UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
INSTITUTO DE INFORMÁTICA
PROGRAMA DE PÓS-GRADUAÇÃO EM COMPUTAÇÃO

ALEXANDRE VINÍCIUS ALMEIDA

Uso de Auto-tuning para Otimização de Decomposição de Domínios Paralela

Dissertação apresentada como requisito parcial
para a obtenção do grau de
Mestre em Ciência da Computação

Prof. Dr. Nicolas Maillard
Orientador

Porto Alegre, dezembro de 2011

CIP – CATALOGAÇÃO NA PUBLICAÇÃO

Almeida, Alexandre Vinícius

Uso de Auto-tuning para Otimização de Decomposição de Domínios Paralela / Alexandre Vinícius Almeida. – Porto Alegre: PPGC da UFRGS, 2011.

57 f.: il.

Dissertação (mestrado) – Universidade Federal do Rio Grande do Sul. Programa de Pós-Graduação em Computação, Porto Alegre, BR–RS, 2011. Orientador: Nicolas Maillard.

1. Auto-tuning. 2. Decomposição de domínios. 3. MPI. 4. Paralelismo. I. Maillard, Nicolas. II. Título.

SUMÁRIO

LISTA DE ABREVIATURAS E SIGLAS

AEOS	Automated Empirical Optimization of Software
ANSI	American National Standards Institute
API	Application Programming Interface
ATLAS	Automatically Tuned Linear Algebra Software
BLAS	Basic Linear Algebra Subprograms
CPU	Central Processing Unit
DAG	Directed Acyclic Graph
DD	Decomposição de Domínios
DFT	Discrete Fourier Transform
EDP	Equação Diferencial Parcial
FFTW	Fastest Fourier Transform in the West
GB	Gigabyte
GEMM	GEneral Matrix Multiply
GPGPU	General-Purpose computing on Graphics Processing Units
GHz	Gigahertz
II-UFRGS	Instituto de Informática-UFRGS
MB/s	Megabyte por segundo
MDF	Método de Diferenças Finitas
MPI	Message Passing Interface
PETSc	Portable, Extensible Toolkit for Scientific Computation
PHiPAC	Portable High Performance Ansi C
POSIX	Portable Operating System Interface for Unix
UFRGS	Universidade Federal do Rio Grande do Sul

LISTA DE FIGURAS

RESUMO

O desenvolvimento de aplicações de forma a atingir níveis de desempenho próximos aos níveis teóricos de uma determinada plataforma é uma tarefa que exige conhecimento técnico do ambiente de hardware, uma vez que o software deve explorar detalhes específicos da plataforma em questão. Pelo fato do software ser específico à plataforma, caso ela evolua ou se altere, as otimizações realizadas podem não explorar a nova arquitetura de forma eficiente. *Auto-tuners* são sistemas que surgiram como um meio automatizado de adaptar um determinado software a uma arquitetura alvo. Essa adaptação ocorre através de uma busca empírica de valores ótimos para parâmetros específicos de uma aplicação, a fim de ajustá-los às características do hardware, ou ainda através da geração de código-fonte otimizado para a plataforma. Este trabalho propõe um módulo *auto-tuner* orientado à adaptação parametrizada de uma aplicação paralela, que trabalha variando os fatores da dimensão do domínio bidimensional, o número de processos e a extensão das regiões de sobreposição. Para cada variação dos fatores, o *auto-tuner* testa a aplicação na arquitetura paralela de forma a buscar a combinação de parâmetros com melhor desempenho. Para possibilitar o *auto-tuning*, foi desenvolvida uma classe em linguagem C++ denominada `Mesh`, baseada no padrão MPI. A classe busca abstrair a decomposição de domínios de uma aplicação paralela por meio do uso de Orientação a Objetos, e facilita a variação da extensão das regiões de sobreposição entre os subdomínios. Os resultados experimentais demonstraram que o *auto-tuner* explora o ganho de desempenho pela variação do número de processos da aplicação, que também é tratado pelo módulo *auto-tuner*. A arquitetura paralela utilizada na validação não se mostrou ideal para uma otimização através do aumento da extensão das regiões sobrepostas entre subdomínios.

Palavras-chave: Auto-tuning, decomposição de domínios, MPI, paralelismo.

Optimizing Parallel Domain Decomposition using Auto-tuning

ABSTRACT

Achieving the peak performance level of a particular platform requires technical knowledge of the hardware environment involved, since the software must explore specific details inherent to the hardware. Once the software is optimized for a target platform, if the hardware evolves or is changed, the software probably would not be as efficient in the new environment. This performance portability problem is addressed by *software auto-tuning*, which emerged in the past decade as an automated technique to adapt a particular software to an underlying hardware. The software adaptation is performed by an *auto-tuner*. The auto-tuner is an entity that empirically adjusts specific application parameters in order to improve the overall application performance, or even generates source-code optimized for the target platform. This dissertation proposes an auto-tuner to optimize the domain decomposition of a parallel application that performs stencil computations. The proposed auto-tuner works in a parameterized adaptation fashion, and varies the dimensions of a 2D domain, the number of parallel processes and the extension of the overlapping zones between subdomains. For each combination of parameter values, the auto-tuner probes the application in the parallel architecture in order to seek the best combination of values. In order to make auto-tuning possible, it is proposed a C++ class called `Mesh`, based on the Message Passing Interface (MPI) standard. The role of this class is to abstract the domain decomposition from the application using the Object Orientation facilities provided by C++, and also to enable the extension of the overlapping zones between subdomain. The experimental results showed that the performance gains were mainly due to the variation of the number of processes, which was one of the application factors dealt by the auto-tuner. The parallel architecture used in the experiments showed itself as not adequate for optimizing the domain decomposition by increasing the overlapping zones extension.

Keywords: Auto-tuning, domain decomposition, MPI, paralelism.

1 INTRODUÇÃO

O desenvolvimento de aplicações de forma a atingir níveis de desempenho próximos aos níveis teóricos de uma determinada plataforma é uma tarefa que exige conhecimento técnico do ambiente de hardware. Isso decorre do fato do software ter de explorar detalhes específicos da plataforma em questão, tais como instruções vetoriais e número de *cores* do processador, uso eficiente da memória *cache*, ou mesmo a exploração do poder de processamento de GPGPUs, que eventualmente estejam disponíveis no ambiente. Pelo fato do software ser específico à plataforma, caso ela evolua ou se altere, as otimizações realizadas podem não explorar a nova arquitetura de forma eficiente (WHALEY; PETITET; DONGARRA, 2001, Part I).

Agregados de computadores (*clusters*) tornaram-se arquiteturas paralelas indispensáveis para o processamento de aplicações de larga escala, como aplicações que envolvem o processamento de modelos físicos (WALKO et al., 2000). Quando o problema de portabilidade de desempenho de uma aplicação é estendido para *clusters* de computadores, ele agrava-se pela heterogeneidade existente entre máquinas paralelas. Considerando ainda a complexidade deste tipo de máquina, muitos novos parâmetros de otimização são introduzidos, e podem ser considerados no momento da adaptação do software (ASANOVIC et al., 2009).

Auto-tuners são sistemas que surgiram como um meio automatizado de adaptar um determinado software a uma arquitetura alvo. Essa adaptação ocorre através de uma busca empírica de valores ótimos para parâmetros específicos de uma aplicação, a fim de ajustá-los às características do hardware, ou ainda através da geração de código-fonte otimizado para a plataforma em questão.

Este trabalho aplica a técnica de *auto-tuning* para a adaptação de aplicações paralelas que processam domínios estruturados bidimensionais, decompostos entre os processos de uma aplicação.

O processo de Decomposição de Domínios, em computação paralela, refere-se à distribuição dos dados de um modelo computacional entre os processadores de uma máquina de memória distribuída, processo este normalmente empregado para o processamento paralelo de Equações Diferenciais Parciais (SMITH; BJØRSTAD; GROPP, 1996). A decomposição do domínio traz como consequência uma natural dependência de dados entre os processos, que ocorre nas regiões de fronteira dos subdomínios. Para contornar a dependência direta, cada processo mantém as regiões de fronteira de seus processos vizinhos em zonas do seu respectivo subdomínio, denominadas de *regiões de sobreposição*.

Como os processos da aplicação devem calcular os pontos de seu subdomínio e, de tempos em tempos, trocar os valores localizados nas fronteiras com os respectivos processos vizinhos, o acesso à rede para a troca de dados entre os processos é frequente. Uma técnica existente para minimizar a comunicação entre processos é efetuar o aumento da

extensão das regiões de sobreposição entre os subdomínios, de um valor $s = 1$ para $s > 1$. Dessa forma, posterga-se a comunicação dos valores de fronteira a cada s iterações, ao custo de cada processo recalcular as fronteiras de seus vizinhos que, de outra forma, deveriam ser comunicadas. O parâmetro s é o fator da decomposição do domínio cujo valor ótimo será buscado pelo *auto-tuner* em um agregado de máquinas.

1.1 Proposta do trabalho

Este trabalho propõe um módulo *auto-tuner* orientado à adaptação parametrizada de uma aplicação paralela, que trabalha variando os fatores da dimensão do domínio bidimensional, o número de processos e a extensão das regiões de sobreposição. Para cada variação dos fatores, o *auto-tuner* testa a aplicação na arquitetura paralela de forma a buscar a combinação de parâmetros com melhor desempenho. Para possibilitar a variação da extensão das regiões de sobreposição, foi proposta uma classe em linguagem C++ denominada `Mesh`, que se baseia no padrão *de facto* MPI. A classe busca abstrair a decomposição de domínios de uma aplicação paralela por meio do uso de Orientação a Objetos, e expõe a extensão das regiões sobrepostas na forma de um parâmetro, tornando possível o nivelamento pelo *auto-tuner*.

1.2 Organização do texto

A dissertação segue a seguinte estrutura: o Capítulo 2 apresenta o conceito de *auto-tuning* de software e o estado da arte de aplicações que fazem uso dessa técnica. Como o trabalho tem o objetivo de aplicar *auto-tuning* sobre a decomposição de um domínio bidimensional, o Capítulo 3 expõe os conceitos associados com decomposição de domínios de aplicações numéricas paralelas, e especifica a origem da dependência de dados entre os subdomínios. Neste capítulo também são expostos trabalhos que aplicam *auto-tuning* sobre domínios decompostos. O Capítulo 4 apresenta a classe `Mesh`, que abstrai a decomposição de domínios da aplicação e possibilita o *auto-tuning* de software. O funcionamento do módulo *auto-tuner* desenvolvido é também apresentado neste capítulo. As validações do aumento da extensão das regiões de sobreposição e do funcionamento do *auto-tuner* são expostas no Capítulo 5 e, por fim, o Capítulo 6 apresenta a conclusão e a discussão final da dissertação.

2 AUTO-TUNING

A técnica de *auto-tuning* surgiu como um meio de possibilitar ao software acompanhar a evolução do hardware sem a necessidade de intervenção manual. A ideia de *auto-tuning*, cuja definição é apresentada na Seção 2.1, surgiu a partir do desenvolvimento de bibliotecas numéricas, em especial pela biblioteca ATLAS, que contribuiu com o paradigma AEOS, descrito na Seção 2.2. Além da biblioteca ATLAS, as bibliotecas autoadaptadas[1] PHiPAC e FFTW são consideradas como outras duas bibliotecas numéricas sequenciais pioneiras, e são apresentadas na Seção 2.3. Por fim, a Seção 2.4 relata as considerações gerais deste capítulo.

2.1 Definição

Auto-tuning é uma técnica de otimização de software concebida a partir do contexto de bibliotecas numéricas sequenciais que, pela sua natureza *CPU bound*, têm a necessidade de extrair o máximo desempenho que uma arquitetura alvo oferece. Nessa técnica aplicada a software numérico, um *auto-tuner* tem a tarefa de buscar, dentro de um espaço de possibilidades, a implementação de uma rotina *kernel* que apresente melhor desempenho em uma determinada arquitetura (WILLIAMS, 2008, Seção 2.4). Rotinas *kernel* são porções de código que resolvem determinadas operações numéricas e integram outros procedimentos na resolução de operações mais complexas. Com isso, a otimização de uma rotina *kernel* em uma arquitetura reflete no desempenho de várias outras operações numéricas pertencentes à biblioteca autoadaptada (WHALEY; PETITET; DONGARRA, 2001, Capítulo 1).

De acordo com Williams (2008, Seção 2.4), *auto-tuning* relaciona-se com três conceitos principais, que são etapas na busca pelo *kernel* que apresente o melhor desempenho: espaço de otimização, geração de código e exploração do espaço de otimização.

2.1.1 Espaço de otimização e geração de código

O espaço de otimização se refere ao conjunto de valores de parâmetros do *kernel* que serão explorados para a busca da combinação que apresente o melhor desempenho. Através da Figura 2.1, Williams (2008, Seção 2.4.1) representa os espaços de busca de acordo com a evolução dos *auto-tuners* no decorrer dos anos.

A classificação toma como base o nível dos parâmetros explorados. No primeiro grupo de *auto-tuners* situam-se aqueles que exploram parâmetros do *kernel* mais próximos da arquitetura, como o uso de instruções vetoriais SIMD, eliminação de desvios condicionais e otimização de laços (por exemplo, *loop unrolling*). No segundo grupo, além dos parâ-

[1] O termo "autoadaptação" é usado como um sinônimo de *auto-tuning* no decorrer do texto.

metros explorados pelo primeiro grupo, os *auto-tuners* experimentam estruturas de dados alternativas ao algoritmo, ou ainda tipos ou *layouts* de dados alternativos. No terceiro e último grupo estão contidos os *auto-tuners* de mais alto nível, que exploram também algoritmos alternativos ao *kernel* sendo autoadaptado.

