3  //A, B and C are zero-initialized
4
5  for (i=0; i<N; i++) {
6    for (j=0; j<N; j++) {
7      double tmp=0;
8
9      for (k=0; k<N; k++)
10       tmp += A[i*N + k] * B[k*N + j];
11
12     C[i*N + j] = tmp;
13   } //for_j
14 } //for_i
```

Figura 4.4: Algoritmo em C do produto matricial.

minístico: inicializamos uma variável como uma sequência de caracteres `a` e realizamos comparações de cada caractere da variável com cada caractere da cadeia; ao encontrar uma sequência correspondente, o algoritmo é interrompido. Nosso vetor global, contudo, é inicializado com uma sequência de caracteres `b` - assim, garantimos que o algoritmo será executado em seu pior caso, já que deverá obrigatoriamente percorrer todo o vetor.

O algoritmo do *benchmark PartitionStringSearch* apresenta um grau de reuso dependente do número de rodadas desejadas. Diferentemente dos *benchmarks EasyBench*, aqui a migração ocorre inteiramente antes das múltiplas rodadas, e porções do vetor global são migradas para diferentes bancos de memória - no caso do *benchmark EasyBench*, as matrizes são consumidas por *threads* e a migração ocorre antes da execução, mas uma migração é realizada para cada matriz processada pelas *threads*, pois não é possível determinar-se de antemão o conjunto total de matrizes consumidas por cada *thread*.

4.2.3 *BucketSort*

Bucket sort, também chamado de *bin sort*, é um algoritmo de ordenação de vetores distribuído. Tal algoritmo particiona um vetor a ser ordenado em *buckets*, e então ordena tais estruturas individualmente. Por fim, o algoritmo concatena os *buckets*, resultando no vetor inicial ordenado. O algoritmo *bucket sort*, por sua estrutura distribuída, pode ser implementado de modo a usar múltiplas *threads* na ordenação de vetores - por exemplo, pode-se determinar que cada *thread* ordene um *bucket* após o particionamento do vetor.

Nossa implementação, apresentada na Figura 4.6, segue uma abordagem simples. Utilizamos algumas funções auxiliares com o intuito de simplificar o código-fonte, já que o algoritmo *bucket sort* é um pouco mais extenso do que os apresentados anteriormente. Primeiramente, cabe notar que `vector` e `Bucket` são estruturas que contém dados e "metadados", como o tamanho alocado da estrutura e o seu número de elementos corrente - os `buckets` possuem um tamanho inicial, e podem ser realocados sob demanda. O algoritmo segue as seguintes etapas: após a alocação e inicialização dos *buckets*, obtém-se os valores mínimo e máximo do vetor a ser ordenado, e através dessa informação, determina-se o tamanho de cada *bucket*; a partir disso, os elementos são distribuídos nos *buckets* e esses, por sua vez, são fornecidos para *threads*, que os ordenam seguindo um algoritmo de orde-

```
1  //All upper-case variables are global initialized ones
2  //TID represents the thread number, from 0 to (NUM_THREADS - 1)
3  //NUM_QUERIES is a program argument
4
5  long my_region = ARRAY_SIZE/NUM_THREADS;
6  long my_start  = TID*my_region;
7  long my_end    = (TID + 1)*my_region;
8  int64_t  i, j, k, num_occurrences;
9
10 for (i=0; i<NUM_QUERIES; i++) {
11
12   char *pattern = (char*)malloc(STR_SIZE);
13
14   for (j=0; j<STR_SIZE; j++)
15     pattern[j] = 'a';
16
17   num_occurrences = 0;
18
19   for (j=my_start; j<my_end - STR_SIZE + 1; j++) {
20       int success = 1;
21
22       for (k=0; k<STR_SIZE; k++)
23         if (v[j + k] != pattern[k]) {
24           success = 0;
25           break;
26         }
27
28     if (success)
29       num_occurrences++;
30   } //for_j
31 } //for_i
```

Figura 4.5: Algoritmo em C do *benchmark PartitionStringSearch*.

nação por seleção - finalmente, após terem sido ordenados, os *buckets* são concatenados. Mais uma vez, no caso do *BucketSort*, a migração ocorre por partes do vetor, ou seja, páginas de memória referentes a uma porção do vetor são migradas para os bancos de memória atrelados aos processadores que executam a *thread* responsável pela computação do determinado trecho do vetor. O reuso do vetor é quadrático, determinado pelo algoritmo de ordenação escolhido, i.e, ordenação por seleção.

Descrevemos abaixo a metodologia usada nos experimentos, bem como seus resultados.

```
1  //NUM_THREADS and vector (array to be sorted) are global variables
2  //InitBucket allocates and zero-initialize the buckets
3  //FindMinMax searches for vector min and max
4  //AddElement adds elements to buckets and reallocate them if needed
5  //CreateThread/ThreadJoin are wrappers to pthread functions
6  //SelSortBucket does a Selection Sort on elements of each bucket
7  //FreeBucket deallocate buckets
8
9  Bucket buckets[NUM_THREADS];
10 long i,j,min,max,index;
11
12 //Allocate Memory
13 for(i=0; i<NUM_THREADS; i++)
14   InitBucket(&buckets[i], (vector.size/NUM_THREADS) + 1);
15
16 //Get min and max values, and calculate bucket_length
17 FindMinMax(vector, &min, &max);
18 int range = max-min;
19 int bucket_length = range/NUM_THREADS;
20
21 //Distribute elements into buckets.
22 for(i=0; i<vector.size; i++) {
23   int element = vector.data[i];
24
25   //Select bucket
26   int BucketNumber = (element-min)/bucket_length;
27   if(BucketNumber >= NUM_THREADS) {BucketNumber = NUM_THREADS-1;}
28   AddElement(&buckets[BucketNumber], element);
29 }
30
31 //Sort each bucket individually
32 for (i=0; i<NUM_THREADS; i++)
33   Create_Thread(i, SelSort_Bucket, &buckets[i]);
34
35 for(i=0; i<NUM_THREADS; i++)
36   Thread_Join(i);
37
38 //Concatenate the results of the buckets in the original vector
39 for (i=0, index=0; i<NUM_THREADS; i++) {
40   for(j=0; j<buckets[i].numElements; j++) {
41     vector.data[index] = buckets[i].data[j];
42     index++;
43   }
44   FreeBucket(&buckets[i]);
45 }
```

Figura 4.6: Algoritmo em C do *benchmark BucketSort*.

4.3 Metodologia e resultados experimentais

Nesta seção, apresentaremos a metodologia utilizada nos experimentos e os resultados experimentais encontrados. Todos os experimentos foram executados num sistema SGI H2106-G7, de arquitetura ccNUMA, com 4 processadores AMD Opteron 6282 SE, contando com 64 núcleos de processamento e 8 nós NUMA. Cada processador possui 16 núcleos, e cada núcleo apresenta 16 KB de *cache* L1 de dados - dois núcleos compartilham

64 KB de *cache* L1 de instruções e 2MB de L2, e todo o processador compartilha 12MB de *cache* L3. O sistema possui um total de 128 GB de memória RAM, distribuídos em 8 bancos de 16GB cada; em nenhum experimento fez-se uso da área de *swap*. A distribuição Linux utilizada foi Ubuntu 12.04 com *kernel* 3.13.

Realizamos os experimentos de modo que cada *benchmark* foi executado por 10 vezes. Como medida de normalização dos resultados, utilizamos a mediana das 10 rodadas, pois acreditamos que nossos experimentos não apresentam uma distribuição normal; assim, a média aritmética seria uma medida injusta. Para todos os *benchmarks*, 10 valores de entrada variável foram fornecidos, de modo que nossos gráficos são compostos por 10 pontos calculados experimentalmente.

Todos os *benchmarks* foram executados com os *thresholds* da heurística nos valores de 200 para reuso e 20% da *cache* para tamanho da estrutura - tais constantes influenciam no código de migração, de modo que quanto maiores, mais improvável é da técnica migrar páginas de memória. Na Figura 4.7, apresentamos um comparativo de *thresholds* de reuso; executamos as aplicações LU e Add do *benchmark* EasyBench com vários valores para tal limiar, com matrizes de dimensão 600, 1600 e 2400. Podemos perceber que o valor de 200 apresenta resultados superiores para grandes cargas, ainda que as diferenças sejam sutis. O eixo das abscissas representa os *thresholds* e o eixo das ordenadas indica o tempo de execução dos *benchmarks*. Sobre o *threshold* de *cache*, fizemos experimentos similares contudo obtivemos resultados não esperados a serem discutidos na seção a seguir.

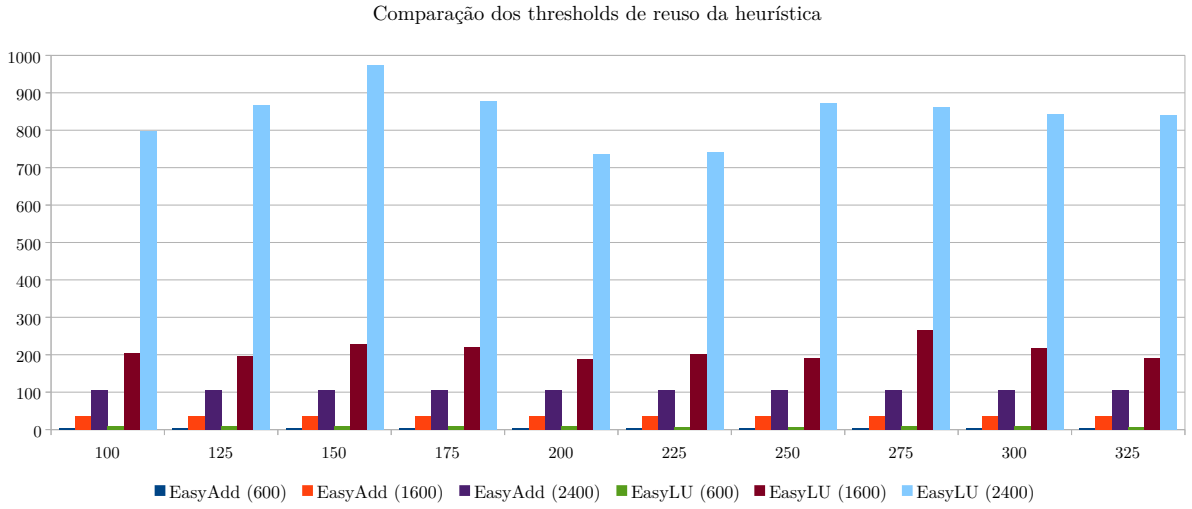


Figura 4.7: Análise comparativa de valores de *threshold* de reuso.