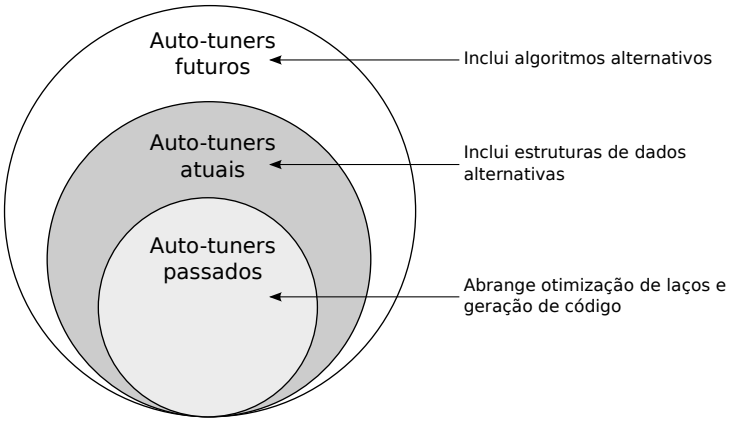


Figura 2.1: Espaços de otimização das classes de *auto-tuners*

É importante notar que em todos os três grupos, compiladores tradicionais teriam dificuldade em adaptar o *kernel* pela pouca informação que um código-fonte normalmente transmite sobre o algoritmo expressado.

Tomando como base os parâmetros de um determinado espaço de otimização, o *auto-tuner* geraria mutações do *kernel* por meio de um gerador de código em uma linguagem de alto nível (Ruby ou Python, por exemplo). A geração de código poderia englobar um subconjunto ou todas as possibilidades do espaço de busca. Embora o processo de geração de código possa produzir um número proibitivo de mutações, a escolha de qual mutação apresenta melhor desempenho pode ser feita através de um método de exploração que restrinja as possibilidades do espaço de busca, como exposto na Seção 2.1.2 seguinte.

2.1.2 Exploração do espaço de otimização

A busca no espaço de otimização pode ser feita através de três abordagens principais (WILLIAMS, 2008, Seção 2.4.1). A mais simples e direta é a busca exaustiva, a qual testa a rotina *kernel* na arquitetura aplicando todas as possibilidades de otimização dentro do espaço de busca. Nessa abordagem, dependendo da explosão combinatória, o espaço de busca pode tornar a adaptação inviável em termos de tempo.

A Figura 2.2, extraída de (WILLIAMS, 2008, Seção 2.4.1), representa os três métodos para exploração do espaço de busca considerando dois parâmetros A e B. As curvas apresentadas nos gráficos denotam faixas de valores onde o desempenho é constante, e a estrela representa a escolha que apresenta melhor desempenho. O comportamento da busca exaustiva é demonstrado pela Figura 2.2(a), onde todo o espaço de busca é coberto. Consequentemente, esse método é bastante custoso em termos de tempo, embora tenha condições de encontrar os valores ótimos para os parâmetros.

Uma forma de reduzir o espaço de busca é através do uso de heurísticas baseadas na arquitetura onde o *auto-tuning* será realizado. Por meio disso, restringe-se o intervalo de valores de certos parâmetros de otimização dentro de limites pré-definidos. Entretanto, heurísticas podem ser imprecisas na geração de espaços de busca que contenham "boas" combinações de valores de parâmetros. Além disso, em geral apenas um subconjunto de parâmetros de otimização podem ser restringidos por heurísticas, sendo necessá-

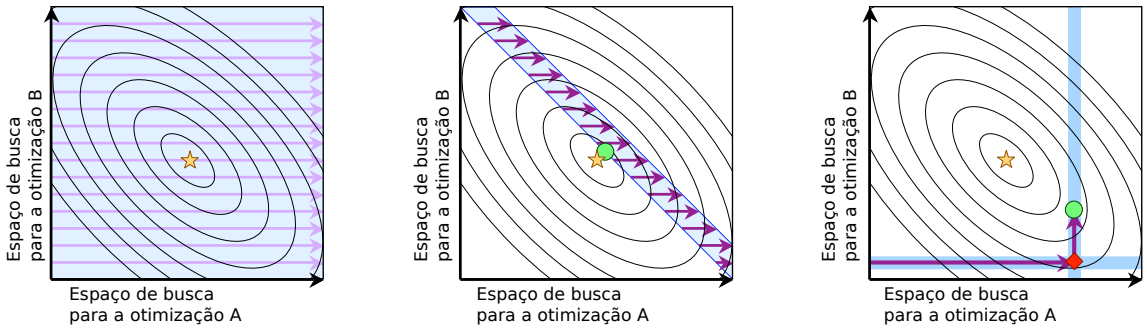


Figura 2.2: Representação dos métodos de busca: (a) busca exaustiva, (b) busca heurística, e (c) *hill climbing*

rio, portanto, realizar buscas adicionais para o restante das otimizações. A Figura 2.2(b) apresenta a abordagem heurística, onde a busca é realizada somente sobre uma faixa de valores, sendo o desempenho resultante (círculo) uma aproximação ao ótimo.

Uma terceira técnica para explorar o espaço de busca é o uso do método guloso *hill climbing*. Nessa abordagem, o *auto-tuner* efetua uma busca linear dentro de todo o espaço de busca de um determinado parâmetro, mantendo fixos os valores dos outros parâmetros. Ao encontrar o valor que apresenta melhor desempenho para o parâmetro sendo testado, o *auto-tuner* repete o processo para um próximo parâmetro, sempre mantendo fixos os demais parâmetros. Portanto, assume-se que o ótimo local de um parâmetro é, também, o ótimo global para o sistema, sendo este um requisito básico para a adoção de uma estratégia gulosa (CORMEN et al., 2002, Capítulo 16). Entretanto, devido a interações implícitas entre parâmetros, o *hill climbing* pode resultar em escolhas de valores muito distantes do ótimo global (DATTA, 2009, Seção 5.4.3).

A Figura 2.2(c) ilustra a execução do método *hill climbing*, onde o parâmetro B é mantido fixo em um valor e todo o espaço de busca de A é explorado. O losango representa o melhor desempenho obtido através da variação de A. Partindo deste ponto, o *auto-tuner* explora todo o espaço de busca de B, mantendo fixo o valor selecionado para A. O círculo representa o ponto no espaço de busca que apresentou melhor desempenho variando B e mantendo A fixo.

2.2 Paradigma AEOS

O paradigma AEOS (*Automated Empirical Optimization of Software*) (WHALEY; PETITET; DONGARRA, 2001) se trata de um conjunto de orientações para *auto-tuning* de bibliotecas numéricas, que foi concebido para a autoadaptação da biblioteca ATLAS (Seção 2.3.1). De acordo com a metodologia AEOS, existem duas maneiras básicas para efetuar a adaptação automática de um software em uma plataforma: por *adaptação parametrizada* ou por *adaptação de código-fonte*.

2.2.1 Adaptação parametrizada

Neste método de *auto-tuning*, pontos específicos do código-fonte são parametrizados, de forma a ser possível tornar o software mais eficiente em uma dada arquitetura, apenas pelo ajuste dos valores dos parâmetros. No paradigma AEOS, os valores para os parâ-

metros seriam determinados em tempo de compilação, onde o software seria executado sucessivas vezes variando-se seus valores. Por meio dos resultados de tempo que seriam obtidos, o conjunto de valores que apresentasse o resultado mais eficiente seria escolhido.

2.2.2 Adaptação de código-fonte

A adaptação de código-fonte é feita através da geração de diferentes implementações para o *kernel*. Esse método se divide em duas técnicas, dependendo de como as diferentes implementações são obtidas, por *múltiplas implementações* ou por *geração de código*.

Na técnica de *múltiplas implementações*, a adaptação ocorre através da escolha de uma dentre várias alternativas de código, todas orientadas a resolver uma operação específica. A escolha seria realizada através da medição das várias alternativas, escolhendo-se aquela que apresentasse melhores resultados. Pelo fato dessa técnica focar em uma parte bastante pontual do software, especialistas poderiam concentrar-se em programar tal operação de forma otimizada para um determinado hardware, sem ser necessária uma compreensão do restante da arquitetura do software. Para que essa técnica seja efetiva, a base de desenvolvedores deve ser larga e bastante especializada.

A segunda técnica de adaptação de código-fonte é feita por *geração de código*. Essa técnica é a mais flexível dentre as mencionadas, uma vez que o próprio software torna-se capaz de gerar código-fonte otimizado para a arquitetura, estendendo a adaptação a outras características do hardware. Por exemplo, a adaptação a um determinado *pipeline* de ponto-flutuante é possível apenas através da geração de um código-fonte, onde outras instruções são intercaladas entre as operações de ponto-flutuante, "escondendo", dessa forma, a latência do *pipeline*.

2.2.3 Orientações gerais

Para que uma biblioteca numérica possa implementar a autoadaptação de suas rotinas, o paradigma AEOS defende as seguintes orientações gerais, que têm o objetivo de dar condições para que a aplicação seja adaptada automaticamente na plataforma:

- Isolamento de *kernels*: identificação e isolamento de operações numéricas básicas, que compõe o núcleo de outras rotinas mais complexas da biblioteca. Essas rotinas serão aquelas a serem autoadaptadas;

- Simulação de cenários para tomadas de tempo: uma vez que a autoadaptação é feita com base no tempo de processamento das variações das rotinas centrais, é importante que as condições do estado da arquitetura quando as rotinas serão chamadas sejam simuladas no momento do *auto-tuning*. Por exemplo, caso se tenha conhecimento que uma determinada rotina será frequentemente invocada com a memória *cache* do processador vazia, antes de cada medição de tempo o *auto-tuner* deve, de alguma forma, esvaziar a memória *cache* a fim de simular o estado real da máquina;

- Escolha de *timers* precisos: como o tempo de processamento é o fator chave para que o *auto-tuner* escolha a melhor variação do *kernel*, é importante que as funções responsáveis por capturar o tempo de processamento sejam precisas e confiáveis;

- Adoção de heurísticas de busca apropriadas: heurísticas devem ser adotadas a fim de eliminar ramos da árvore de busca, acelerando, assim, a procura pelos valores ótimos dos parâmetros, ou pelo código que apresente melhor desempenho, no caso

de geração automática de código. Além do mais, como a busca pode ser um procedimento demorado, o *auto-tuner* deve tolerar falhas que venham a interromper o processo de *auto-tuning*, de forma a não reiniciar a busca do começo.

2.3 Estado da arte

Esta seção apresenta os aspectos técnicos de *auto-tuning* das bibliotecas científicas ATLAS, PHiPAC e FFTW.

2.3.1 ATLAS

A biblioteca ATLAS (*Automatically Tuned Linear Algebra*) (WHALEY; PETITET, 2005) é uma biblioteca de funções em linguagem C orientada à resolução de operações da álgebra linear, tais como multiplicações matriciais e vetoriais. Essa biblioteca implementa as rotinas da API BLAS (*Basic Linear Algebra Software*) sobre os princípios do paradigma AEOS.

O conjunto de rotinas para álgebra linear especificadas pela BLAS são divididas em três níveis, todos implementados na biblioteca ATLAS: no Nível 1 situam-se rotinas que realizam operações entre vetores; no Nível 2 estão contidas todas as rotinas que efetuam operações entre matrizes e vetores; por fim, o Nível 3 abrange as funções que realizam procedimentos entre matrizes (DONGARRA, 2002).

A biblioteca ATLAS efetua a autoadaptação das rotinas contidas nos Níveis 2 e 3 da BLAS, uma vez que as operações desses níveis podem tirar proveito de otimizações relacionadas, principalmente, à localidade de referência dos dados em memória *cache*. Com relação às rotinas do Nível 1, como as possíveis otimizações de código nesse nível envolvem o uso das unidades de ponto-flutuante e técnicas de otimização de laços, a adaptação à arquitetura é repassada às estratégias de otimização do próprio compilador.

No caso das rotinas do Nível 3, o ganho de desempenho é obtido através da adaptação do *kernel* de multiplicação de matrizes GEMM (*GEneral Matrix Multiply*), rotina essa central a todas operações contidas neste nível e adaptada tanto por meio de parâmetros quanto através de geração de código. Esse *kernel* trata-se de uma multiplicação de matrizes recursiva, que transforma as matrizes envolvidas em blocos de dados de tamanho N_b contíguos em memória. A partir do momento que a recursão atinge um fator de bloco N_b, as submatrizes são multiplicadas através do algoritmo convencional, de complexidade $O\left(n^3\right)$.

A transformação das matrizes em blocos tem um custo de $O\left(n^2\right)$, que pode dominar o tempo de execução caso as matrizes que farão parte do cálculo não sejam grandes o suficiente. Portanto, a biblioteca determina um *tamanho mínimo* para as matrizes tal que compense o custo da transformação em blocos. Caso os operandos sejam menores que o tamanho mínimo estabelecido, a multiplicação é realizada diretamente sobre os operandos passados por parâmetro à função, sem efetuar as transformações em blocos.

O tamanho mínimo dos operandos é um parâmetro cujo valor é estabelecido no momento da compilação da biblioteca. A ATLAS determina esse ponto de corte comparando as multiplicações, *com* e *sem* a transformação dos operandos, variando o tamanho das matrizes até que a abordagem que realiza a transformação apresente ganho de desempenho.