Os parâmetros específicos fornecidos para cada *benchmark* foram os seguintes:

1. Para os *benchmarks* do conjunto *EasyBench* foram passados os parâmetros: 256 pares/ternas de matrizes e 64 *threads* por rodada - as dimensões variaram de 600 até 2400.

2. Para o *benchmark partitionStringSearch* foram fornecidos os argumentos: 64 *threads* e vetor global de tamanho 4GB. O número de rodadas (argumento `NUM_QUERIES`) variou de 8 a 4096, numa escala exponencial (8, 16, ..., 2048, 4096).

3. Para o *benchmark BucketSort*, foram passados os parâmetros 64 *threads* de execução e número de elementos do vetor variando de 10000000 a 28000000, com incrementos de 2000000.

Tais parâmetros foram escolhidos pois permitem uma avaliação dos *benchmarks* em situações desde as mais "leves" até as mais extremas em relação ao *memory footprint*; o tempo de execução, ao se usar tais parâmetros, também ficou num intervalo distribuído, não tendo sido muito pequeno nem exageradamente grande.

Sobre o tempo de compilação, vale citar que nosso passo de otimização agrega mais análises, e portanto acarreta aumento no tempo de compilação. Realizamos um experimento com todos os *benchmarks*, apresentado na Figura 4.8 - o eixo das abscissas representa o *benchmark* e o eixo das ordenadas, o tempo de compilação (em segundos). Comparamos os tempos de compilação regular (chamado de normal na Figura), da compilação com a otimização SPM habilitada e também com SPM mais o mecanismo de travamento de *threads* ativado - esses valores são apresentados em barras na Figura 4.8. Nota-se que para as aplicações *EasyBench*, o impacto da otimização é menor visto que o tempo de compilação regular é maior, dada a natureza mais complexa do código. Como a compilação da aplicação não é uma atividade corriqueira, não consideramos a sobrecarga no tempo de compilação com um fator negativo da otimização, em especial visto que muitos sub-passos são agregados na otimização SPM, e portanto o aumento nesse tempo é esperado.

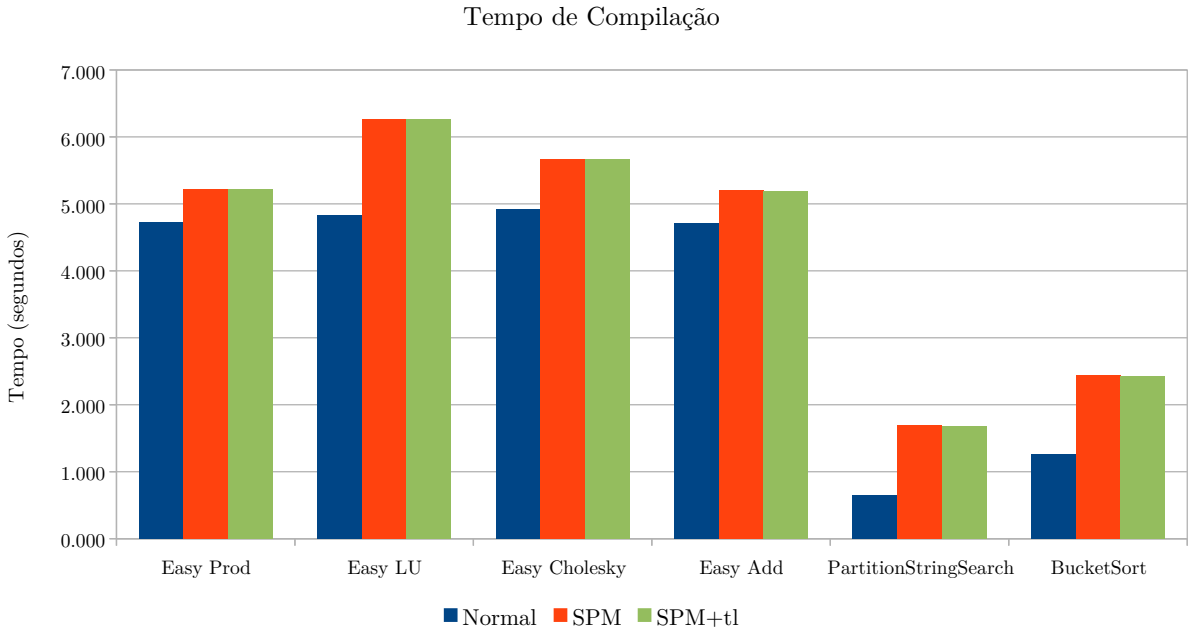


Figura 4.8: Tempo de compilação dos *benchmarks*.

Abaixo, descrevemos e discutimos os resultados experimentais.

4.3.1 Resultados dos *benchmarks EasyBench*

Nossos resultados apresentaram alguns comportamentos bastante inesperados - através de algumas análises fizemos descobertas interessantes, que demonstram tanto a técnica

de migração de páginas memória como o travamento de *threads* como sendo passíveis de sofrer influência/influenciar outros mecanismos de otimização do sistema operacional com os quais não esperávamos nos relacionar num primeiro momento.

Iniciaremos nossa exposição de resultados com um gráfico de desempenho do *benchmark EasyBench Product*, por ser o que apresenta resultados mais díspares. O gráfico mostrado na Figura 4.9 exibe um comparativo de técnicas, tais como SPM, SPM+tl (em que o mecanismo de *threadlock* está ativado), `numactl` (em que não houve migração de páginas e sim alocação de memória de forma distribuída, com a ferramenta `numactl`) e AutoNUMA. O eixo das abscissas representa a dimensão das matrizes (sempre quadradas) e o das ordenadas, o desempenho obtido com a técnica representada numa determinada linha do gráfico.

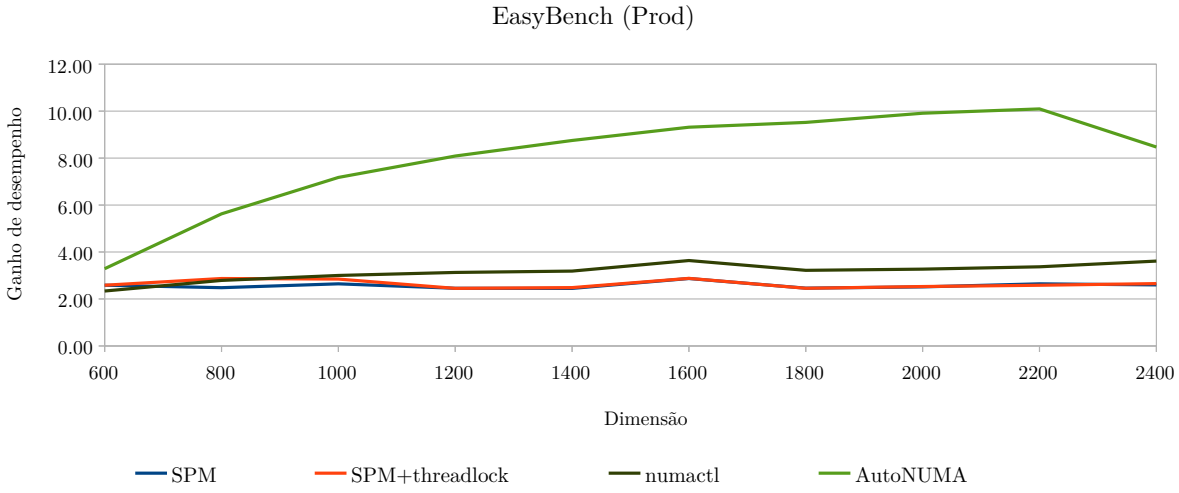


Figura 4.9: Resultados do *benchmark EasyBench* Product.

Podemos observar primeiramente a disparidade encontrada entre os resultados obtidos com o mecanismo AutoNUMA em comparação com todas as outras técnicas. O AutoNUMA obteve um ganho de desempenho de até 10x, ao passo que com a técnica SPM (com ou sem *thread lock*) esse foi de no máximo 3x, e com `numactl` o ganho foi de até 3.7x. Além dessa variação acentuada no ganho de desempenho entre as técnicas, notamos um comportamento não esperado em relação ao desempenho da técnica SPM, que ficou abaixo do `numactl` e ainda não obteve melhora ao se utilizar do mecanismo *thread lock*. De todos os *benchmarks* que avaliamos, esse foi o que obteve maior disparidade nos resultados e portanto optamos por investigar mais a fundo as causas por trás disso.

Vale ressaltar aqui, antes de nos aprofundarmos nas explicações sobre os resultados encontrados na avaliação do *EasyBench Product*, que esse *benchmark* contempla um algoritmo bastante sujeito à queda de desempenho baseada na má localidade dos dados e portanto tal algoritmo tende a causar um alto número de *misses* de acessos à TLB. Num experimento realizado com o utilitário de perfilamento `perf`, observamos que no caso de uma execução do *EasyBench Product* com matrizes de dimensão 1600 e a técnica SPM+tl ativada, tivemos um total de 48% de *misses* nos acessos à TLB. Esse mesmo experimento com o mecanismo AutoNUMA ativado nos rendeu apenas 0.02% de *misses*,

isto é, uma queda de mais de 2000x nesse indicador - o número total de acessos à TLB foi praticamente o mesmo em ambos os casos.

Através de uma avaliação de vários indicadores estatísticos de acesso e manipulação da memória no Linux (utilizando para isso dados presentes no arquivo `/proc/vmstat`), encontramos pistas que nos levaram a uma análise mais aprofundada do mecanismo de *Transparent Huge Pages* (THP) do Linux. Tal mecanismo não foi levado em conta num primeiro momento nesse trabalho, por aparentemente não estar relacionado com migração de memória e escalonamento de *threads*, mas mostrou-se bastante influente nos resultados de nosso trabalho.

O mecanismo THP tem como objetivo agrupar de forma automática um conjunto de páginas de memória numa única página maior, denominada *huge page*. Em arquiteturas x86 e x86-64, tipicamente páginas de memória têm 4KB - é possível em geral optar por diferentes tamanhos, pois os processadores modernos têm suporte para múltiplos tamanhos, mas o caso padrão é o de 4KB. Existem também bibliotecas para que as aplicações possam fazer uso de *huge pages*, mas o mecanismo THP provê essa otimização de forma automática, sem dificuldades para o programador. O funcionamento do mecanismo se baseia em monitorar *page faults*, de modo que novas *huge pages* são geradas a cada falta e tentativamente preenchidas com dados presentes em páginas de tamanho regular contíguas na memória. Além do benefício de reduzir as *page faults*, as *huge pages* impactam de forma considerável no desempenho das aplicações por diminuírem drasticamente o número de acessos à TLB - uma página de 2MB, tamanho das *huge pages* do Linux, implica em condensar 512 entradas de páginas de 4KB na TLB numa única entrada para a respectiva *huge page* [10].