Com relação ao parâmetro N_b, seu valor é determinado de acordo com o tamanho da *cache* de nível 1 (L1) do processador. Para tanto, o *auto-tuner* determina empiricamente o tamanho da *cache* L1, iniciando com um determinado tamanho de bloco e sucessivamente reduzindo-o. Os tempos de processamento para cada tamanho são anotados, e a maior

diferença de tempo entre dois tamanhos sucessivos é assumida como sendo o limite do tamanho da *cache*. A fim de agilizar a busca, apenas tamanhos em potências de 2 são considerados.

2.3.2 PHiPAC

Em (BILMES et al., 1998), é apresentada uma série de orientações para a codificação de *kernels* em linguagem ANSI C, que visam auxiliar a extração do máximo desempenho de arquiteturas, mantendo a possibilidade de adaptação dessas rotinas, a fim de torná-las portáveis. A essa metodologia de desenvolvimento deu-se o nome de PHiPAC (*Portable, High Performance ANSI C*).

Dentre as várias orientações para codificação de software autoadaptável em linguagem C estão as seguintes: eliminação de falsas dependências entre instruções; conversão de multiplicações por adições; desenrolar laços de forma explícita, a fim de expor oportunidades de otimização; minimização de desvios para evitar instruções de comparação e esvaziamento de *pipeline*.

Utilizando as orientações apresentadas, foi proposto o gerador `mm_gen` que produz código em linguagem C para o *kernel* GEMM da BLAS. O gerador de código reestrutura o algoritmo para efetuar uma multiplicação em blocos, aproveitando a localidade em *cache* dos dados das matrizes. O gerador é capaz de produzir código para aproveitar a localidade de vários níveis da hierarquia de memória, como registradores, *cache* L1, *cache* L2, dentre outros níveis.

O procedimento GEMM efetua a operação $C = \alpha op\,(A)\, op\,(B) + \beta C$, onde α e β são escalares quaisquer, $op\,(A)$ e $op\,(B)$ e C são matrizes de tamanho $M \times K$, $K \times N$ e $M \times N$, respectivamente, e $op\,(X)$ pode ser a operação $transposta\,(X)$, ou apenas a matriz X inalterada. O gerador `mm_gen` sintetiza a operação GEMM dados os tamanhos de blocos para cada nível de *cache*.

Um exemplo de chamada ao `mm_gen` seria:

```
mm_gen -cb M0 K0 N0 -cb M1 K1 N1
```

onde $M0$, $K0$ e $N0$ referem-se ao tamanho do bloco para os registradores, e $M1$, $K1$ e $N1$ especificam o tamanho do bloco para a *cache* L1. De uma forma geral, para um dado nível de memória i, a multiplicação em blocos acumularia em uma matriz destino C de tamanho $M_i \times N_i$, a multiplicação das matrizes A e B de tamanho $M_i \times K_i$ e $K_i \times N_i$, respectivamente.

A determinação dos tamanhos de blocos é feita por um *script* de busca codificado em linguagem Perl, que assume o papel de *auto-tuner* da biblioteca. Diferentemente da ATLAS, o *auto-tuner* da PHiPAC possui um arquivo de parâmetros onde são especificadas as características da arquitetura, tais como o número de registradores para valores inteiros e ponto-flutuante, e os tamanhos das *caches* L1, L2 e L3 (opcional para os níveis L2 e L3). A invocação do *auto-tuner* possui a seguinte forma:

```
search.pl [-long|-default|-short] -machine machine_specs \
          -prec [single|double] -level [0|1|2]
```

onde o parâmetro `-machine` especifica as configurações da arquitetura de acordo com o arquivo `machine_specs`; o parâmetro `-prec` informa se os dados de entrada devem ser tratados em precisão simples ou dupla; e o parâmetro `-level` informa o nível da hierarquia de memória para o qual a busca será feita (0 para registrador, 1 para *cache* L1

e 2 para *cache* L2). O parâmetro `-level` aceita, também, valores maiores que 2 para hierarquias mais altas de memória. Caso o tamanho de algum nível de *cache* não seja especificado em `machine_specs` (com exceção da L1), o *auto-tuner* tentará determinar o tamanho com base no tamanho do nível imediatamente inferior.

Os parâmetros `[-long|-default|-short]` especificam se o espaço de busca deve ser explorado de forma exaustiva (`long`) ou de uma forma mais superficial (`short`). A escolha de um ou outro impacta diretamente no tempo de busca, sendo a opção `default` um meio termo entre `long` e `short`.

A busca pelos tamanhos de blocos é feita pelo *auto-tuner*, que varia os parâmetros do gerador de código `mm_gen` e testa cada código-fonte produzido. A busca começa pelo tamanho de bloco para os registradores (nível 0), variando os parâmetros M_0 e N_0, cujo espaço de busca fica no intervalo $1 \leq M_0 N_0 \leq NR$, onde NR é a quantidade de registradores de ponto-flutuante especificada no arquivo `machine_specs`. O intervalo de valores para K_0 varia de $1 \leq K_0 \leq 20$. Segundo os autores, empiricamente, valores maiores que 20 nunca mostram ganhos de desempenho e, portanto, pode-se assumir esse valor como limite para K_0 (BILMES et al., 1998, Seção 4.1). Assim que um tamanho de bloco para o nível 0 seja definido, a busca prossegue para os níveis mais altos de memória.

A medida que a busca é realizada, são criados arquivos de *checkpoint* que informam o desempenho alcançado com uma dada configuração de tamanho de bloco. Dessa forma, o *auto-tuner* tem condições de retomar a busca a partir do último *checkpoint* em caso de falhas.

2.3.3 FFTW

A FFTW (*Fastest Fourier Transform in the West*) (FRIGO; JOHNSON, 2005) é uma implementação da transformada discreta de Fourier (DFT) que cria um *plano de execução* otimizado para o hardware antes de realizar o cálculo da transformada. A entrada para uma DFT é representada através de um grafo acíclico direcionado (DAG), sendo o plano de execução composto por *codelets*, os quais podem ser considerados os *kernels* da FFTW. Os *codelets* são fragmentos de código em linguagem C que calculam uma DFT com um tamanho fixo, e são empregados para o cálculo de partições da DAG.

Os *codelets* podem ser produzidos por um gerador de código denominado `genfft` (FRIGO, 1999), distribuído conjuntamente com a FFTW. Embora exista um gerador específico para os *codelets*, a autoadaptação é feita apenas pela produção dos planos de execução, que utiliza uma base de, aproximadamente, 150 *codelets* distribuídos com a biblioteca.

O usuário interage com a FFTW em dois estágios: no primeiro estágio, é feita uma chamada ao módulo de planejamento (módulo *planner*), que, a partir de uma descrição do problema, cria um plano de execução adaptado ao hardware; no segundo estágio, é feito o processamento da DFT através do plano criado no primeiro estágio. Portanto, o *auto-tuning* é realizado em tempo de execução, porém não de forma dinâmica.

O *planner*, quando invocado, gera uma série de planos de execução que decompõe a DAG recursivamente em subproblemas, realizando medições de desempenho de cada plano a fim de escolher aquele que apresente o melhor desempenho. A Figura 2.3 é uma adaptação da figura apresentada em (WILLIAMS, 2008, Capítulo 2), e representa três diferentes planos de execução para uma DFT de 8 pontos, onde os números indicam as partições da DAG, bem como a ordem de execução do problema.

Na Figura 2.3, cada caixa com bordas seccionadas representa uma partição da DAG, que é um subproblema menor (ou igual) da DFT original, a ser resolvido por um *codelet*.

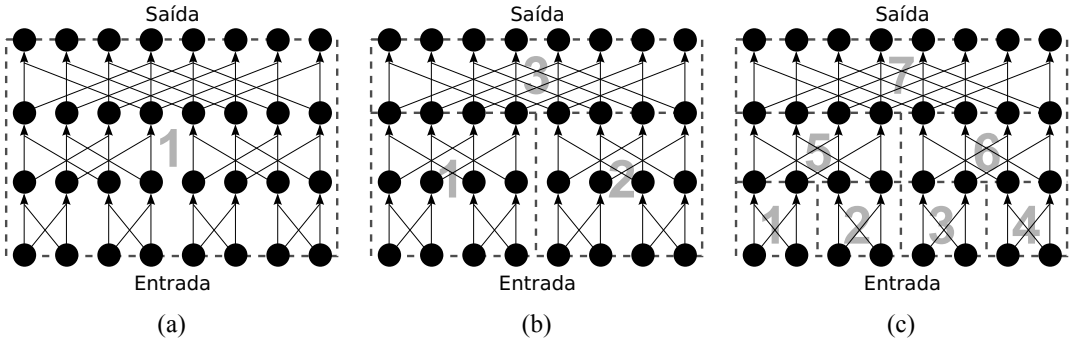


Figura 2.3: Exemplos de planos de execução para uma DFT de 8 pontos

A Figura 2.3(a) representa a resolução direta da DFT de 8 pontos. Já na Figura 2.3(b) o *planner* determinou que a maneira mais eficiente seria executar primeiro as DFTs 1 e 2, ambas de 4 pontos, para então executar a DFT 3, de 8 pontos. Na Figura 2.3(c) o problema é dividido ainda mais, de forma a ter a resolução de 7 subproblemas menores até a solução final.

Pela subdivisão recursiva da DAG em problemas menores, a FFTW utiliza técnicas de algoritmos *cache oblivious*, de forma a explorar a localidade de dados em hierarquias de memória mais baixas, sem possuir informações sobre o tamanho destes níveis de *cache*. Em contraste, seguindo a classificação de (FRIGO et al., 1999), as bibliotecas ATLAS e PHiPAC podem ser consideradas *cache aware*, uma vez que ambas necessitam obter, de alguma forma, a informação do tamanho da *cache* a fim de adaptar seus *kernels*.

2.4 Considerações do capítulo

Auto-tuning se mostra como uma técnica de otimização eficaz e bem estabelecida para a adaptação de rotinas de bibliotecas numéricas em arquiteturas sequenciais. Os métodos e definições de *auto-tuning* encontrados ainda são focados na adaptação de *kernels* de bibliotecas numéricas, principalmente por essa técnica ter nascido dentro da classe de software numérico.

No contexto deste trabalho, *auto-tuning* será aplicado em software paralelo que trabalha com Decomposição de Domínios estruturados. O Capítulo 3, a seguir, aborda as noções e conceitos de Decomposição de Domínios em software paralelo, além de apresentar trabalhos que tiram proveito da técnica de *auto-tuning* para otimizar o processamento de domínios estruturados em arquiteturas paralelas.

3 DECOMPOSIÇÃO DE DOMÍNIOS ESTRUTURADOS

Para que um domínio físico possa ser processado por um computador, ele deve ser transformado de sua forma física contínua para uma representação discreta, composta por um conjunto finito de pontos. Uma vez o domínio discretizado, torna-se possível simular computacionalmente um determinado fenômeno físico através de Equações Diferenciais Parciais (EDP).

Este trabalho concentra-se no processamento de EDPs sobre domínios estruturados, apresentado na Seção 3.1, bem como na sua decomposição em ambientes de memória distribuída, abordada na Seção 3.2. Assim como um domínio deve ser discretizado, uma EDP também deve sofrer uma transformação de sua forma contínua para uma representação discreta. Esse processo é descrito na Seção 3.3, que serve de base para discutir as noções de estêncil e região de sobreposição na Seção 3.4.

O estado da arte, na Seção 3.5, apresenta a biblioteca PETSc e trabalhos que abordam *auto-tuning* envolvendo decomposição de domínios estruturados. O fechamento do capítulo é apresentado na Seção 3.6.

3.1 Classes de domínios

Dependendo do tipo de domínio físico, os domínios uma vez discretizados podem assumir uma forma estruturada ou não estruturada. Domínios estruturados são aqueles onde cada ponto possui a mesma quantidade de pontos vizinhos. Nesse tipo de domínio, cada ponto é indexado em uma linha, coluna e profundidade no domínio (profundidade para o caso de domínios tridimensionais). Domínios estruturados são representados computacionalmente por meio de matrizes densas. Portanto, a varredura dos pontos e o acesso aos respectivos pontos vizinhos são feitos através do incremento e decremento das coordenadas da matriz. A Figura 3.1(a) apresenta um exemplo dessa classe de domínio.

No caso de domínios não estruturados, o número de pontos adjacentes a cada ponto vizinho pode não ser constante. A representação desse tipo de domínio deve ser feita através de grafos, por meio de matrizes de adjacências. A Figura 3.1(b) apresenta um domínio não estruturado com 10 pontos (GALANTE, 2006, Cap. 2).

Seja o domínio estruturado ou não estruturado, o volume de dados gerado pela discretização costuma ser quadrático, cúbico, ou pode assumir ordens ainda mais elevadas. Esse fator, de forma isolada, implica em um volume de dados e processamento que requer o emprego de arquiteturas paralelas de memória distribuída. O uso de arquiteturas paralelas torna necessária a partição dos dados do domínio entre os processos da aplicação. A Seção 3.2 seguinte apresenta o processo de partição de domínios estruturados, processo esse conhecido como de decomposição de domínios.