Em nossos experimentos observamos que ao se usar a técnica SPM tanto quanto AutoNUMA, *huge pages* são criadas pelo mecanismo THP; o grande problema no caso de SPM é que operações de manipulação de memória como migrações ou alterações das permissões de páginas incorrem num *fallback* para páginas regulares de 4KB, característica essa já conhecida do mecanismo de THP [24]. Como nossas aplicações possuem grande *memory footprint* e mostraram se beneficiar bastante de *huge pages* no geral, a migração acaba por anular o benefício trazido por THP e obtivemos disparidade de resultados nos *benchmarks*, sendo o caso mais emblemático o do *EasyBench Product*, já apresentado acima. Discutiremos mais sobre isso no capítulo 5 - optamos aqui por desativar o mecanismo THP em nossos experimentos para que a comparação com os outros métodos de otimização seja mais justa e sem influência dessa nova variável.

Abaixo, apresentamos em sequência os resultados dos *benchmarks EasyBench* Product e LU, já com o mecanismo THP desabilitado. As características do gráfico se preservam: o eixo das abscissas representa a dimensão das matrizes (sempre quadradas) e o das ordenadas, o ganho de desempenho obtido com a técnica representada numa determinada linha.

Ao observar o gráfico representado na Figura 4.10, podemos notar claramente o impacto que o mecanismo de THP havia aplicado nos experimentos. Aqui, com o THP desabilitado, temos resultados mais condizentes com o esperado, e observamos que as variantes da técnica SPM (com ou sem *thread lock*) apresentam desempenho similar às técnicas AutoNUMA e `numactl`. O maior *speedup* encontrado foi de 4.35x com o meca-

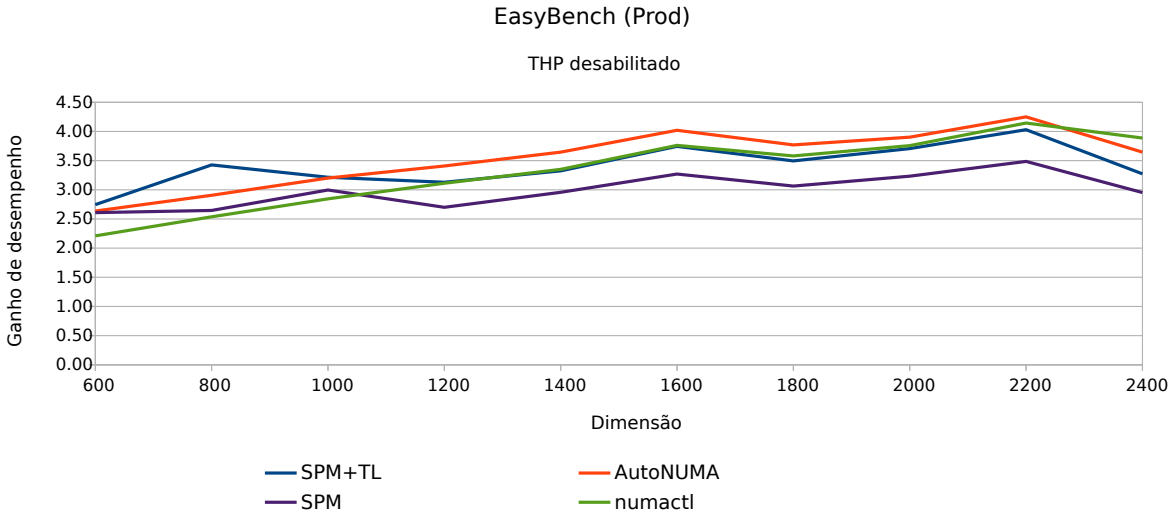


Figura 4.10: Resultados do *benchmark EasyBench* Product - THP desabilitado.

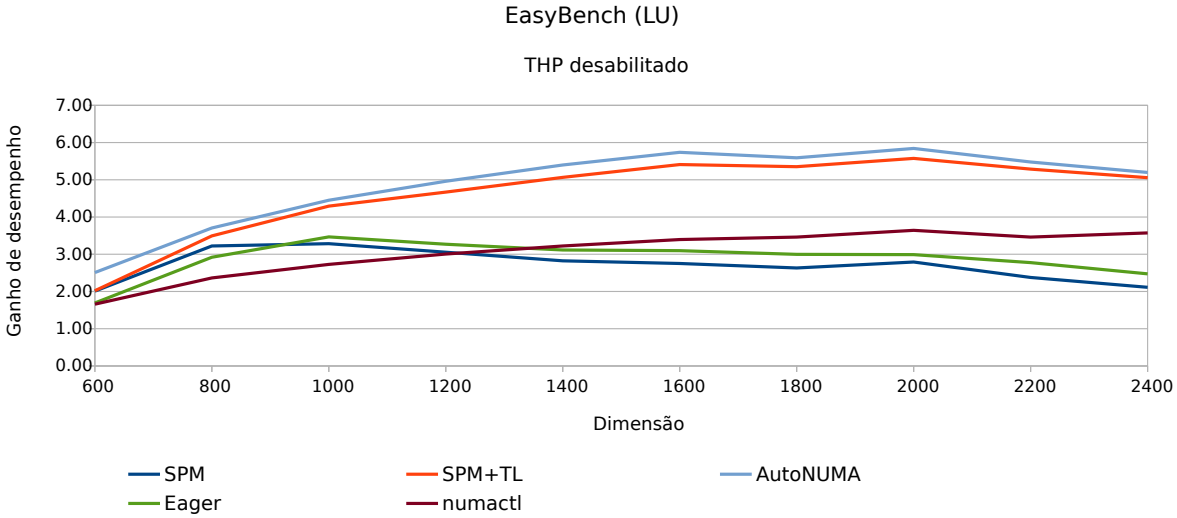


Figura 4.11: Resultados do *benchmark EasyBench* LU - THP desabilitado.

nismo AutoNUMA e 4.03x com SPM+*threadlock*, numa carga de matrizes de dimensão 2200.

Esperávamos que nossa técnica fosse superior tanto ao AutoNUMA como `numactl`, já que AutoNUMA realiza perfilamento em tempo de execução e `numactl` sequer migra páginas de memória, sendo que apenas atua na distribuição prévia dos dados de maneira *round-robin* nos bancos de memória. Como esperado, a técnica SPM sem mecanismo de travamento de *threads* obteve resultados abaixo da versão com o mecanismo ativado, já que ao migrar páginas sem levar em conta o escalonamento de *threads* incorremos no risco do escalonador afastar as *threads* dos dados já migrados e quebrar assim a afinidade atingida por SPM - nesse caso, teríamos desperdiçado as migrações, que são custosas.

Ainda assim, o mecanismo SPM+tl ficou abaixo do AutoNUMA e praticamente empatado com `numactl` para matrizes maiores, sendo possível notar uma queda no ganho de desempenho obtido conforme o aumento de carga. Esse resultado é interessante e uma

investigação adicional se fez necessária para entendê-lo. Primeiramente vamos avaliar os resultados do *benchmark EasyBench LU*, apresentados na Figura 4.11 - o comportamento foi similar e optamos por realizar nossas análises em cima do LU, ao invés do Product.

No caso do *benchmark EasyBench LU*, incluímos um experimento adicional ao conjunto: a linha Eager no gráfico representa o que denominamos *Eager Migration*, ou migração antecipada dos dados - nesse experimento, alteramos o código-fonte do *benchmark* e realizamos a migração antes dos laços interessantes manualmente. É um comparativo interessante com a técnica SPM sem *threadlock* pois valida que nosso passo de otimização do LLVM realmente é capaz de avaliar os pontos interessantes a serem migrados. Conforme observado no gráfico da Figura 4.11, os resultados de SPM e Eager foram similares, com uma leve vantagem da última abordagem - vale ressaltar que na migração antecipada fizemos uma análise manual precisa, e a variação de desempenho entre SPM e Eager foi no máximo de 17%, portanto consideramos o trabalho do passo como satisfatório, ainda que passível de alguma melhora.

Podemos notar também que no caso do *benchmark* LU, os resultados do utilitário `numactl` foram piores até do que SPM sem travamento de *threads*, o que era esperado. O ganho máximo de desempenho encontrado no *benchmark* LU foi de 5.84x com AutoNUMA e 5.57x com SPM+*threadlock* no caso de matrizes com dimensão 2000. O *benchmark* Product apresentou uma inversão: o utilitário `numactl` obteve ligeira vantagem em relação à otimização SPM - nesse caso o algoritmo "ingênuo" de multiplicação se beneficia mais da distribuição prévia de dados do que da migração por causa do tempo necessário para executar a computação sobre cada matriz; uma vez que esse é o *benchmark* mais custoso computacionalmente, o maior tempo de execução aumenta a probabilidade de um novo escalonamento de *threads* quebrar a afinidade que a otimização SPM atingiu.

O que realmente não era esperado e aconteceu em ambos os *benchmarks* foi a superação dos nossos resultados pelo mecanismo AutoNUMA, que requer perfilamento em tempo de execução e portanto supostamente incorreria numa sobrecarga maior do que nossa técnica. Aqui, a investigação nos levou para um caminho diferente: tanto SPM como AutoNUMA migram uma quantidade similar de páginas de memória, contudo a diferença é grande entre o comportamento das *threads* entre as 2 técnicas. O grande problema é que nosso mecanismo de regulação de *threads* para evitar que sejam escalonadas em processadores "distantes" dos dados após a migração desses envolve o travamento de *threads* em nós específicos. Ainda que num primeiro momento essa abordagem não seja teoricamente prejudicial ao escalonamento, na prática houve um dano ao mecanismo de agendamento de *threads*, de modo que com o uso de SPM+tl, o *benchmark* LU teve uma disparidade no tempo de execução das *threads*.

Através de um experimento prático, em que determinamos o tempo do início e do fim de cada *thread* criada, pudemos construir gráficos no estilo *Gantt chart* que demonstram o comportamento das *threads*. A Figura 4.12 representa uma amostra de 30 *threads* do *benchmark* LU. Observa-se que as *threads* se iniciam praticamente ao mesmo tempo, e sua duração também é similar - esse comportamento indica que o escalonamento das *threads* está eficiente.