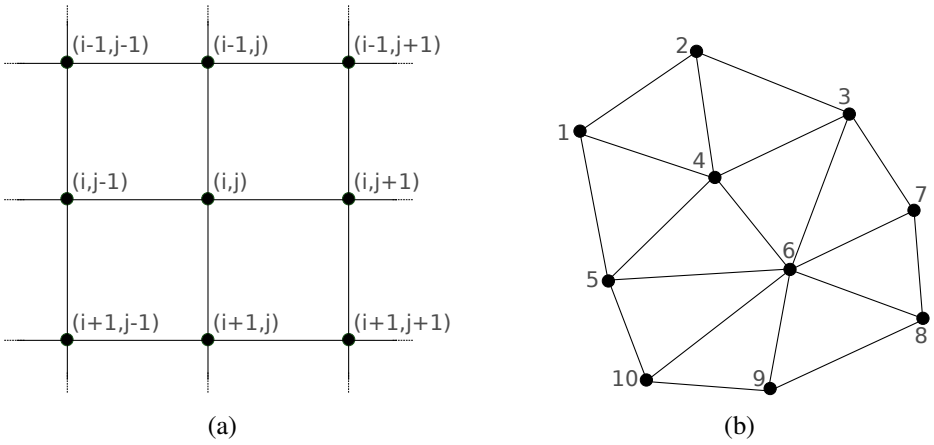


Figura 3.1: Domínio estruturado (a) e não estruturado (b)

3.2 Decomposição de domínios

De acordo com Smith, Bjørstad e Gropp (1996), na área de computação paralela, Decomposição de Domínios (DD) se refere ao processo de distribuição dos dados de um domínio entre os processadores de uma máquina de memória distribuída, a fim de possibilitar o processamento paralelo de Equações Diferenciais Parciais. Por meio de DD, um domínio é particionado em vários subdomínios, que são mapeados nos processadores da máquina paralela. Essa divisão de dados permite que cada processador faça uma varredura no domínio local aplicando uma determinada função sobre cada célula do subdomínio, ou resolva o problema local aplicando um conjunto de equações sobre os dados. A solução global é obtida por meio da combinação de cada resultado local (SAAD, 1994).

A Figura 3.2 apresenta a decomposição de um domínio estruturado bidimensional entre 9 processos dispostos em uma topologia de tamanho $\sqrt{P} \times \sqrt{P}$, sendo P o número total de processos. Cada processo recebe uma porção do domínio denominada *subdomínio*, cada qual de tamanho $m \times n$, onde $m = \lfloor \frac{M}{\sqrt{P}} \rfloor$ e $n = \lfloor \frac{N}{\sqrt{P}} \rfloor$.

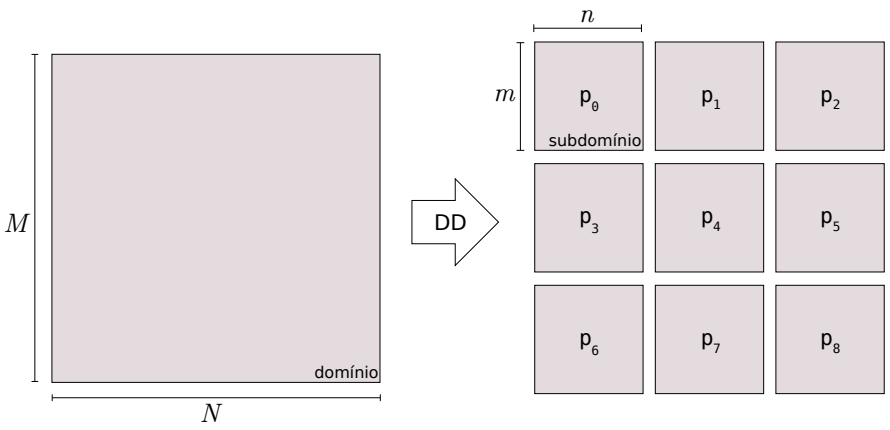


Figura 3.2: Decomposição de um domínio estruturado bidimensional de tamanho $M \times N$ entre 9 processos

De posse dos subdomínios, os processos varrem suas partições do domínio a fim de determinar o valor de cada ponto da partição por meio de um estêncil. A ideia de estêncil nasce a partir da discretização da EDP, que é discutida na Seção 3.3.

3.3 Background

Uma EDP está relacionada à uma derivada parcial definida sobre funções contínuas, cuja diferenciação ocorre em todas as dimensões do domínio. Como exemplo, a Fórmula (3.1) apresenta a derivada parcial da Equação do Calor, cuja EDP é baseada no operador Laplaciano (LEISERSON, 2010). Essa EDP modela a dispersão física de calor sobre um espaço bidimensional em função do tempo.

$$\underbrace{\frac{\partial u(t,x,y)}{\partial t}}_{\text{tempo}} = \alpha \underbrace{\left(\frac{\partial^2 u(t,x,y)}{\partial x^2} + \frac{\partial^2 u(t,x,y)}{\partial y^2} \right)}_{\text{espaço}}. \tag{3.1}$$

Para que as arquiteturas atuais sejam capazes de processar esse tipo de modelo físico, é necessário que as diferenciações temporal e espacial da EDP sejam traduzidas de sua forma contínua para uma representação discreta, possibilitando ao computador processar iterativamente cada célula do domínio. Um método bastante usual de discretização de EDPs em domínios estruturados é o Método de Diferenças Finitas (MDF) (RIZZI, 2002, Anexo 1), que aproxima as derivadas parciais do tempo e do espaço da Fórmula (3.1) nas seguintes Fórmulas (3.2) e (3.3):

$$\frac{\partial u(t,x,y)}{\partial t} \approx \frac{U^{n+1}_{x,y} - U^{n}_{x,y}}{\Delta t}, \tag{3.2}$$

$$\begin{aligned} \frac{\partial^2 u(t,x,y)}{\partial x^2} &\approx \frac{U^{n}_{x-1,y} + U^{n}_{x+1,y} - 2U^{n}_{x,y}}{\Delta x^2}, \\ \frac{\partial^2 u(t,x,y)}{\partial y^2} &\approx \frac{U^{n}_{x,y-1} + U^{n}_{x,y-1} - 2U^{n}_{x,y}}{\Delta y^2}. \end{aligned} \tag{3.3}$$

$$\begin{aligned} U^{n+1}_{x,y} = U^{n}_{x,y} &+ \frac{\alpha \Delta t}{\Delta x^2} \left(U^{n}_{x-1,y} + U^{n}_{x+1,y} - 2U^{n}_{x,y} \right) + ... \\ &... + \frac{\alpha \Delta t}{\Delta y^2} \left(U^{n}_{x,y-1} + U^{n}_{x,y+1} - 2U^{n}_{x,y} \right). \end{aligned} \tag{3.4}$$

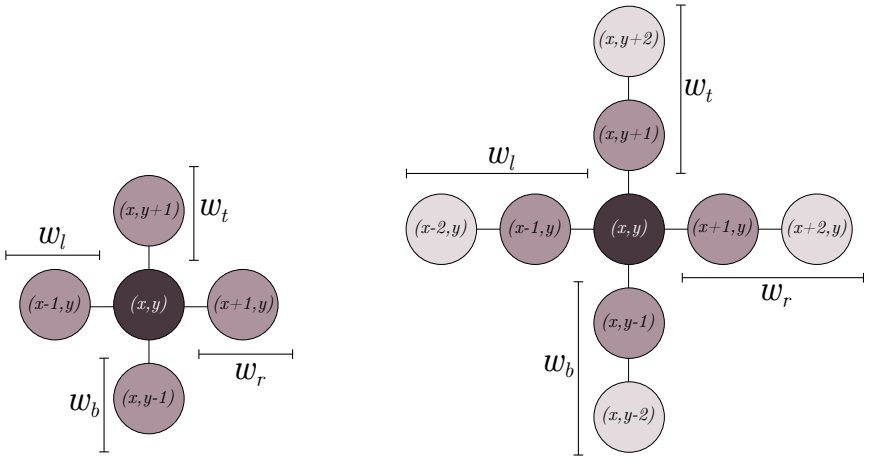
A relação entre as derivadas discretizadas resulta na Equação (3.4), que representa o cálculo de cada célula (x, y) na iteração $n + 1$. Nessa equação, tanto o aspecto temporal quanto o espacial da Fórmula (3.1) são contemplados. Pelo aspecto espacial, o valor de cada célula na posição (x, y) é determinado com base na soma dos valores das células vizinhas imediatas. Já pelo aspecto temporal, o valor contido na célula (x, y) de 1 iteração passada (iteração n) deve ser adicionado à soma do valor das células vizinhas da iteração atual. O aspecto espacial da Equação (3.4) tem um papel importante por introduzir a noção de *estêncil*, apresentada na Seção 3.4 a seguir.

3.4 Estêncil

A partir da Equação (3.4) é possível estabelecer o seguinte padrão de processamento para cada célula do domínio: o cálculo de uma célula arbitrária (x, y) depende sempre

do valor de um determinado subconjunto de células vizinhas relativas a (x, y), especificamente nas posições $(x+1, y)$, $(x-1, y)$, $(x, y+1)$ e $(x, y-1)$. Esse padrão de cálculo é conhecido como computação baseada em estêncil, onde um *kernel* é responsável por varrer toda a extensão do domínio e calcular o valor de cada posição. O estêncil é definido como um elemento de alto nível que especifica quais células vizinhas contribuirão para o cálculo dos pontos do domínio.

A Figura 3.3(a) ilustra o respectivo estêncil da Equação (3.4), onde os parâmetros w_t, w_b, w_l e w_r representam as larguras do estêncil, que correspondem à quantidade de pontos nas respectivas orientações. No contexto de EDPs, a largura do estêncil depende da ordem da derivada parcial e da precisão desejada para o resultado do sistema. Por exemplo, uma derivada parcial de mais alta ordem necessitaria de um estêncil mais largo, como o da Figura 3.3(b). De uma forma geral, o estêncil pode assumir diferentes formatos por estar intrinsecamente relacionado a uma determinada aplicação. Em (CHRISTEN; SCHENK; BURKHART, 2011) são apresentadas outras rotinas *kernel* de aplicações que possuem estênceis diferentes.



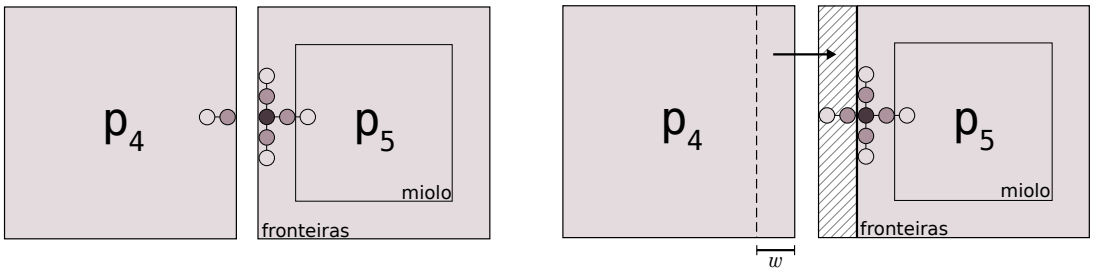
(a) Estêncil de 5 pontos (b) Estêncil de 9 pontos

Figura 3.3: Estênceis de largura $w = 1$ (a) e $w = 2$ (b). Cada ponto nas respectivas direções contribui para o valor do ponto central.

O formato do estêncil define a dependência de dados entre subdomínios de um domínio decomposto. Considerando subdomínios de tamanho $m \times n$ e um estêncil de largura w, onde $w = w_t = w_b = w_l = w_r$, a dependência de dados teria um tamanho de $m \times w$ com os subdomínios vizinhos à direita e à esquerda, e $w \times n$ com os subdomínios superiores e inferiores. Essa dependência de dados ocorre pela necessidade dos processos calcularem os pontos situados nas *fronteiras* do subdomínio.

As fronteiras são definidas como as w linhas e colunas mais externas da respectiva matriz que armazena os dados do subdomínio. A Figura 3.4(a) ilustra a dependência de dados de um processo p_5 com relação ao vizinho p_4, considerando o estêncil da Figura 3.3(b). Assim como as fronteiras são as bordas mais externas do subdomínio, o *miolo* é definido como o subdomínio completo desconsiderando-se suas fronteiras.

Como se considera que a execução ocorra em um ambiente paralelo de memória distribuída não compartilhada, a situação da Figura 3.4(a) é inverossímil, uma vez que p_5 está acessando diretamente os pontos no espaço de memória de outro processo para calcular a respectiva célula da fronteira. Portanto, é necessário que todos os processos mantenham



(a) (b)

Figura 3.4: Região de sobreposição. O processo p_5 mantém as duas últimas colunas do vizinho p_4 para calcular seus pontos de fronteira.

uma cópia das fronteiras de seus vizinhos em memória local, assim como ilustra a Figura 3.4(b), onde a área hachurada representa a fronteira direita de p_4 armazenada em p_5. Essa área é denominada *Região de Sobreposição*, pelo fato dos valores desse *buffer* serem idênticos aos valores da fronteira do processo vizinho.

3.5 Estado da arte

Esta seção estabelece uma ponte entre o presente capítulo e o Capítulo 2, que abordou a técnica de *auto-tuning* voltado para software sequencial. A Seção 3.5.2 apresenta trabalhos relacionados a *auto-tuning* de aplicações paralelas baseadas no processamento de domínios estruturados. Embora não esteja relacionada a *auto-tuning*, a Seção 3.5.1 apresenta a biblioteca científica paralela PETSc, que está relacionada, em específico, com decomposição de domínios estruturados e o gerenciamento da troca de dados entre processos.

3.5.1 Biblioteca PETSc

A PETSc (*Portable, Extensible Toolkit for Scientific Computation*) (BALAY et al., 2008) é uma biblioteca em linguagem C para o desenvolvimento de aplicações paralelas científicas. A biblioteca é dividida em várias camadas de abstração que, nas camadas mais altas e médias, envolvem *solvers* para sistemas lineares, não lineares e métodos para pré-condicionamento de sistemas. Nas camadas mais baixas, situam-se a representação de matrizes densas e esparsas, e o subsistema de comunicação entre processos paralelos da biblioteca, suportado pelas rotinas do padrão MPI.

Dentro da camada de abstração de matrizes densas, a PETSc suporta a representação e o gerenciamento de domínios estruturados distribuídos, denominados pela biblioteca de *Distributed Arrays* (DA). DAs são estruturas de dados especializadas que têm o papel de gerenciamento da comunicação entre os subdomínios, feita por meio das regiões de sobreposição.

A PETSc suporta dois tipos de estênceis, e os classifica como *box* e *star*. Os DAs adaptam as regiões de sobreposição dependendo do tipo do estêncil que a aplicação adota. As áreas sombreadas da Figura 3.5, extraída de (BALAY et al., 1997), ilustram os dois tipos de regiões de sobreposição para um processo 6 qualquer, onde cada quadrado corresponde a um processo. A Figura 3.6 ilustra os formatos de estêncil *box* e *star* suportados pela PETSc. Como o estêncil do tipo *box* referencia os pontos nas diagonais, a zona de sobreposição deve satisfazer também essa dependência de dados, a fim de possibilitar o

cálculo dos pontos situados nos cantos do subdomínio. O estêncil do tipo *star* define, também, que os subdomínios nas diagonais serão reconhecidos como vizinhos do processo.

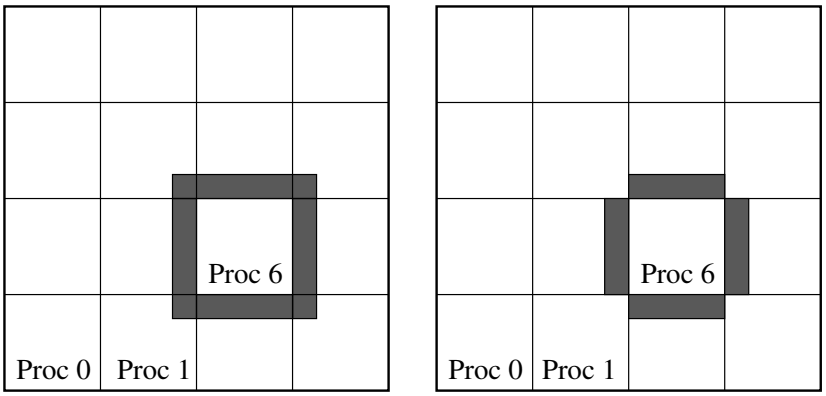


Figura 3.5: DAs dos tipos de estêncil *box* e *star*

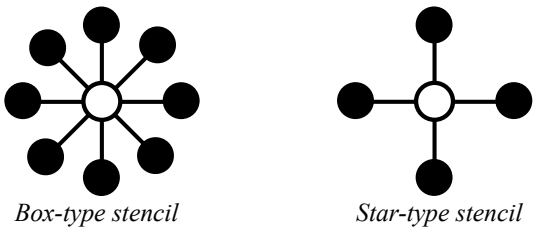


Figura 3.6: Estênceis suportados pelos DAs, dos tipos *box* e *star*

3.5.2 Aplicações paralelas

Embora as técnicas de *auto-tuning* tenham surgido no contexto de bibliotecas numéricas a fim de adaptá-las automaticamente em máquinas *single-core*, tem crescido o interesse em aplicar *auto-tuning* de software no contexto de aplicações que executam em máquinas paralelas (ASANOVIC et al., 2009). Essa seção apresenta trabalhos encontrados na literatura cujas técnicas de *auto-tuning* sejam orientadas para aplicações paralelas que se baseiam no processamento de domínios estruturados.

3.5.2.1 Decomposição de domínio em múltiplos níveis

O trabalho de Datta et al. (2008) apresenta um esquema de Decomposição de Domínios tridimensionais (3D) em 4 níveis, a fim de explorar a localidade dos dados em hierarquias de memória de baixa latência. O processo de *auto-tuning* tem a função de escolher o melhor tamanho para cada nível, de forma a explorar a arquitetura eficientemente. A aplicação utilizada no trabalho foi a Equação do Calor.

A Figura 3.7, extraída de (DATTA et al., 2008), ilustra a estratégia de decomposição em 4 níveis de um domínio 3D. O primeiro nível de decomposição é feito dividindo-se o domínio completo de tamanho $NX \times NY \times NZ$ em *core blocks*, de tamanho $CX \times CY \times CZ$ (Figura 3.7(a)). O segundo nível de decomposição, representado na Figura 3.7(b), é obtido dividindo-se cada *core block* em vários *thread blocks* de tamanho $TX \times TY \times CZ$. O terceiro nível de decomposição é alcançado através do particionamento das *thread blocks* em *register blocks* de tamanho $RX \times RY \times RZ$, representado pela Figura 3.7(c).

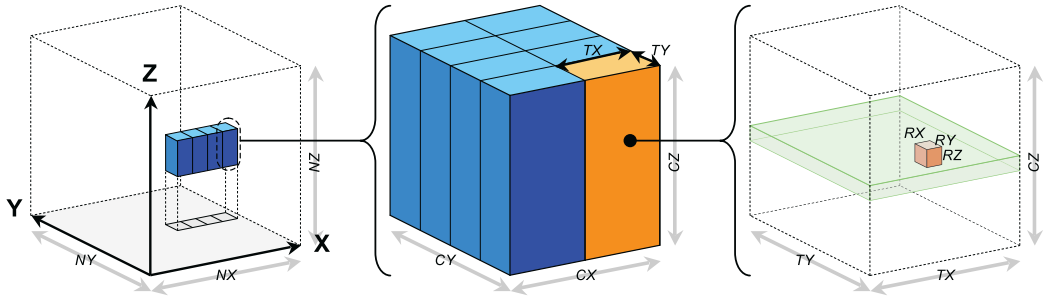


Figura 3.7: Decomposição de um domínio 3D em 4 níveis

O quarto nível de decomposição é o agrupamento dos *core blocks* em *chunks*. Os quatro *core blocks* da Figura 3.7(a) representam um *chunk*. A quantidade de *core blocks* que devem compor cada *chunk* é um fator que depende da arquitetura, sendo, portanto, um dos parâmetros adaptados pelo *auto-tuner*.

Dado esse esquema de decomposição, para cada *core* de uma arquitetura *multi-core* é mapeado um *chunk*. Os *core blocks* que compõem cada *chunk* são processados por um conjunto de *threads*, que efetuam o cálculo dos *thread blocks*. Dessa forma, a aplicação explora o paralelismo entre os *cores* da arquitetura, e ainda um possível paralelismo *multi-thread* em cada *core*.

O *auto-tuning* é feito através de um gerador de código escrito em linguagem Perl, que produz código *multi-thread* em linguagem C. Um segundo componente do *auto-tuner* explora o espaço de busca dos parâmetros a serem considerados.

O artigo demonstra a autoadaptação nas arquiteturas Intel Core2, AMD Barcelona, Sun Niagara2, STI Cell eDP SPE e NVIDIA GT200 SM. Com isso, o espaço de busca dos parâmetros é limitado de acordo com heurísticas baseadas nas características das arquiteturas consideradas, a fim de reduzir o tempo de *auto-tuning*. Além disso, o código *multi-thread* é dependente da arquitetura, sendo gerado ou via *Pthreads* (Core2, Barcelona e Niagara2), ou por meio da biblioteca *libspe* (Cell), ou ainda utilizando o modelo de programação CUDA 2.0 (NVIDIA GT200). O gerador é capaz, ainda, de sintetizar código SIMD SSE para as arquiteturas Intel Core2 e AMD Barcelona.

3.5.2.2 Autoadaptação de código anotado

O trabalho de Kamil et al (2009) apresenta uma abordagem semelhante ao de Datta et al (2008), adaptando automaticamente o domínio através dos 4 níveis ilustrados na Figura 3.7. Entretanto, enquanto Datta et al (2008) trabalha especificamente com a Equação do Calor, Kamil et al (2009) apresenta um *framework* que generaliza o *auto-tuning* para qualquer *kernel*.

Para efetuar a adaptação, o programador deve anotar o *kernel* no código, que será extraído pelo *framework* e autoadaptado na arquitetura. O *framework*, por sua vez, cria uma representação da rotina na forma de uma AST (*Abstract Syntax Tree*) (AHO; SETHI; ULLMAN, 1995, Capítulo 5), através da qual são feitas transformações no código a fim de otimizá-lo. As otimizações aplicadas na AST são referentes à adaptação em blocos de registradores (por meio do desenrolar de laços), em blocos de *cache* e propagação de constantes numéricas.

Após o *auto-tuner* aplicar as otimizações sobre a AST, o gerador de código produz um fonte em linguagem C, que é compilado e executado na arquitetura, anotando-se o

desempenho obtido. Esse ciclo é iterado para cada transformação aplicada na AST, ao final escolhendo o caso que apresenta melhor desempenho. O *auto-tuning* referente ao domínio da aplicação é feito da mesma maneira que o trabalho de Kamil et al (2009), apresentado na Seção 3.5.2.1 anterior.

3.5.2.3 Adaptação do número de regiões de sobreposição

Quando um domínio estruturado é decomposto em um ambiente paralelo de memória distribuída, cada processo deve manter as linhas e colunas de fronteira dos seus subdomínios vizinhos, a fim de possibilitar a determinação dos pontos das fronteiras locais. Assume-se, portanto, que cada subdomínio *sobrepõe-se* com seus subdomínios vizinhos em uma ou mais linhas e colunas. Na Figura 3.8(a), as linhas e colunas sombreadas denotam as Regiões de Sobreposição (BALAY et al., 2008, Seção 2.4) que o processo p_4 mantém sobre seus subdomínios vizinhos. Dessa forma, a cada iteração do algoritmo os processos devem receber as regiões de fronteira de seus respectivos vizinhos, a fim de atualizar suas regiões de sobreposição.

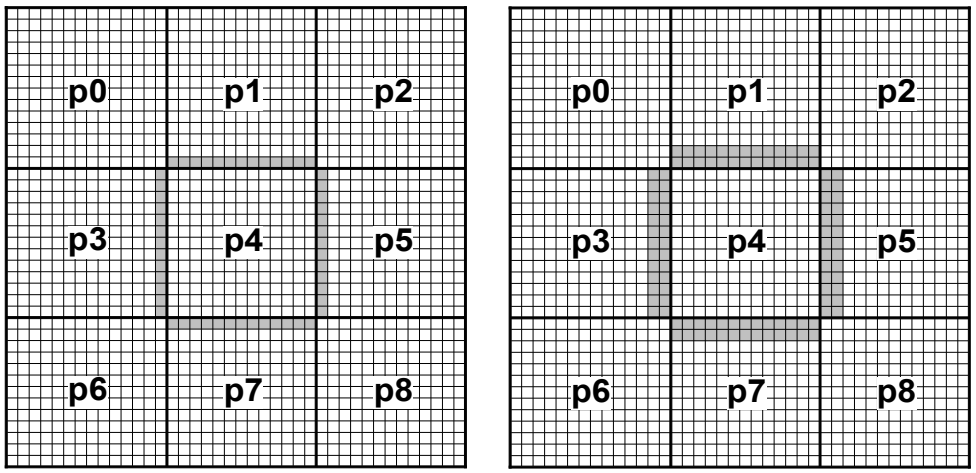


Figura 3.8: Regiões de sobreposição em um domínio 2D decomposto entre 9 processos

Aumentando-se o número de regiões de sobreposição (g), assim como ocorre na Figura 3.8(b), possibilita-se que os processos calculem os pontos de fronteira dos subdomínios vizinhos, evitando, dessa forma, o recebimento destes pela rede. Com isso, posterga-se a comunicação a cada g iterações do algoritmo, ao custo de um aumento do volume de dados a ser transferido. Além disso, uma sobrecarga de cálculo adicional é atribuída aos processos, que devem determinar pontos não pertencentes ao seu subdomínio. No exemplo da Figura 3.8(b), como há duas regiões de sobreposição, os processos receberão os dados de seus vizinhos a cada duas iterações.

O trabalho de Allen et al (2001) apresenta um estudo de *auto-tuning* em domínios decompostos entre os nós de uma máquina paralela de memória distribuída. Uma das funcionalidades do *auto-tuner* proposto pelos autores é a de buscar um número ideal de regiões de sobreposição, de tal forma a aumentar a eficiência de execução da aplicação. O artigo aplica o *auto-tuning* sobre um domínio 3D, explorando uma aplicação na área da relatividade numérica.

Os autores consideram a execução em um ambiente largamente distribuído (*grid*), onde as condições da rede de interconexão podem mudar bruscamente no decorrer do tempo. A autoadaptação do número de regiões de sobreposição é justificada pelo fato de

que, nesse tipo de ambiente paralelo, a latência de rede pode estar na ordem de dezenas a centenas de milissegundos. Além disso, como as condições da rede podem variar no decorrer da execução, a autoadaptação do número de regiões de sobreposição é feita de forma dinâmica.

A metodologia adotada para autoadaptar o número de regiões sobrepostas foi a seguinte: ao iniciar a execução da aplicação paralela, todos subdomínios dos processos se sobrepõem com seus vizinhos em uma linha/coluna (assim como na Figura 3.8(a)). Após uma certa quantidade de iterações[1] do algoritmo, o número de regiões é aumentado para 2, e a eficiência da execução é comparada com a configuração anterior. Caso o módulo determine que a eficiência foi melhorada, o número de regiões é aumentado em uma unidade, e o processo se repete. Caso seja observado que a eficiência foi reduzida, ou que ela ficou dentro de um certo intervalo com relação ao valor anterior (intervalo fixado em $\pm 20\%$), o número de regiões de sobreposição é decrementado em uma unidade, estabilizando neste valor por um certo número de iterações. Salienta-se que o *auto-tuner* é executado individualmente em cada processo, de forma a possibilitar que o número de regiões de um subdomínio possa assumir valores diferentes com relação aos outros subdomínios.