Na Figura 4.13, observa-se uma situação bastante diferente: algumas *threads* se iniciam em momentos diferentes, e ainda as que são iniciadas em tempos similares têm sua

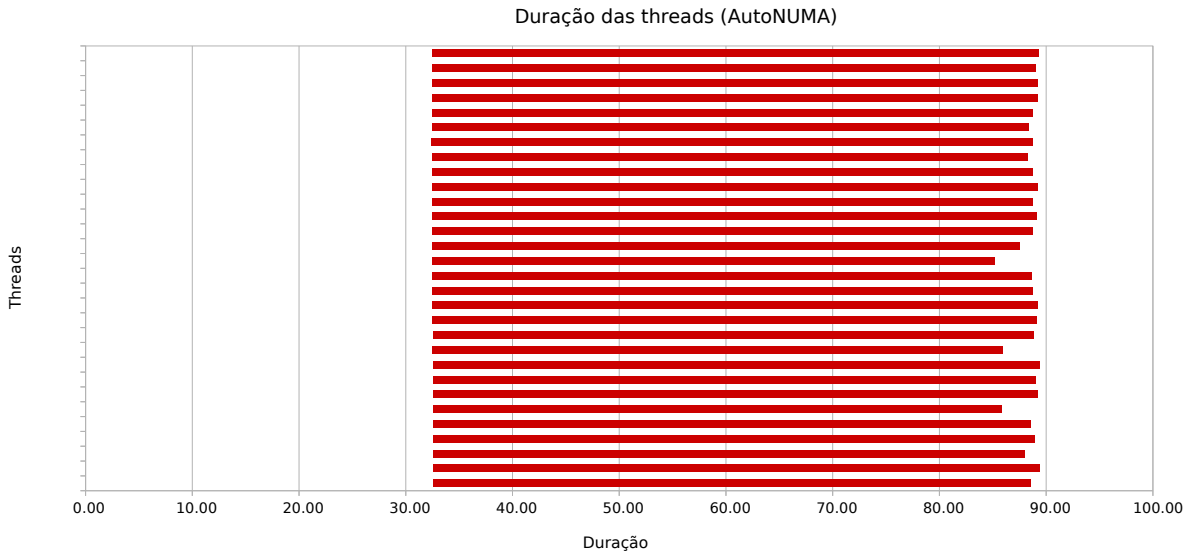


Figura 4.12: Amostra do início e duração de 30 *threads* do *benchmark EasyBench* LU com uso do mecanismo AutoNUMA.

duração variável. Algumas *threads* duram 75 segundos, enquanto outras duram em torno de 22 segundos. De alguma forma, nosso mecanismo de travamento de *threads* influencia negativamente na execução das *threads*. Realizamos a medição do tempo de início como primeira atividade no código das *threads*, mas sabemos que nosso mecanismo de *threadlock* instrumenta o código e adiciona uma chamada a uma função de travamento de *threads* exatamente no início da função, por isso temos essa disparidade no tempo de início das *threads* - nossa chamada está atrasando o começo de algumas *threads* e influenciando no escalonador de modo que algumas *threads* recebem mais tarefas do que outras. A soma total do tempo das *threads* foi superior quando SPM foi usada, indicando uma piora no desempenho em relação ao AutoNUMA, conforme apresentado na Figura 4.13. Também vale ressaltar que o uso médio de CPU foi inferior quando do uso de SPM (2000%) em relação ao AutoNUMA (3700%), isto é, SPM com o mecanismo de *thread lock* ativado está levando a uma sub-utilização da capacidade total de processamento.

Resta observar que nossos *benchmarks* da suíte *EasyBench* possuem uma grande quantidade de matrizes a serem consumidas por *threads*, portanto a nossa influência negativa no escalonamento nos leva a uma situação em que algumas *threads* consomem mais matrizes do *pool* enquanto outras ficam ociosas; dessa forma, sub-utilizamos a capacidade de processamento do sistema. Ainda, o mecanismo AutoNUMA possui, conforme já detalhado no início desse capítulo, políticas especiais de escalonamento para mover *threads* para nós em que seus dados estão presentes. Ou seja, o escalonador eficiente de *threads* do AutoNUMA unido com o sistema de perfilamento e migração de páginas quando necessário se mostra uma combinação mais efetiva do que nossa técnica, atingindo resultados levemente melhores ainda que com a sobrecarga do perfilamento. Trataremos mais sobre essa assunto na seção de Trabalhos Futuros, no capítulo 5.

A seguir, apresentamos os resultados do *benchmark EasyBench* Cholesky, na Figura 4.14. Os resultados ficam bem parecidos nesse caso, de modo que como nosso código é

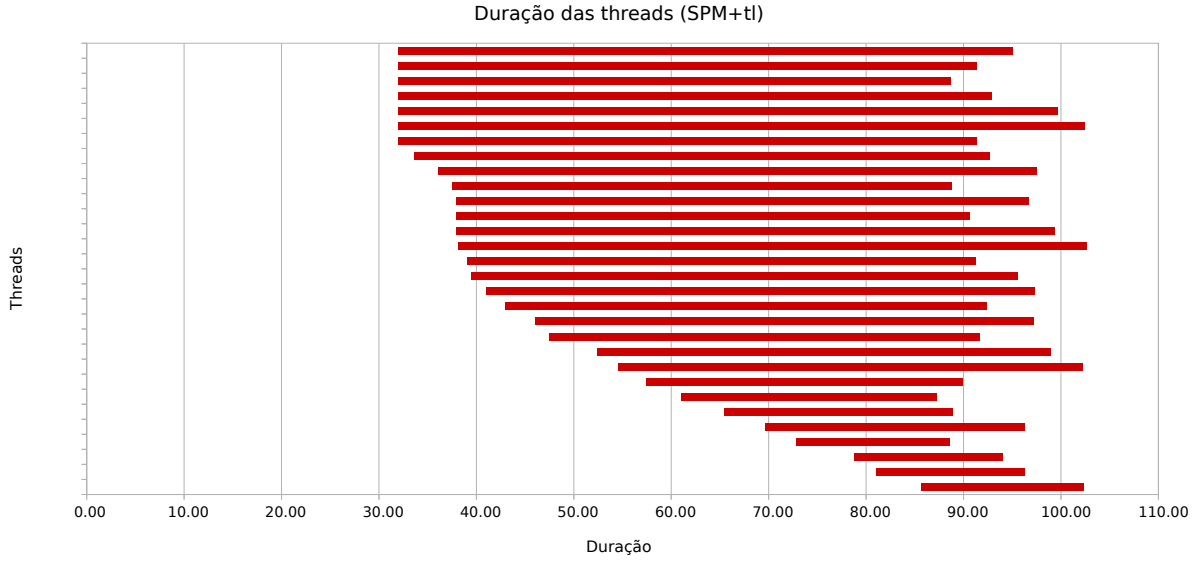


Figura 4.13: Amostra do início e duração de 30 *threads* do *benchmark EasyBench* LU com uso do mecanismo SPM+*threadlock*.

comum para todos os *benchmarks* exceto pelo algoritmo algébrico em si, percebemos uma tendência de desempenho em casos de grande *memory footprint* e alto reuso de dados, com tarefas consumidas pelas *threads*. Na seção a seguir veremos outros tipos de *benchmarks*, em que a divisão de tarefas ocorre de antemão e não há posterior consumo de tarefas por *threads*. No caso do *benchmark* Cholesky, a técnica SPM sem travamento de *threads* se mostra menos escalável do que `numactl` por exemplo, já que claramente vemos uma queda conforme o aumento de carga. A técnica SPM com travamento mostra bons resultados escaláveis, mas ainda fica levemente inferior ao mecanismo AutoNUMA, pelos motivos já mencionados.

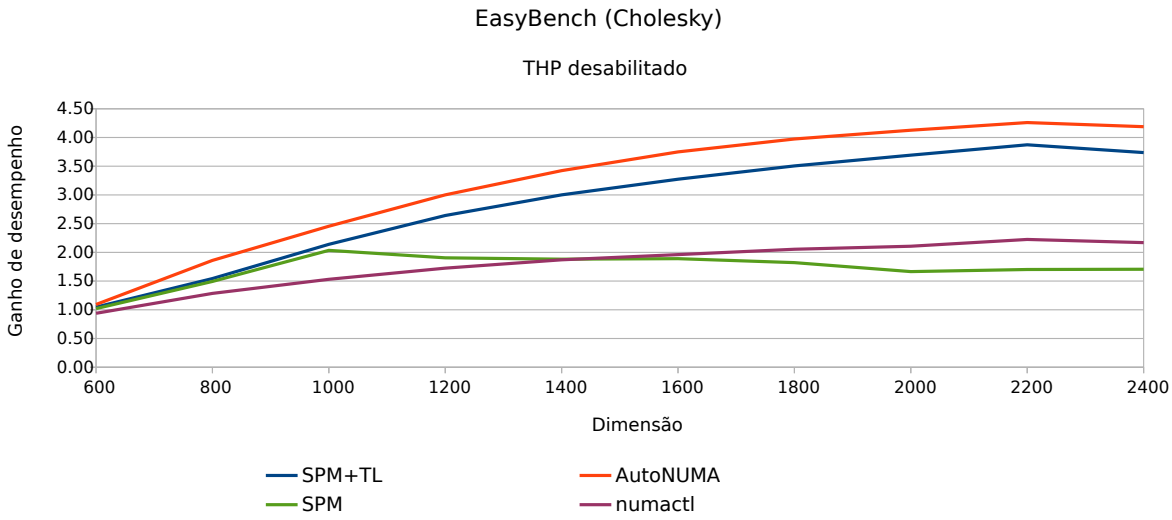


Figura 4.14: Resultados do *benchmark EasyBench* Cholesky - THP desabilitado.

Finalmente, apresentamos os resultados do último componente da suíte *EasyBench*; o

benchmark Add apresentou resultados bastante positivos, exibidos na Figura 4.15. Nesse caso, incluímos também o experimento de migração antecipada, pois a comparação entre o mecanismo AutoNUMA, migração manual e SPM é bastante interessante aqui, já que esse *benchmark* não apresenta reuso de dados e portanto nossa heurística deveria evitar migrações prejudiciais.