A codificação da aplicação foi feita sobre o *framework* Cactus (GOODALE et al., 2003), uma plataforma modular voltada para o suporte à execução de aplicações numéricas. A fim de possibilitar a execução paralela em um ambiente de *grid*, os autores fizeram a integração do *middleware* Globus (FOSTER, 2005) ao Cactus. Com isso, a aplicação foi paralelizada utilizando o padrão para troca de mensagens MPI, por meio da implementação MPICH-G2 provida pelo Globus. O *auto-tuner* foi acoplado ao módulo do Cactus responsável por tratar a decomposição de domínios estruturados, abstraindo, assim, o *auto-tuning* da camada da aplicação.

Relacionado ao trabalho de (ALLEN et al., 2001), o trabalho de (RIPEANU; IAMNITCHI; FOSTER, 2001) estima um número ótimo de regiões de sobreposição entre subdomínios, considerando ambientes de *clusters* de computadores e de *cluster* de *clusters*, interconectados por uma *grid*. A aplicação utilizada no trabalho simula a colisão entre dois buracos negros, realizando o processamento sobre um domínio 3D. Assim como em (ALLEN et al., 2001), o desenvolvimento da aplicação foi feito sobre o *framework* Cactus integrado ao Globus.

3.5.2.4 Regiões de sobreposição em GPGPUs

O trabalho de Meng e Skadron (2009) apresenta um modelo teórico de desempenho para aplicações que efetuam processamento sobre domínios estruturados em GPGPUs, por meio do ambiente de programação CUDA. Através da modelagem teórica, os autores buscam um número de regiões de sobreposição ótimo para as aplicações consideradas. Além disso, o trabalho propõe um *framework* que, usando anotações explícitas de código, automatiza a implementação de múltiplas regiões de sobreposição no programa, além de estimar automaticamente o melhor número de regiões para a aplicação.

Os autores apresentam o estudo aplicado sobre quatro algoritmos que utilizam domínios estruturados: *PathFinder* (MENG; SKADRON, 2009), que encontra um caminho de peso mínimo a partir da base de um domínio 2D até o seu topo, por meio de um algoritmo de programação dinâmica; *HotSpot* (HUANG et al., 2004), que resolve uma Equação Diferencial Ordinária sobre um domínio 2D de forma iterativa, a fim de estimar a temperatura de unidades funcionais de um processador, onde a temperatura de cada ponto do

[1] A frequência com que o *auto-tuner* interfere na execução para avaliar uma possível adaptação não foi especificada no artigo.

domínio é determinada com base na temperatura de pontos vizinhos; *Poisson* (GROPP; LUSK; SKJELLUM, 1999, Capítulo 4), que resolve a equação de Poisson, utilizando o operador de Laplace para estimar os pontos de um domínio 2D; e *Cell* (GARDNER, 1970), um autômato celular utilizado para a execução do Jogo da Vida sobre um domínio 3D, onde cada célula do domínio é marcada como viva ou morta, dependendo do número de células vivas na vizinhança.

A GPU utilizada foi uma GeForce GTX 280 (plataforma NVIDIA Tesla), cuja arquitetura é constituída por 30 *Streaming Multiprocessors* (SMs) que se comunicam pelo compartilhamento da memória central do dispositivo, sendo cada SM capaz de executar até 512 *threads*. No CUDA, um grupo de 512 *threads* constitui um *thread block*, e cada *thread block*, por sua vez, é alocado a um SM sem sofrer preempção durante o processamento. As *threads* executando em uma SM compartilham, ainda, uma memória local denominada PBSM (*per-block shared memory*), onde os dados do subdomínio de cada *thread* devem ser explicitamente carregados no início da execução.

No momento em que uma *thread block* termina sua execução, o resultado da computação é transferido da PBSM para a memória principal do dispositivo, por meio da qual as *thread blocks* atualizam suas regiões de sobreposição com os pontos calculados nos outros subdomínios. Portanto, na situação onde os subdomínios possuem apenas uma região de sobreposição, a cada iteração do algoritmo, as *thread blocks* devem sincronizar seus valores da região de sobreposição com os valores de fronteira dos subdomínios vizinhos, causando um elevado sobrecusto de acesso à memória. Aumentando-se o número de regiões de sobreposição, as *thread blocks* são capazes de recalcular os pontos vizinhos, evitando, assim, a sincronização por um determinado número de iterações.

A Figura 3.9, extraída do artigo, ilustra a forma como os autores tratam os subdomínios das aplicações. O artigo denomina os subdomínios como *tiles* e a região de sobreposição como *halo*, assim como ilustra a Figura 3.9(a). Uma *halo region* é a menor região de sobreposição que um subdomínio pode ter. Quando a largura de uma região de sobreposição é aumentada, essa região é denominada *ghost zone* sendo, em outras palavras, constituída de múltiplas *halo zones*, assim como demonstra a Figura 3.9(b).

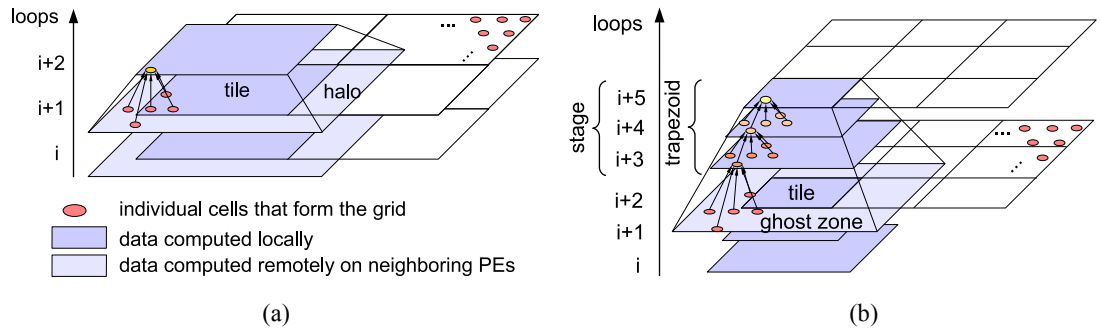


Figura 3.9: Subdomínios empilhados formando trapézios

Nessa última situação, os autores visualizam as múltiplas regiões de sobreposição empilhadas em estágios (*stages*), onde em cada estágio os pontos do subdomínio são determinados, assim como pontos dentro da região de sobreposição, sem haver troca de dados entre os Elementos Processadores (PEs) (*i.e.*, os SMs da GPU). Dessa forma, cada estágio possui uma região de sobreposição a menos a medida que se aproxima do topo da pilha, formando a figura de um trapézio (Figura 3.9(b)). A altura do trapézio reflete, portanto, o número de regiões de sobreposição que o subdomínio possui, e a quantidade

de iterações que os pontos do subdomínio podem ser determinados sem necessidade de sincronização entre os PEs.

A altura ideal do trapézio é estimada por um *auto-tuner*, que a determina por meio da modelagem de desempenho da GPU apresentada no artigo. Além disso, o *auto-tuner* é interno a um *framework* que, através de anotações no código-fonte, automatiza a implementação das regiões de sobreposição no programa. A Figura 3.10 ilustra uma simplificação da aplicação *HotSpot* com as anotações de código (em destaque) propostas pelos autores.

```
float **A, **B;
for k = 0 : num_iterations with trapezoid.height = [H]
    for_all i = 0 : ROWS-1 and j = 0 : COLS-1
        /* define o dominio da aplicacao */
        apply trapezoid.obj = [B]
        /* define o numero de dimensoes do dominio */
        apply trapezoid.dimension = 2
        /* define a largura do 'halo' em todas as dimensoes */
        apply trapezoid.gather[-1, +1][-1, +1]
        top = max(i - 1, 0);
        bottom = min(i + 1, ROWS-1);
        left = max(j - 1, 0);
        right = min(j + 1, COLS-1);
        A[ i ][ j ] = B[ i ][ j ] + B[ top ][ j ] + B[ bottom ][j] \
                                  + B[ i ][ left ] + B[ i ][ right ];
    swap(A, B);
```

Figura 3.10: Exemplo de código anotado que sofrerá autoadaptação

No código, o programador especifica o *array* que conterá o domínio da aplicação (matriz B, no exemplo), o número de dimensões (2 dimensões), e a largura da região *halo* do subdomínio (1 região *halo* em cada direção). De posse dessas informações, o *framework* transforma o código anotado em um código-fonte com as regiões de sobreposição implementadas. Por fim, o *framework* realiza uma estimativa de intensidade de computação utilizando a ferramenta *CUDA Profiler*, cujo resultado é utilizado como entrada no modelo teórico desenvolvido, a fim de estimar uma altura ótima para o trapézio (parâmetro `H`, no código).

3.6 Considerações do capítulo

Este capítulo definiu a base de decomposição de domínios, assim como a origem do estêncil, que define um padrão de processamento para o cálculo de cada célula do domínio. O capítulo apresentou também a biblioteca PETSc, que lida com a decomposição de domínios e o estêncil para o processamento paralelo de aplicações numéricas. Além disso, o estado da arte mostra uma gama de trabalhos que aplicam *auto-tuning*, principalmente sobre aplicações paralelas orientadas a arquiteturas de memória compartilhada e GPGPUs.

Este trabalho intersecciona-se com o de Ripeanu, Iamnitchi e Foster (2001) no que tange a variação da extensão das regiões sobrepostas. Entretanto, o trabalho vai além, propondo uma estrutura de dados orientada a objetos que abstraia a complexidade da extensão das regiões de sobreposição entre subdomínios, de forma a possibilitar o uso de *auto-tuning* em uma máquina de memória distribuída. Assim como a extensão das regiões de sobreposição, todo o gerenciamento da decomposição do domínio é abstraído

da aplicação por meio de Orientação a Objetos. Ressalta-se que não foi encontrada na literatura uma biblioteca que efetivamente abstraia o gerenciamento da decomposição de domínios de uma aplicação paralela.

Embora a documentação da PETSc comente sobre a adoção de Orientação a Objetos pela biblioteca, as estruturas de dados em si foram definidas através de `structs` da linguagem C, que não podem conter procedimentos que efetuam ações sobre elementos da própria estrutura. Portanto, a aplicação ainda deve lidar com chamadas de funções externas para operarem sobre as estruturas de dados da biblioteca, o que tende a tornar o código-fonte da aplicação prolixo e poluído.

A orientação a objetos da linguagem C++ é rica o suficiente para permitir uma abstração em um nível tão alto quanto o da biblioteca PETSc, tendo condições de manter os mesmos níveis de desempenho da aplicação. Assim, o Capítulo 4 a seguir apresenta uma abstração de domínios estruturados bidimensionais, desenvolvida sobre os conceitos de abstração de dados da linguagem C++. A classe possibilita estender as regiões de sobreposição entre os subdomínios vizinhos, permitindo que o *auto-tuner* tenha condições de adaptar a aplicação paralela na arquitetura.

4 CLASSE MESH E AUTO-TUNING

Assim como apresenta a Seção 3.4, uma aplicação que trabalha com decomposição de domínios deve tratar a atualização das regiões sobrepostas entre os subdomínios. Basicamente, essa atualização ocorre pelo envio e recebimento dos valores de borda entre os processos, que recebem os dados dos vizinhos e os armazenam nos respectivos *buffers* das regiões de sobreposição.

Além do envio e recebimento, a atualização das regiões de sobreposição pode acontecer também por recálculo. Através dessa técnica de atualização, cada processo atualiza suas regiões calculando localmente os pontos que seriam transferidos. O cálculo das regiões de sobreposição é possível quando se aumenta a extensão das regiões sobrepostas entre os subdomínios. Com isso, evita-se a atualização por meio de envio e recebimento por um determinado número de iterações, ao custo de um aumento do volume de cálculos em cada processo. A extensão das regiões sobrepostas é o fator da decomposição de domínios que sofrerá *auto-tuning*.

Com o intuito de abstrair da aplicação a complexidade do gerenciamento da atualização das regiões de sobreposição, ora feita por envio e recebimento, ora por recálculo, foi desenvolvida a classe `Mesh` em linguagem C++. Além do controle da atualização dos dados entre subdomínios, a classe abstrai todo o gerenciamento da decomposição de uma malha bidimensional entre os processos da aplicação paralela.

A Seção 4.1 apresenta uma visão em alto nível das classes desenvolvidas; a Seção 4.2 descreve a atualização das regiões de sobreposição por recálculo e por envio e recebimento; a Seção 4.3 apresenta o gerenciamento da topologia de processos, que é utilizado como base para a decomposição do domínio feita pela classe `Mesh`, apresentada na Seção 4.4. A generalidade da classe `Mesh` para diferentes formatos de estênceis é discutida na Seção 4.5. O *auto-tuner* desenvolvido é apresentado na Seção 4.6. Por fim, as considerações do capítulo são apresentadas na Seção 4.7.

4.1 Visão geral

A abstração desenvolvida é composta pelas seguintes classes, que gerenciam a comunicação entre processos e a manipulação dos dados do domínio estruturado:

- `Mesh`: classe de mais alto nível que gerencia a decomposição de domínios;
- `Topology`: abstrai a topologia bidimensional de processos do padrão MPI;
- `Ghost_Zones`: agrupa as regiões de sobreposição do subdomínio, abstraídas pelas classes `Halos`;

- `Halo_Left`, `Halo_Right`, `Halo_Top`, `Halo_Bottom`: gerenciam a atualização das regiões de sobreposição entre os subdomínios de cada direção, onde a atualização pode ocorrer tanto pela rede quanto pelo recálculo local dos pontos vizinhos.