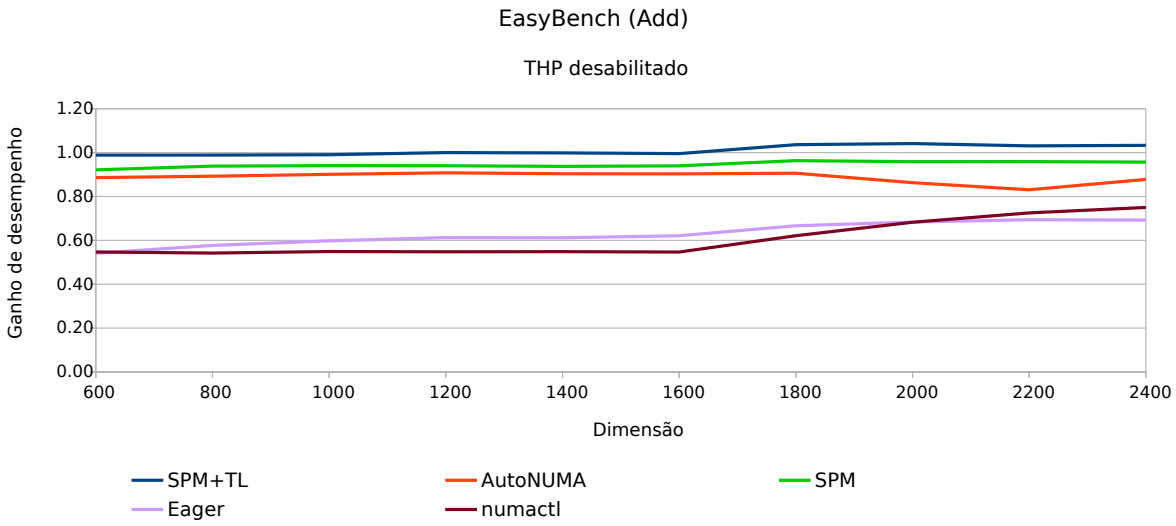


Figura 4.15: Resultados do *benchmark EasyBench* Add - THP desabilitado.

Observando o gráfico percebemos que o comportamento esperado ocorreu. A migração antecipada prejudicou o desempenho, ao passo que as otimizações SPM com e sem travamento de *threads* não prejudicaram o desempenho da aplicação - para cargas maiores, a otimização SPM com *threadlock* apresentou resultados melhores, tendo obtido um desempenho sempre igual ou melhor que o *baseline*, possivelmente devido a pequenas migrações de estruturas com reuso não presentes no algoritmo algébrico. A versão sem travamento de *threads* apresenta uma ligeira queda de desempenho praticamente negligenciável, de no máximo 4%, ao passo que a migração manual apresenta quedas de até 50% no desempenho. O resultado interessante aqui fica por conta do mecanismo AutoNUMA, que obteve quedas de desempenho de até 18% devido à migrações desnecessárias que realiza.

A partir dos experimentos acima apresentados pudemos concluir que, apesar do nosso mecanismo de travamento de *threads* estar aquém do esperado, ainda foi possível obter bons resultados com a técnica SPM+*threadlock*. Há que se melhorar o mecanismo de travamento bem como evitar a influência negativa de nossa otimização no mecanismo de THP; exploraremos algumas idéias a esse respeito na seção de Trabalhos Futuros do capítulo 5. Na seção a seguir, apresentaremos os resultados dos *benchmarks* que não pertencem ao conjunto *EasyBench*.

4.3.2 Resultados dos *benchmarks BucketSort* e *partitionStringSearch*

Todos os *benchmarks* do conjunto *EasyBench* apresentam uma estrutura similar, diferindo apenas na computação em si, no núcleo algébrico de cada um. Os *benchmarks partiti-*

onStringSearch e *BucketSort*, por outro lado, apresentam estrutura diversa e métodos de divisão de tarefas entre *threads* também diferentes da suíte *EasyBench*. Assim, esperamos alguns resultados diferentes aqui.

Primeiramente, analisamos os resultados do *benchmark partitionStringSearch*, apresentados na Figura 4.16. Tal *benchmark* possui uma estrutura de divisão de tarefas bastante diferente da suíte *EasyBench* - aqui, um vetor é particionado entre as *threads* e a computação ocorre nos dados dessa porção do vetor. Não há consumo de novos dados pelas *threads*, o que torna o trabalho da otimização SPM mais efetivo, pois os cálculos dos intervalos do vetor a serem migrados permanecem válidos por toda a computação e não há novas migrações de dados como no caso dos *benchmarks EasyBench*, em que a cada nova matriz consumida por uma *thread*, uma reinvocação da função de migração é realizada.

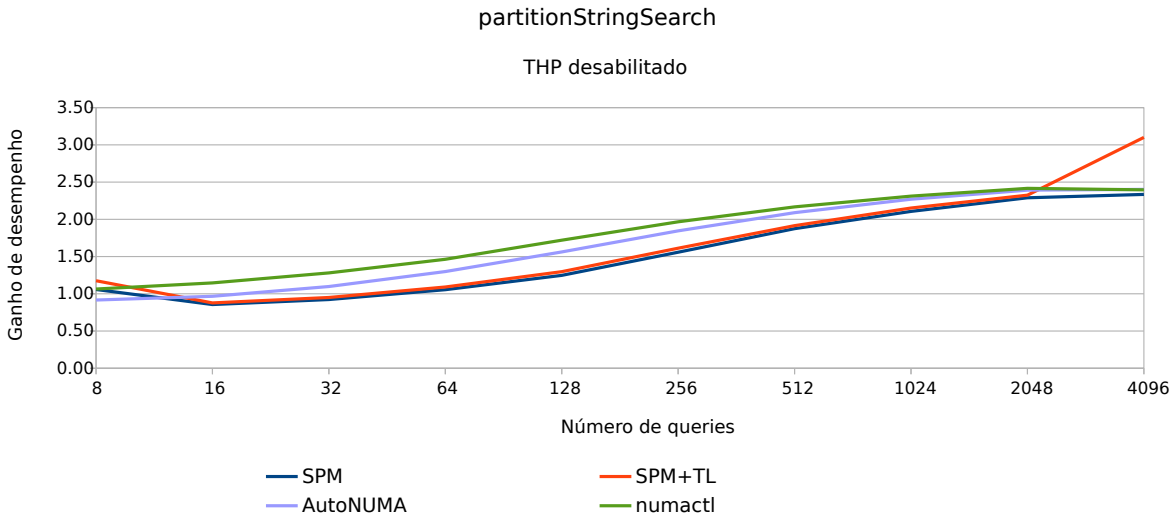


Figura 4.16: Resultados do *benchmark* partitionStringSearch - THP desabilitado.

Observamos que a técnica SPM apresenta uma queda no desempenho entre 16 e 32 *queries*, e depois seu ganho de desempenho é crescente. Isso ocorre pois nossa heurística não é eficiente nesse caso, evitando com sucesso a migração para o caso de 8 *queries*, mas migrando dados de forma prejudicial para os casos de 16 e 32 *queries*. A partir de 64 *queries* os custos de migração são pagos e portanto nossa heurística acerta ao migrar páginas. É interessante notar que nesse *benchmark* a ferramenta `numactl` se sobressai no início e a taxa de variação de seus ganhos é decrescente.

Isso ocorre pois a carga computacional do *partitionStringSearch* bem como seu *memory footprint* são pequenos comparados à suíte *EasyBench*. Note que a sobrecarga de perfilamento do AutoNUMA faz com que, especialmente no início, seus resultados não sejam tão positivos como os da ferramenta `numactl`. Outro dado bastante interessante e também relacionado com essas caraterísticas do *partitionStringSearch* é a evolução dos ganhos da otimização SPM. A partir de 64 *queries* a taxa de variação dos ganhos de desempenho da otimização SPM com ou sem travamento são crescentes, em especial entre 2048 e 4096 *queries*. Nesse ponto, o tempo de computação começa a ficar considerável, e o reuso dos dados também, assim, a escalabilidade da otimização SPM+*threadlock* se torna visível. Uma vez que nossa migração é acertada e só ocorre uma vez antes da computação,

o número de *queries* em que ultrapassamos o AutoNUMA por conta de sua sobrecarga de perfilamento é de 4096. Ao mesmo tempo, esse é o ponto em que a otimização SPM sem o mecanismo de travamento apresenta ganhos decrescentes de desempenho por conta da quebra de afinidade de dados e *threads* provocada pelo escalonador do sistema.

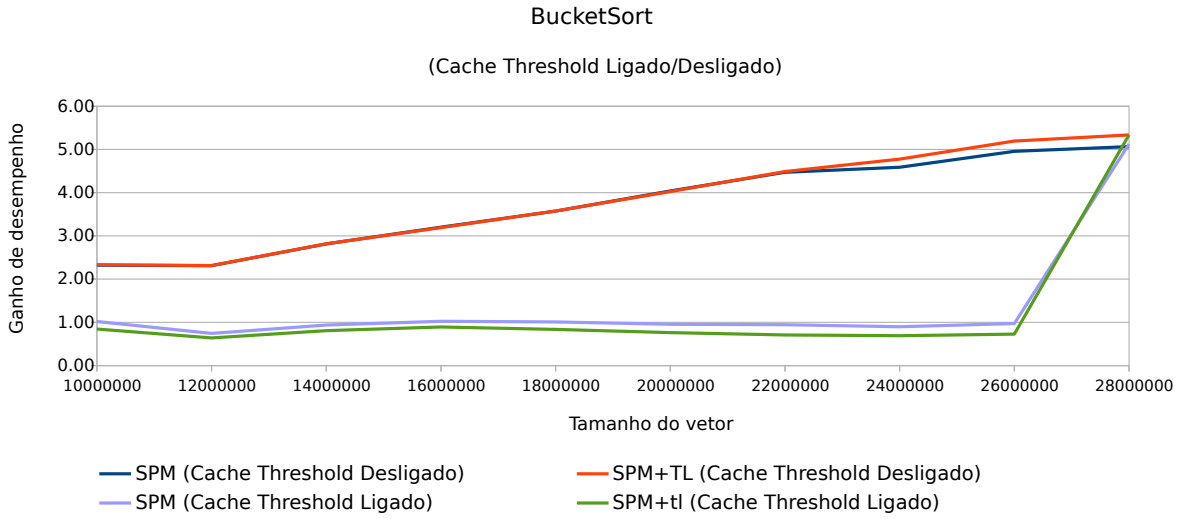
O maior ganho de desempenho encontrado no *benchmark partitionStringSearch* foi de 3.1x com o uso da otimização SPM aliada ao mecanismo de *thread lock*. Consideramos esse *benchmark* bastante interessante pois demonstra na prática uma tendência teoricamente esperada em aplicações com a mesma característica de distribuição de tarefas entre *threads*. Ainda que com um pequeno deslize por parte da heurística - que poderia ser mais acertada, conforme discutiremos no capítulo 5 -, consideramos nossos resultados bastante interessantes no caso do *partitionStringSearch*.

Finalmente, discutiremos agora os resultados de nosso último *benchmark*, o *BucketSort* - obtivemos também aqui resultados bastante notáveis. Numa primeira rodada de tal *benchmark*, os resultados apresentaram quedas de desempenho ou ganho nulo com o uso da otimização SPM (com ou sem *thread lock*) em todas as execuções, exceto quando executamos o *benchmark* com a maior entrada possível, na qual obtivemos um considerável ganho de desempenho.

Esse comportamento intrigante sugeriu que havia algum erro no *benchmark* ou na heurística, pois os resultados demonstraram um salto tremendo entre as execuções com as 2 últimas entradas. Avaliando detalhadamente encontramos um problema sutil e que afeta diretamente nossa proposta de heurística usada nesse trabalho: o *threshold* de *cache*, que avalia o tamanho das estruturas de dados - para evitar migrações de páginas que poderiam ser alocadas na *cache* -, estava impedindo migrações positivas de ocorrerem. Em resumo: nossa heurística falhou para todos os casos desse *benchmark* exceto quando executamos com a maior entrada.

Executamos então o *benchmark BucketSort* com uma heurística modificada, que apenas leva em conta o reuso das estruturas de dados - consideramos esse *threshold* o mais importante e o *benchmark EasyBench* Add nos confirmou que ele é efetivo, já que naquele caso evitou migrações prejudiciais. A Figura 4.17 exibe os resultados comparativos da técnica SPM (com e sem o travamento de *threads*) com o *threshold* de *cache* ligado e desligado - os resultados são bastante díspares.

O salto entre as execuções com as 2 últimas entradas é bastante acentuado. Houve uma leve queda de desempenho no caso da otimização SPM+*threadlock* com o *threshold* de *cache* ativado pois a migração não ocorreu e mesmo assim as *threads* foram travadas - esse comportamento não é benéfico para a otimização SPM em casos nos quais não há migração e será discutido na seção de Trabalhos Futuros do capítulo 5. Em se tratando do *threshold* de *cache*, fizemos experimentos com a suíte *EasyBench* e não obtivemos ganhos adicionais ou perdas por remover esse *threshold*. Nossa conclusão é a de que esse limiar na heurística não foi útil como imaginávamos e poderia ser suprimido para todos os *benchmarks* avaliados nesse trabalho. Contudo, isso não significa que avaliar o tamanho das estruturas seja irrelevante: num cenário em que há bastante reuso de um conjunto muito pequeno de dados, a migração apenas incorreria em perda de desempenho, pois tal conjunto reduzido de dados poderia ser alocado na *cache*. Assim, um ajuste fino do *threshold* de *cache* aliado a uma quantidade maior de experimentações poderia



35

Figura 4.17: Comparativo dos resultados do *benchmark* BucketSort com e sem o *threshold* de tamanho da estrutura de dados na heurística de migração.

permitir integrá-lo à heurística sem que prejudicasse o desempenho das aplicações - um benefício interessante seria no *benchmark partitionStringSearch*, que conforme discutido acima realizou em 2 experimentos migrações que prejudicaram seu desempenho, por falta por exemplo de um *threshold* de *cache* mais preciso. No capítulo 5 discutiremos algumas idéias para otimizar a heurística como um todo.

Na Figura 4.18 apresentamos os resultados do *benchmark BucketSort* - optamos por realizar os experimentos com o *threshold* de *cache* desativado, para que os resultados reflitam o potencial da otimização SPM num cenário em que a heurística tenha êxito. Podemos observar que o ganho de desempenho da otimização SPM foi muito bom, tendo sido o maior ganho desse *benchmark* atingido pela otimização SPM+*threadlock*, no valor de 5.34x.

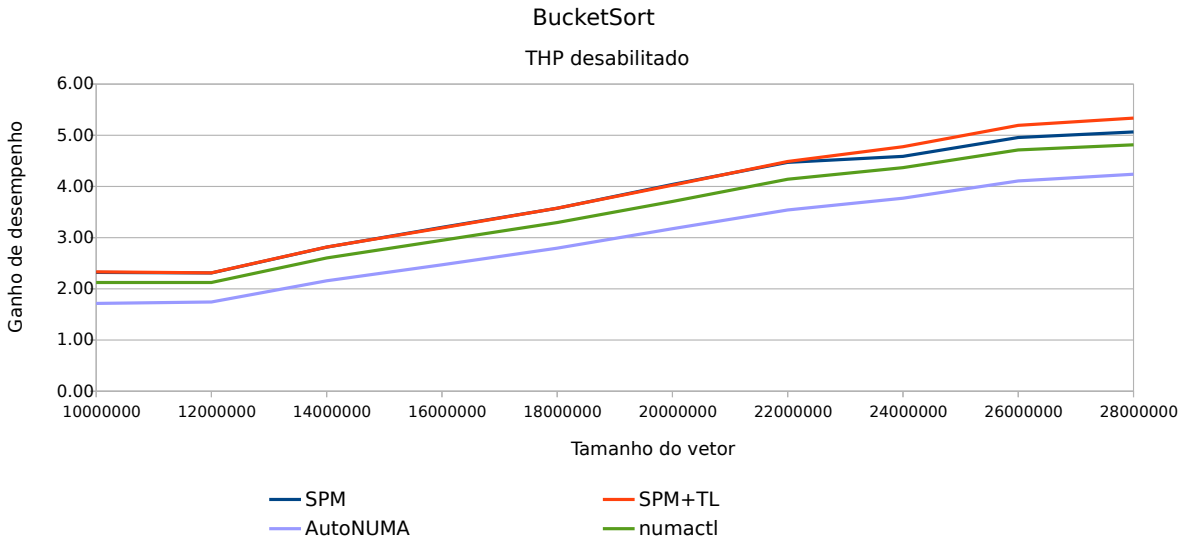


Figura 4.18: Resultados do *benchmark* BucketSort - THP desabilitado.

Novamente podemos notar que os resultados do mecanismo AutoNUMA não foram tão bons como de nossa otimização - mais uma vez aqui a sobrecarga de perfilamento aliada a um reduzido *memory footprint* tem como resultado um desempenho acima do *baseline*, mas não tão interessante como a migração prévia dos dados. Mais ainda, avaliamos a quantidade de dados migrados e pudemos constatar que nossa otimização SPM migra 11% a mais de páginas do que o mecanismo AutoNUMA, portanto o perfilamento de AutoNUMA não foi tão eficiente aqui. Finalmente, vale ressaltar que o travamento de *threads* - ainda que contando com problemas já discutidos nesse capítulo - apresenta ganhos escaláveis e se mostrou essencial para nossa otimização; nas cargas maiores dos *benchmarks* a otimização SPM com *thread lock* obteve resultados melhores do que SPM sem travamento de *threads*.

Terminamos nossa apresentação de resultados com a conclusão de que a otimização SPM tem bom potencial, em especial para aplicações com perfil de distribuição de tarefas menos dinâmico e variável - em casos nos quais a distribuição de dados a serem computados por *threads* não se modifica durante o programa, a otimização SPM tende a ter resultados melhores e superar tanto a ferramenta `numactl` como o mecanismo AutoNUMA. No capítulo seguinte apresentamos as limitações da técnica SPM bem como algumas sugestões de trabalhos futuros, em especial no que toca melhorias na heurística.

Capítulo 5

Conclusão e Trabalhos Futuros

Neste trabalho apresentamos a técnica de otimização denominada *Selective Page Migration* (SPM), uma otimização no âmbito de compiladores capaz de avaliar laços e os vetores internos a esses para determinar se a migração das páginas de memória de tais vetores poderia gerar ganho de desempenho na aplicação. Para tanto, fez-se uso de uma heurística que tem como base *thresholds* de reuso de dados e tamanho das estruturas, para evitar a migração de estruturas pouco reutilizadas ou que caibam na memória *cache*, já que que a sobrecarga imposta pela migração de páginas não é trivial. Realizamos experimentos com a técnica SPM e com outros 2 "competidores" modernos, e obtivemos resultados interessantes. Também apresentamos uma avaliação teórica de uma grande variedade de mecanismos e técnicas de múltiplos âmbitos para atenuar problemas de contenção e má distribuição de dados na memória em arquiteturas ccNUMA.

Dentre as nossas contribuições, acreditamos que utilizar um compilador para instrumentar código e inserir chamadas para uma função de migração que obedece a uma heurística é uma inovação, pois não pudemos encontrar abordagens similares em estudos da área. Além de considerarmos os resultados obtidos bons, temos a infraestrutura de instrumentação de código pronta para ser modificada, para que novas idéias com base na avaliação de laços e inferência de seu número de iterações sejam implementadas. Ainda, o estudo teórico de técnicas com mesmo objetivo e abordagens diferentes oferece um panorama atual do conjunto de soluções possíveis para problemas de contenção e má distribuição de dados em arquiteturas ccNUMA.

Como pontos fortes de nossa técnica, consideramos bons seus resultados e apreciamos sua generalidade nas análises, que podem ser aplicadas em qualquer código em C/C++ que faça uso da biblioteca `pthreads`. Também é relevante o fato da otimização SPM ser inócua nos casos em que não consegue otimizar - é uma técnica pouco destrutiva que quando não otimiza as aplicações, ao menos não prejudica. Além disso, por utilizar a infraestrutura do LLVM, nossa técnica apresenta um código claro e modular, fácil de ser modificado e adaptado para outras necessidades. Já como pontos negativos, o fato de não sermos capazes de instrumentar aplicações que fazem uso de OpenMP restringiu nosso escopo de modo razoável. Ainda, os *thresholds* de nossa heurística foram calculados empiricamente, com base na microarquitetura em que executamos nossos experimentos, e especialmente no caso do *threshold* de *cache*, nossos cálculos experimentais se mostraram falhos em ao menos um *benchmark*. Seria muito interessante o estudo de um algoritmo

heurístico capaz de obter tais *thresholds* automaticamente.

Nas próximas sessões detalharemos as limitações de nossa técnica bem como apresentaremos sugestões de trabalhos futuros.