A dependência entre as classes é apresentada pela camada intermediária da Figura 4.1. A aplicação paralela localiza-se na camada mais alta e interage apenas com a classe `Mesh`, enquanto que a biblioteca MPI mantém-se na camada inferior, provendo suporte ao paralelismo para as classes desenvolvidas.

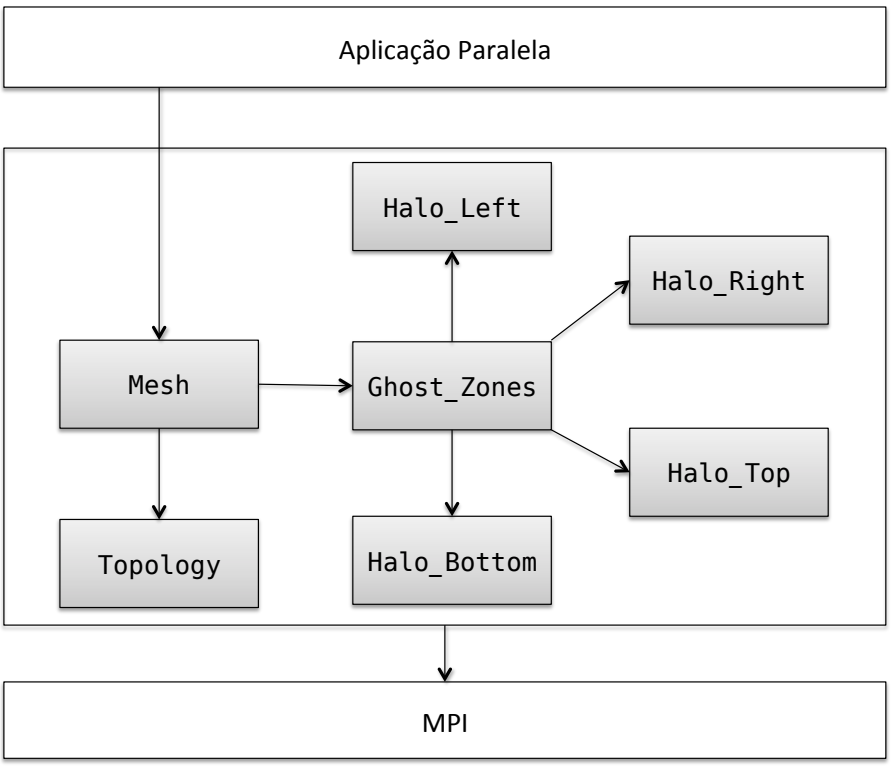


Figura 4.1: Diagrama de classes

As classes `Halos` são especialmente importantes pois elas são responsáveis pelo gerenciamento das regiões de sobreposição entre os subdomínios vizinhos, que é detalhado na Seção 4.2 a seguir.

4.2 Gerenciamento das regiões de sobreposição

A atualização das regiões de sobreposição é feita pelas classes `Halos`, que irão decidir por atualizar as regiões ou por recálculo dos pontos ou pelo recebimento através da rede de comunicação. O procedimento de recálculo dos pontos das regiões é apresentado na Seção 4.2.1. O acesso aos pontos das regiões de sobreposição é feito por meio de iteradores da linguagem C++, que são vistos como um meio de abstrair a varredura dos pontos da região com o estêncil. Os iteradores são expostos na Seção 4.2.2.

4.2.1 Atualização por recálculo

O recálculo dos pontos da região de sobreposição é possível quando os processos possuem uma parcela suficientemente grande da borda do subdomínio vizinho, de tal forma a ser possível aplicar o estêncil sobre os pontos da região. Para que um processo tenha condições de calcular os pontos situados na fronteira do subdomínio, é necessário que ele possua uma região de sobreposição que estenda-se no mínimo w linhas/colunas sobre o subdomínio vizinho, assim como demonstra a Figura 4.2(a). Nesse caso, p_5 acessa w pontos da região para calcular os pontos localizados na fronteira.

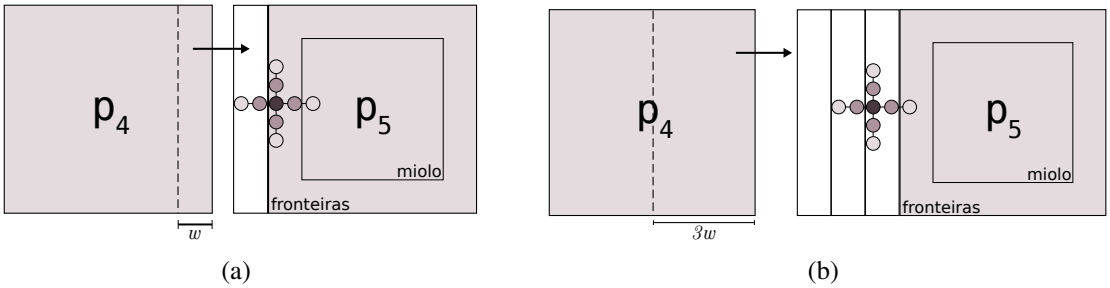


Figura 4.2: Aumento da extensão das regiões de sobreposição

Da mesma forma, para que a região sobreposta possa ser atualizada por recálculo, basta que as regiões tenham uma extensão de $s \cdot w$ linhas/colunas sobre os processos vizinhos onde $s > 1$, assim como demonstra a Figura 4.2(b). Nesse exemplo, o processo p_5 possui pontos vizinhos suficientes para aplicar o estêncil sobre os pontos da região. Como esses mesmos pontos são calculados no processo vizinho p_4, entende-se que atualização seja feita por *recálculo*.

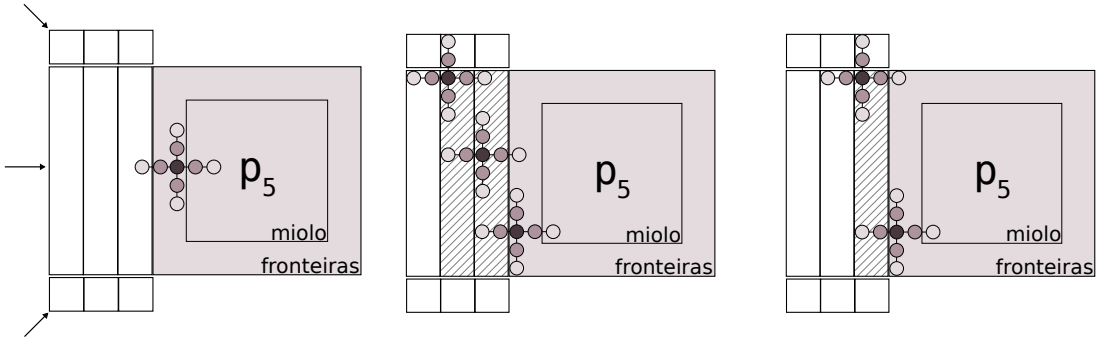
A atualização por recálculo possibilita que o processo permaneça $s - 1$ iterações sem trocar as regiões de sobreposição pela rede. Considerando o *layout* da Figura 4.2(b), onde $s = 3$, p_5 seria capaz de atualizar as regiões de sobreposição por recálculo por até 2 iterações.

Na primeira iteração, o processo receberia todas as regiões de sobreposição dos processos vizinhos, inclusive dos processos situados nas diagonais, assim como apresenta a Figura 4.3(a). Na segunda iteração, a troca de dados pela rede seria substituída pelo processamento dos pontos das duas regiões de sobreposição hachuradas na Figura 4.3(b). Na terceira iteração, a atualização por recálculo seria feita apenas na última região de sobreposição (Figura 4.3(c)). Como a atualização das regiões por recálculo é saturada na terceira iteração, na quarta iteração todas as regiões seriam atualizadas pela rede e o ciclo seria reiniciado.

A extensão das regiões de sobreposição é estática nas classes `Halos`. Ou seja, a aplicação inicia sua execução com uma extensão s, que permanece fixa durante a execução do programa. Além disso, o valor de s é igual entre todas as direções.

O recebimento das s linhas/colunas pela rede é feito pelas classes `Halos` através da primitiva `MPI_Irecv`, que inicia um recebimento não bloqueante da região com o respectivo processo vizinho. No caso da atualização por recálculo, o processamento das regiões de sobreposição é feito por *threads* POSIX, fato que torna essa forma de atualização também não bloqueante. Com isso, a aplicação é capaz de sobrepor a atualização das regiões com o processamento do miolo do subdomínio.

O acesso aos pontos da região de sobreposição é feito por meio de iteradores da linguagem C++. O uso desse tipo de estrutura de controle permite abstrair o acesso aos



(a) (b) (c)

Figura 4.3: Representação do ciclo de atualização das regiões de sobreposição pela rede (a) e por recálculo (b,c)

pontos e aplicar o estêncil sobre a região através de uma classe especializada.

4.2.2 Iteradores das regiões de sobreposição

Um iterador em C++ é definido como uma abstração de um ponteiro sobre os elementos de uma determinada sequência. Iteradores devem suportar três operações básicas: obter o valor do elemento sendo apontado (*dereferencing*), avançar para o próximo elemento da lista, e permitir a operação de igualdade entre os elementos. Qualquer estrutura de controle que implemente tais operações sobre uma lista pode ser considerada um iterador (STROUSTRUP, 1997, Cap. 19).

Internamente, as classes `Halos` utilizam iteradores para acessar e calcular os pontos das regiões de sobreposição. Cada iterador é relacionado a uma linha/coluna das regiões sobrepostas. Como ilustração, considerando que a região de sobreposição da Figura 4.2(b) tenha um tamanho de 600×6, a Figura 4.4 representa o *buffer* contíguo de armazenamento da região, e a Tabela 4.1 representa os iteradores sobre o *buffer* com os índices iniciais e finais associados.

0 1 2	...	99
100	...	199
200	...	299
300	...	399
400	...	499
500	...	599

Figura 4.4: *Buffer* contíguo de uma região de sobreposição de tamanho 600×6

Iterador	first()	last()
i_0	0	99
i_1	100	199
i_2	200	299
i_3	300	399
i_4	400	499
i_5	500	599

Tabela 4.1: Iteradores sobre o *buffer* da Figura 4.4

Os iteradores são definidos por meio da classe `halo_iterator` apresentada pelo Código 4.1, que especifica as operações básicas de incremento, comparação e acesso ao valor dos pontos. Além disso, na própria definição da classe iteradora é injetada a noção do estêncil por meio do método `reduce`. O método `reduce` recupera do *buffer* todos os valores referenciados pelo estêncil e calcula o valor do ponto. No caso do Código 4.1, o padrão de acesso segue o formato do estêncil de 9 pontos da Figura 3.3(b).

```
class halo_iterator : public iterator<forward_iterator_tag, struct neighbor> {
    private:
        double *curr; // Valor atual sendo apontado
        int stride;   // Deslocamento para acesso aos pontos norte/sul
                      // do estencil no buffer contiguo
    public:
        halo_iterator(double *curr, int stride) : curr(curr), stride(stride) {}
        // Operador de pre-incremento
        halo_iterator & operator++() {
            ++curr;
            return *this;
        }
        // Operador de pos-incremento
        halo_iterator & operator++(int) {
            curr++;
            return *this;
        }
        // Operador de comparacao ==
        bool operator==(const halo_iterator& rhs) {
            return curr == rhs.curr;
        }
        // Operador de comparacao !=
        bool operator!=(const halo_iterator& rhs) {
            return curr != rhs.curr;
        }
        // Operador para "dereferencing"
        double& operator*() {
            return *curr;
        }

        // Reducao dos valores do estencil
        // Este metodo depende do formato do estencil
        double reduce(void) {
            return (*(curr - 2) + *(curr - 1) +
                    *(curr + 1) + *(curr + 2) +
                    *(curr - stride) + *(curr - 2*stride)
                    *(curr + stride) + *(curr + 2*stride)) * 0.25;
        }
};
```

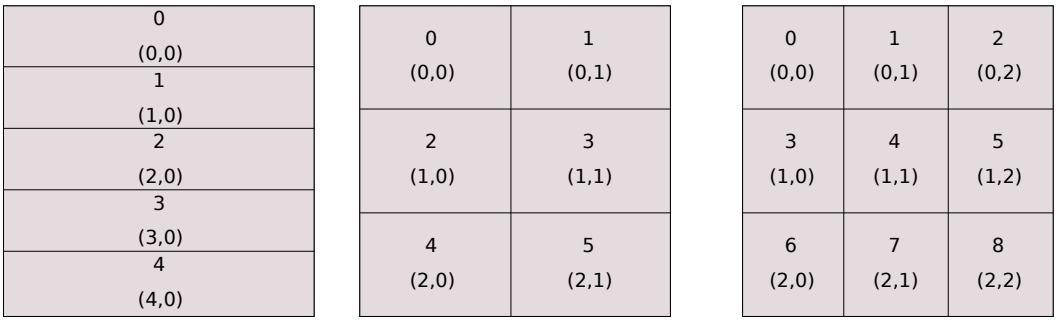
Código 4.1: Definição da classe de iteradores

4.3 Abstração da topologia de processos

A topologia virtual do padrão MPI organiza logicamente os processos de um determinado comunicador em uma distribuição em grade ou em um grafo de processos. No caso da distribuição em grade, cada processo recebe uma coordenada cartesiana que identifica sua posição dentro da estrutura.