5.1 Limitações da técnica SPM

Consideramos que a otimização proposta nesse trabalho apresentou bons resultados; no entanto claramente existe espaço para melhorias na técnica SPM. Determinamos uma série de limitações encontradas que contribuíram negativamente nos resultados, e se superadas, levariam a técnica SPM a um patamar de maior maturidade e eficácia. Na lista abaixo procuramos elencar as principais limitações de nosso projeto, numa ordem de severidade, de acordo com nossa avaliação. Possíveis soluções para algumas delas são apresentadas na próxima seção.

Mecanismo de travamento de *threads*: O mecanismo *thread lock* apresenta 2 falhas fundamentais, relacionadas ao modo como escalona *threads* e à inserção das chamadas da função de travamento no código-fonte instrumentado. Primeiramente, obtivemos um efeito negativo do travamento de *threads* no escalonador do sistema e na criação das *threads*, o que levou a um desbalanceamento de carga - algumas *threads* executam por mais tempo que outras quando usamos o mecanismo. Além disso, atualmente o mecanismo *thread lock* realiza o travamento independentemente se a migração de dados ocorre ou não; ou seja, ainda que migrações sejam prejudiciais ao desempenho e nossa heurística não as realize, as *threads* são travadas se o mecanismo estiver habilitado, ocasionando possivelmente uma queda no desempenho da aplicação.

Determinação dos *thresholds* da heurística: Uma grande limitação de nosso trabalho no que tange automatização e otimização para múltiplas plataformas está relacionada à derivação dos *thresholds* da heurística de migração de páginas. Ainda que nossa técnica por parte do compilador seja bastante automatizada e agnóstica de arquitetura, o mecanismo de migração está relacionado à arquitetura em que a aplicação irá executar. Os *thresholds* - em especial aquele relacionado ao tamanho das estruturas a partir do qual se faz interessante a migração de páginas - são dependentes de arquitetura; mais ainda, tais limitantes foram calculados empiricamente, pois nossa técnica não provê um mecanismo automático para obtenção desses *thresholds*, seja durante a compilação ou em tempo de execução das aplicações instrumentadas. Especificamente, o *threshold* de *cache* apresentou um comportamento errático no *benchmark BucketSort* e acreditamos que tal falha grave poderia acometer inúmeras aplicações, pois nossa metodologia empírica se mostrou falha no caso desse *threshold*.

Interação com o mecanismo THP (*Transparent Huge Pages*): O mecanismo THP do Linux provê automação na criação de *huge pages* - tal otimização é bastante importante no que toca o desempenho de aplicações que fazem uso de um grande número de

páginas distintas do programa em curtos espaços de tempo. Infelizmente a migração de páginas de memória desfaz o trabalho de THP para as páginas migradas, de modo que essas deixam de ser componentes de *huge pages*. Nossa técnica, por realizar migração de páginas, pode portanto anular a otimização oferecida pelo mecanismo THP, e tal comportamento é bastante indesejado, em especial pois ambas as otimizações favorecem um conjunto de aplicações similares, e associadas poderiam prover um ganho de desempenho interessante.

Aplicações que utilizam OpenMP: Uma das mais complexas limitações de nosso trabalho é a impossibilidade da técnica SPM de avaliar programas desenvolvidos com base na biblioteca OpenMP. O grande problema é que o código compilado com diretivas de OpenMP sofre uma transformação que dificulta nossas análises - as variáveis de indução saem do formato SSA e passam a residir na memória, pois o *runtime system* de OpenMP lida com os endereços de tais variáveis para escalonar tarefas em *threads*. Além disso, a divisão de tarefas é um processo que ocorre numa porção estática (no compilador) e outra dinâmica, pelo *runtime system* de OpenMP. Assim, para efetivamente tratar esses casos, precisaríamos de certa forma ter uma segunda técnica, independente da atual SPM, apenas para lidar com OpenMP. Optamos portanto pela restrição de nossas análises à biblioteca `pthreads`, incorrendo numa brusca redução de escopo de nossa técnica, mas permitindo um desenvolvimento mais ágil e conciso.

Escopo de busca de funções executadas em *threads*: Nosso passo denominado `GetWorkerFunctions`, já detalhado no capítulo 3, apresenta uma limitação que não afetou o desempenho dos nossos *benchmarks* aqui testados, mas pode afetar uma variedade de aplicações que fazem uso da biblioteca `pthreads`. A avaliação das funções que executam em *threads* é realizada em 2 etapas: primeiramente, todas as chamadas à função `pthread_create` são avaliadas para se determinar as *worker functions*; após isso a técnica SPM segue sua execução regular e toda vez que uma das *worker functions* encontradas está sendo avaliada, ocorre a inserção de uma chamada para a função de travamento de *threads* caso o mecanismo esteja ativado. O problema é que o escopo do passo `GetWorkerFunctions` é por módulo/unidade de tradução (*translation unit*), logo se houver um caso em que *worker functions* são declaradas em módulos diferentes dos que as utilizam, nosso mecanismo de travamento não será capaz de instrumentá-las.

Na seção seguinte apresentaremos algumas sugestões/idéias de trabalhos futuros que visam corrigir algumas das limitações acima mencionadas, garantindo a evolução da técnica SPM, e portanto a continuidade desse trabalho.

5.2 Trabalhos futuros

Conforme descrito na seção anterior, a técnica SPM apresenta algumas limitações que podem prejudicar o potencial ganho de desempenho das aplicações por ela instrumen-

tadas. Exceto pela impossibilidade em se tratar códigos escritos com base na biblioteca OpenMP, cuja solução seria algo de relativa complexidade, discutiremos algumas possíveis idéias para a redução dos problemas da técnica, e deixaremos algumas sugestões para a continuidade desse trabalho.

Em primeiro lugar, em se tratando de melhorias no mecanismo `SPM thread lock`, cabe uma investigação detalhada do escalonador do sistema e sua interação com as primitivas usadas pela biblioteca `hwloc` para efetuar o travamento de *threads* em determinados nós de processamento. Aparentemente a técnica usada pela biblioteca acarreta num desbalanceamento, por exemplo por alterar a prioridade que as *threads* recebem do escalonador do sistema.

Outra interessante oportunidade de melhoria no mecanismo *thread lock* se faz presente na forma de uma avaliação do travamento baseada na migração ou não de páginas de memória pela heurística. Em outras palavras, apenas devemos travar as *threads* se a migração de páginas de fato ocorrer. Uma idéia de implementação seria a inserção da função de travamento exatamente após cada chamada da função de migração, de modo que a função de travamento poderia receber um endereço de memória como parâmetro, na forma de uma variável global. Caso a função de migração não realizasse migrações, o endereço seria nulo e a função de travamento seria inócua. Do contrário, o endereço conteria o conjunto de processadores os quais pertencem ao mesmo nó em que os dados foram recentemente migrados. Assim, a função de travamento realizaria o escalonamento da *thread* no conjunto de núcleos corretos e no caso de não haver migração, nenhum travamento seria realizado. Nesse caso, a função de migração ficaria responsável por escolher um nó para a migração, e como tratamos de aplicações que executam várias *threads*, faz-se necessário um mecanismo "central" para administrar os destinos das páginas migradas, para evitar sobrecarga/ociosidade de nós.

Ainda em se tratando do mecanismo de travamento de *threads*, através de variáveis estáticas o módulo `GetWorkerFunctions` pode ser alterado para atuar de modo global, no sentido que a varredura das *worker functions* poderia ser feita numa primeira rodada percorrendo-se todos os arquivos-objeto, e o "consumo" das funções encontradas seria realizado na segunda rodada, em que o restante da técnica continuaria atuando do mesmo modo. Assim, o problema do escopo de avaliação das funções a serem travadas em *threads* seria resolvido. Obviamente uma abordagem de travamento mais sofisticada como a proposta no parágrafo anterior poderia acarretar uma modificação completa no passo `GetWorkerFunctions`, portanto cabe uma análise conjunta de melhorias no mecanismo de travamento de *threads*.

A respeito da interação negativa de nossa otimização com o mecanismo THP, acreditamos que uma abordagem interessante seria a avaliação das páginas na função de migração para determinar se são componentes de *huge pages*. Em caso positivo, poderia-se utilizar uma *syscall* de migração de *huge pages* para migrar o bloco todo ou ainda a migração individual de páginas poderia continuar sendo realizada, contudo o agrupamento das páginas regulares em *huge pages* passaria a ser refeito automaticamente em nossa função de migração.

Com relação ao cálculo dos *thresholds*, uma idéia interessante para automatizar sua derivação - com base na arquitetura, em tempo de execução das aplicações - seria a propo-

sição de um modelo de geração dos mesmos através de estudos empíricos de *benchmarks* em várias arquiteturas. Basicamente, seria necessário determinar quais características de uma arquitetura influenciam a heurística - métricas de avaliação do balanceamento de carga e inter-relação das *threads* podem ser usadas para inferir os *thresholds* em tempo de execução. Diener *et al.* [27] propõem um conjunto de métricas baseadas numa matriz de comunicação entre *threads*, de modo que poderíamos usar uma abordagem de balanceamento de carga otimizado ao derivar os *thresholds* por exemplo. Ainda, metodologias sofisticadas de avaliação do padrão de memória dos *benchmarks* podem ser utilizadas para validar a acurácia dos *thresholds*, como por exemplo a ferramenta visual de análise de memória e detecção de problemas de distribuição de carga TABARNAC, proposta por Beniamine *et al.* [14].

Finalmente, mais experimentos poderiam ser realizados com foco em aumentar o grau de generalização das análises - quanto maior for o número de análises de variadas construções de linguagem para distribuir tarefas em *threads*, maior será a acurácia da técnica em tratar os mais diversos casos, e portanto maior seu grau de generalidade. Ainda, nossa técnica poderia ser usada como apoio em outros trabalhos para se determinar oportunidades de migração e /ou replicação de dados. Por exemplo, no que toca o trabalho de Noordergraaf e van der Pas [44] - citado no capítulo 2 - nossa técnica, se executada no protótipo Sun Wildfire, poderia ser levemente modificada para identificar os vetores *read-only* para replicação e *read-write* para migração.

Obtenção do código-fonte: O código-fonte da técnica SPM - distribuído gratuitamente segundo a licença LGPL3 [11] - está disponível em:

`https://github.com/guilhermepiccoli/selective-page-migration-ccnuma`.

Nesse *link* existem instruções para compilação e uso do passo.

Referências Bibliográficas

[1] http://www.nas.nasa.gov/publications/npb.html.

[2] http://www.spec.org/cpu2006.

[3] http://www.spec.org/mpi.

[4] http://www.threadingbuildingblocks.org.

[5] https://github.com/labmec/neopz.

[6] http://llvm.org/docs/LangRef.html.

[7] http://www.ginac.de.

[8] http://www.sympy.org.

[9] http://parsec.cs.princeton.edu/parsec3-doc.htm.

[10] https://www.kernel.org/doc/Documentation/vm/transhuge.txt.

[11] https://www.gnu.org/copyleft/lesser.html.

[12] Alfred Aho, Monica Lam, Ravi Sethi, and Jeffrey D Ullman. *Compilers: Principles, Techniques, and Tools*. Addison Wesley, 2 edition, 2006.

[13] Manu Awasthi, David W. Nellans, Kshitij Sudan, Rajeev Balasubramonian, and Al Davis. Handling the problems and opportunities posed by multiple on-chip memory controllers. In *Proceedings of the 19th International Conference on Parallel Architectures and Compilation Techniques (PACT '10)*, pages 319–330. ACM, 2010.

[14] David Beniamine, Matthias Diener, Guillaume Huard, and Philippe O. A. Navaux. TABARNAC: Visualizing and resolving memory access issues on NUMA architectures. In *Proceedings of the 2nd Workshop on Visual Performance Analysis*, pages 1–9. ACM, 2015.

[15] Sergey Blagodurov, Sergey Zhuravlev, Alexandra Fedorova, and Ali Kamali. A case for NUMA-aware contention management on multicore systems. In *Proceedings of the 19th International Conference on Parallel Architectures and Compilation Techniques (PACT '10)*, pages 557–558. ACM, 2010.

[16] Edson Borin and Philippe Devloo. Programming finite element methods for ccNUMA processors. In *Proceedings of the Third International Conference on Parallel, Distributed, Grid and Cloud Computing for Engineering*. Civil-Comp Press, 2013.

[17] Edson Borin, Philippe R. B. Devloo, Gilvan S. Vieira, and Nathan Shauer. Accelerating engineering software on modern multi-core processors. *Advances in Engineering Software*, 2015.

[18] Francois Broquedis, Jérôme Clet-Ortega, Stéphanie Moreaud, Nathalie Furmento, Brice Goglin, Guillaume Mercier, Samuel Thibault, and Raymond Namyst. hwloc: A generic framework for managing hardware affinities in HPC applications. In *Proceedings of the 18th Euromicro International Conference on Parallel, Distributed and Network-Based Processing*, pages 180–186. IEEE, 2010.

[19] François Broquedis, Nathalie Furmento, Brice Goglin, Raymond Namyst, and Pierre-André Wacrenier. Dynamic task and data placement over NUMA architectures: an OpenMP runtime perspective. In *Evolving OpenMP in an Age of Extreme Parallelism*, pages 79–92. Springer, 2009.

[20] François Broquedis, Nathalie Furmento, Brice Goglin, Pierre-André Wacrenier, and Raymond Namyst. ForestGOMP: an efficient OpenMP environment for NUMA architectures. *International Journal of Parallel Programming*, 38(5-6):418–439, 2010.

[21] J Mark Bull and Chris Johnson. Data distribution, migration and replication on a cc-NUMA architecture. In *Proceedings of the fourth European workshop on OpenMP*, 2002.

[22] Carlos Carvalho. The gap between processor and memory speeds. In *Proc. of the International Conference on Control and Automation*, 2002.

[23] Jonathan Corbet. AutoNUMA: the other approach to NUMA scheduling. http://lwn.net/Articles/488709.

[24] Jonathan Corbet. Transparent huge pages in 2.6.38. https://lwn.net/Articles/423584.

[25] Eduardo H. M. Cruz, Matthias Diener, Marco A. Z. Alves, Laércio L. Pilla, and Philippe O. A. Navaux. Optimizing memory locality using a locality-aware page table. In *26th International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD '14)*, pages 198–205. IEEE, 2014.

[26] Mohammad Dashti, Alexandra Fedorova, Justin Funston, Fabien Gaud, Renaud Lachaize, Baptiste Lepers, Vivien Quema, and Mark Roth. Traffic management: a holistic approach to memory placement on NUMA systems. In *Proceedings of the 18th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '13)*, pages 381–394. ACM, 2013.

[27] Matthias Diener, Eduardo H. M. Cruz, Marco A. Z. Alves, Mohammad S. Alhakeem, Philippe O. A. Navaux, and Hans-Ulrich Heiß. *Euro-Par 2015: 21st International Conference on Parallel and Distributed Computing*, chapter Locality and Balance for Communication-Aware Thread Mapping in Multicore Systems, pages 196–208. Springer, 2015.

[28] Matthias Diener, Eduardo H. M. Cruz, Philippe O. A. Navaux, Anselm Busse, and Hans-Ulrich Heiß. kMAF: Automatic kernel-level management of thread and data affinity. In *Proceedings of the 23rd International Conference on Parallel Architectures and Compilation Techniques (PACT '14)*, pages 277–288. ACM, 2014.

[29] Ulrich Drepper. What every programmer should know about memory. 2007.

[30] Hesham El-Rewini and Mostafa Abd-El-Barr. *Advanced computer architecture and parallel processing*, volume 42 of *Wiley Series on Parallel and Distributed Computing*. Wiley, 2005.

[31] Brice Goglin and Nathalie Furmento. Enabling high-performance memory migration for multithreaded applications on Linux. In *Parallel & Distributed Processing, 2009. IPDPS 2009. IEEE International Symposium on*, pages 1–9. IEEE, 2009.

[32] Mel Gorman. Foundation for automatic NUMA balancing. http://lwn.net/Articles/523065.

[33] John L. Hennessy and David A. Patterson. *Computer Architecture: A Quantitative Approach*. The Morgan Kaufmann Series in Computer Architecture and Design. Morgan Kaufmann, 5th edition, 2011.

[34] Andi Kleen. A NUMA API for Linux. *Novel Inc*, 2005.

[35] Christoph Lameter. Local and remote memory: Memory in a Linux/NUMA system. In *Linux Symposium*, 2006.

[36] Christoph Lameter. NUMA (non-uniform memory access): An overview. *Queue*, 11(7), 2013.

[37] Chris Lattner and Vikram Adve. The LLVM instruction set and compilation strategy. *CS Dept., Univ. of Illinois at Urbana-Champaign, Tech. Report UIUCDCS*, 2002.

[38] Chris Lattner and Vikram Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *International Symposium on Code Generation and Optimization (CGO)*, pages 75–86. IEEE, 2004.

[39] Yong Li, Ahmed Abousamra, Rami Melhem, and Alex Jones. Compiler-assisted data distribution for chip multiprocessors. In *Proceedings of the 19th International Conference on Parallel Architectures and Compilation Techniques (PACT '10)*, pages 501–512. ACM, 2010.

[40] Henrik Löf and Sverker Holmgren. affinity-on-next-touch: increasing the performance of an industrial PDE solver on a cc-NUMA system. *Proceedings of the 19th annual international conference on Supercomputing*, pages 387–392, 2005.

[41] Zoltan Majo and Thomas R Gross. Matching memory access patterns and data placement for NUMA systems. In *Proceedings of the 10th International Symposium on Code Generation and Optimization*, pages 230–241. ACM, 2012.

[42] Jaydeep Marathe, Vivek Thakkar, and Frank Mueller. Feedback-directed page placement for ccNUMA via hardware-generated memory traces. *Journal of Parallel and Distributed Computing*, 70(12):1204–1219, 2010.

[43] Dimitrios S Nikolopoulos, Theodore S Papatheodorou, Constantine D Polychronopoulos, Jesús Labarta, and Eduard Ayguadé. User-level dynamic page migration for multiprogrammed shared-memory multiprocessors. *Parallel Processing, 2000. Proceedings. 2000 International Conference on*, 95–103, 2000.

[44] Lisa Noordergraaf and Ruud van der Pas. Performance experiences on Sun's Wildfire prototype. In *Proceedings of the 1999 ACM/IEEE conference on Supercomputing (CDROM)*, page 38. ACM, 1999.

[45] Markus Nordén, Henrik Löf, Jarmo Rantakokko, and Sverker Holmgren. Geographical locality and dynamic data migration for OpenMP implementations of adaptive PDE solvers. In *OpenMP Shared Memory Parallel Programming*, pages 382–393. 2008.

[46] Takeshi Ogasawara. NUMA-Aware memory manager with Dominant-Thread-Based copying GC. In *Proceedings of the 24th ACM SIGPLAN Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA '09)*, pages 377–390. ACM, 2009.

[47] Guilherme Piccoli, Henrique Nazaré Santos, Raphael Ernani Rodrigues, Christiane Pousa, Edson Borin, and Fernando Magno Quintão Pereira. Compiler support for selective page migration in NUMA architectures. In *Proceedings of the 23rd International Conference on Parallel Architectures and Compilation Techniques (PACT '14)*, pages 369–380. ACM, 2014.

[48] Christiane Pousa Ribeiro, Jean-François Méhaut, and Alexandre Carissimi. Memory affinity management for numerical scientific applications over multi-core multiprocessors with hierarchical memory. In *IPDPS Workshops*, 2010.

[49] Mustafa M Tikir and Jeffrey K Hollingsworth. Using hardware counters to automatically improve memory performance. In *Supercomputing, 2004. Proceedings of the ACM/IEEE SC2004 Conference*, pages 46–46. IEEE, 2004.

[50] Rik van Riel and Vinod Chegu. Automatic NUMA balancing. In *Red Hat Summit (KVM Forum)*, April 2014.

[51] Ben Verghese, Scott Devine, Anoop Gupta, and Mendel Rosenblum. Operating system support for improving data locality on CC-NUMA compute servers. In *Proceedings of the 7th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS VII)*, pages 279–289. ACM, 1996.

[52] Kenneth M Wilson and Bob B Aglietti. Dynamic page placement to improve locality in CC-NUMA multiprocessors for TPC-C. *Proceedings of the 2001 ACM/IEEE conference on Supercomputing (CDROM)*, pages 33–33, 2001.

[53] Markus Wittmann and Georg Hager. Optimizing ccNUMA locality for task-parallel execution under OpenMP and TBB on multicore-based systems. *arXiv preprint arXiv:1101.0093*, 2010.