A disposição dos processos na topologia é exemplificada pela Figura 4.5, que demonstra as topologias formadas por uma execução com 5, 6 e 9 processos. O MPI fatora o número de processos e dispõe o maior fator obtido no número de linhas da topologia (FORUM, 2009, Seção 7.2). Com isso, caso a raiz quadrada do número de processos não seja exata, cada processo terá um subdomínio com um número maior de colunas do que linhas.

A classe `Topology` (Código 4.2) é responsável por abstrair as operações de criação da topologia de processos, determinar as coordenadas cartesianas do processo atual e fazer o reconhecimento dos processos vizinhos.



(a) (b) (c)

Figura 4.5: Topologias cartesianas com 5 (a), 6 (b) e 9 processos (c)

```
class Topology {
 private:
    struct coords coords;  // Coordenadas x,y do processo na topologia
    int *periods, ndims;   // Numero de dimensoes da topologia
    struct dims dims;      // Numero de processos em cada dimensao
    MPI_Comm comm;         // Comunicador criado para a topologia
    int rank;              // Rank do processo na topologia
    int nprocs;            // Numero de processos na topologia
    int num_neighbors;     // Numero de vizinhos do processo
    struct neighbor *neighbors;   // Vizinhos do processo
    int init(MPI_Comm comm_old, int ndims, int *periods, int reorder);
    int meet_the_neighbors(void); // Reconhece a vizinhanca
 public:
    Topology(int ndims, int periods[] = NULL,
             int reorder = 0, MPI_Comm comm_old = MPI_COMM_WORLD);
    MPI_Comm get_comm(void); // Retorna o comunicador da topologia
    int get_rank(void);   // Retorna o rank do processo
    int get_nprocs(void); // Retorna o numero de procs. que compoe a topologia
    int get_ndims(void);  // Retorna o numero de dimensoes da topologia
    struct dims get_dims(void);  // Retorna o numero de procs. em cada dimensao
    struct coords get_coords(void); // Coordenadas do proc. na topologia
};
```

Código 4.2: Classe que abstrai a topologia de processos

4.4 Classe Mesh

A classe de mais alto nível é a classe `Mesh`[1], que tem a responsabilidade em abstrair a decomposição de domínios da aplicação. Ao instanciar um objeto `Mesh`, a aplicação deve informar dois parâmetros M e N, que correspondem ao número de linhas e colunas do domínio a ser decomposto. A classe toma como base a topologia criada por `Topology` para calcular os intervalos de linhas e colunas pertencentes a cada processo na topologia. Com isso, a classe alcança a decomposição discutida na Seção 3.2, dividindo o domínio completo em subdomínios menores de tamanho $m \times n$. O Código 4.3 apresenta a definição da classe desenvolvida.

A aplicação paralela pode invocar a atualização das regiões de sobreposição através do método `send_borders`, que irá iniciar o envio não bloqueante das bordas do subdomínio para os processos vizinhos. O método `sync` é responsável por sincronizar o fim da comunicação com a aplicação paralela. A remontagem dos subdomínios formando o domínio completo é possível através do método `gather`, que reune todos os subdomínios no processo com identificador 0 (zero).

[1] Código-fonte disponível em https://github.com/lxalmeida/Mesh2D

```
class Mesh {
    private:
        double **data_u, **data_v;
        double **buff_left, **buff_right;
        Ghost_Zones *ghost_zones;
        struct range_t range;
        bool transfer;
        MPI_Request sreq_left, sreq_right, sreq_top, sreq_bottom;
        MPI_Request sreq_top_left,
                    sreq_top_right,
                    sreq_bottom_left,
                    sreq_bottom_right;
        MPI_Datatype type_corner;
        void alloc_mesh(void);
        void free_mesh(void);
        void set_interval(int size, int rank, int nprocs, int *start, int *end);
        int num_threads;
    public:
        Mesh(int m, int n);
        Mesh(int m, int n, int num_ghost_zones, int numthreads = 1);
        void swap(void);
        Topology *topology;
        double **data_source, **data_dest;
        double **final_result;
        void send_borders(void);
        void sync(void);
        double get_source_halo(int row, int col);
        void gather(void);
};
```

Código 4.3: Classe Mesh

4.5 Generalidade da classe Mesh

Tendo o papel de uma biblioteca paralela para decomposição de domínios, a classe `Mesh` deve conhecer a dependência de dados entre processos, a vizinhança entre os subdomínios, além do *kernel* numérico envolvido no cálculo da região de sobreposição. Essas informações são fortemente dependentes do estêncil da aplicação, e podem ser inferidas a partir de seu formato.

A Figura 4.6 apresenta quatro exemplos de estênceis com diferentes geometrias. Pelo estêncil da figura (a), cada processo deveria comunicar-se com 2 vizinhos horizontais e 2 vizinhos verticais em cada direção. O estêncil da figura (b) segue o mesmo reconhecimento de processos vizinhos, variando apenas a dependência de dados entre os vizinhos verticais em relação à figura (a).

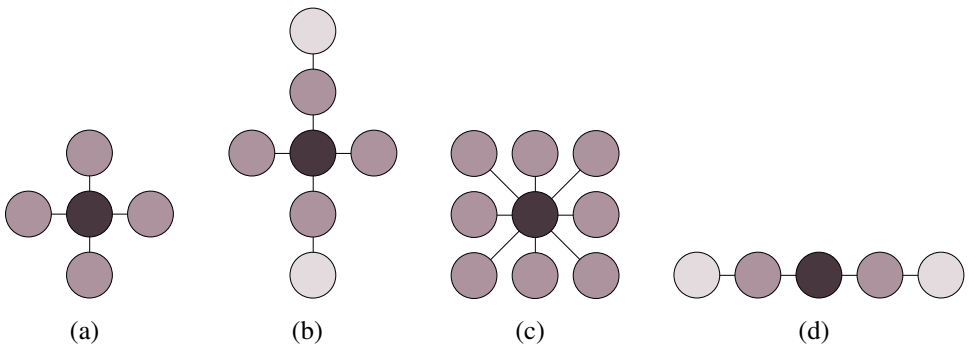


Figura 4.6: Exemplos de estênceis com diferentes geometrias

Os estênceis (c) e (d) expressam um reconhecimento diferenciado de subdomínios

```
for(int i = 0; i < M; i++)
    for(int j = 0; j < N; j++)
        u[i][j] = (v[i-2][j] + v[i-1][j] + v[i+1][j] + v[i+2][j] +
                   v[i][j-1] + v[i][j+1]) * constante;
```

Código 4.4: *Kernel* para o estêncil da Figura 4.6(b)

vizinhos em relação aos estênceis (a) e (b). No caso do estêncil (c), cada subdomínio estaria sobreposto com os vizinhos localizados nas diagonais, além dos vizinhos horizontais e verticais. Já no caso do estêncil (d), como o cálculo do ponto central não depende de pontos na dimensão vertical, cada processo reconheceria apenas os vizinhos horizontais na topologia de processos.

Além da dependência de dados e da vizinhança de processos, o formato do estêncil possibilita a criação da respectiva rotina *kernel*. O Código 4.4 apresenta um *kernel* que seria extraído a partir do estêncil da Figura 4.6(b), onde cada posição de v referencia um ponto específico do estêncil.

Embora a classe `Mesh` como um todo seja estática para um determinado formato de estêncil, a partir de uma descrição da geometria do estêncil seria possível adaptar automaticamente a classe para o novo formato. Com isso, a descrição do estêncil da aplicação poderia alimentar uma ferramenta geradora de código que sintetizaria as partes específicas da classe `Mesh`.

Especificamente, as porções de código que deveriam ser adaptadas seriam os métodos de envio e recebimento das regiões de sobreposição, o reconhecimento da vizinhança através da classe `Topology` e a classe que define os iteradores das regiões de sobreposição (Código 4.1), onde o *kernel* deveria ser injetado no método `reduce` para o cálculo dos pontos das regiões.

4.6 Auto-tuner

Assumindo o número de regiões de sobreposição como um parâmetro da aplicação, o *auto-tuner* foi desenvolvido em linguagem Ruby sobre o princípio de Adaptação Parametrizada (Seção 2.2.1), e é orientado a aplicações paralelas escritas sobre o padrão MPI.

O *auto-tuner* trabalha coletando amostras de tempo resultantes de sucessivas execuções da aplicação paralela na arquitetura, de forma a alimentar um arquivo XML que contenha cada amostra de tempo relacionada com os respectivos valores dos parâmetros utilizados. Uma vez que a base de amostras tenha sido montada, a aplicação pode ser executada por intermédio do *auto-tuner* que, para o tamanho de entrada, consulta a combinação de valores dos parâmetros que apresenta o menor tempo de processamento, e executa a aplicação com a combinação obtida.

Os parâmetros variados pelo *auto-tuner* são denominados *fatores*, e a faixa de valores a ser variada é denominada *nível* do fator. Os fatores a serem explorados pelo *auto-tuner* correspondem ao número de regiões de sobreposição da decomposição de domínios, ao número de processos da aplicação paralela e ao número de linhas e colunas do domínio. Os níveis a serem explorados em cada fator devem ser especificados pelo usuário na chamada ao *auto-tuner*.

O *auto-tuner* foi denominado mpiexec[2], homônimo ao lançador de aplicações encon-

[2]Código-fonte disponível em https://github.com/lxalmeida/mpiexec.rb

trado nas implementações do padrão MPI como MPICH2 (Argone National Laboratory, 2011) e Open MPI (The Open MPI Project, 2011). Antes da aplicação ser executada por intermédio do *auto-tuner*, a base de amostras deve ter sido previamente construída. A construção da base de amostras é realizada informando ao mpiexec os intervalos de valores que deverão ser variados em cada fator da aplicação. A combinação dos intervalos de valores possíveis de cada parâmetro irá gerar um espaço de busca com todas as combinações possíveis de cenários de execução. Este espaço de busca será, posteriormente, explorado pelo *auto-tuner*. O comando de construção da base de amostras segue a seguinte estrutura:

```
./mpiexec.rb <parâmetros mpiexec> ./executavelParalelo <parâmetros aplicação>
```

Os fatores devem ser especificados tanto em `<parâmetros mpiexec>` quanto em `<parâmetros aplicação>`. Em `<parâmetros mpiexec>` devem ser especificados todos os parâmetros do *mpiexec* da distribuição MPI necessários para lançar a aplicação paralela, enquanto que em `<parâmetros aplicação>` devem ser especificados os fatores específicos da aplicação. Como o número de processos é um parâmetro da aplicação controlado pelo MPI, ele deve ser especificado em `<parâmetros mpiexec>` através da letra p, seguida de uma lista de números de processos entre colchetes, onde cada número deve ser separado por uma vírgula. Por exemplo, a seguinte chamada especifica que o *auto-tuner* deve executar a aplicação com 1, 2, 4 e 6 processos:

```
./mpiexec.rb -n p[1,2,4,6] ./executavelParalelo <parâmetros aplicação>
```

O tamanho da entrada e o número de regiões de sobreposição da decomposição de domínios são fatores de `<parâmetros aplicação>`. O tamanho da entrada é especificado pelo número de linhas e colunas do domínio por meio das letras r e c, respectivamente, seguidas de um intervalo e um incremento. Da mesma forma ocorre para o número de regiões de sobreposição, que é identificado pela sequência de letras ngz. Para cada fator r, c e ngz, o *auto-tuner* cria os cenários de execução expandindo os intervalos especificados, do limite inferior até o limite superior, seguindo o valor de incremento. A seguinte chamada informa ao *auto-tuner* a criação de cenários para domínios de tamanho que variam de 100 a 1000 ao passo de 100, e a variação do número de regiões de sobreposição de 2 a 10 ao passo de 2:

```
./mpiexec.rb -n p[1,2,4,6] ./executavelParalelo r[100,1000,100] c[100,1000,100] \
                                                                    ngz[2,10,2]
```

O espaço de busca conteria cada combinação possível de valores nos parâmetros. Como os níveis de cada fator devem ser combinados, o número de cenários no espaço de busca tende a assumir um crescimento exponencial. Por exemplo, a chamada anterior ao *auto-tuner* geraria 2000 combinações, e os seguintes cenários de execução estariam contidos no respectivo espaço de busca:

```
mpiexec -n 6 ./executavelParalelo 100 100 2
mpiexec -n 6 ./executavelParalelo 100 100 4
mpiexec -n 6 ./executavelParalelo 100 100 6
mpiexec -n 6 ./executavelParalelo 200 200 2
mpiexec -n 4 ./executavelParalelo 200 200 2
mpiexec -n 2 ./executavelParalelo 200 200 2
...
```

De posse do espaço de busca, cabe ao *auto-tuner* escolher as linhas de comando que efetivamente serão testadas na arquitetura, de forma a tornar o tempo de *auto-tuning* viável.

4.6.1 Exploração do espaço de busca

O *auto-tuner* restringe o espaço de busca para domínios quadrados, onde o número de linhas e colunas sejam iguais. A obtenção das amostras de tempo é feita pela execução de cada uma das linhas de comando do espaço de busca na arquitetura segundo um algoritmo, que interrompe a execução para um determinado tamanho de entrada a partir do momento que o *auto-tuner* verificar perda de desempenho.

A coleta das amostras de tempo é feita iniciando a execução dos cenários a partir do maior tamanho de entrada e do maior número de processos. Como se busca o impacto do aumento da extensão das regiões de sobreposição, o *auto-tuner* executa todos os cenários de execução que variam as regiões de sobreposição para cada número de processos. Por exemplo, considerando a seguinte chamada ao *auto-tuner*:

```
./mpiexec.rb -n p[1,2,4,6,8,9] ./executavelParalelo r[100,1000,100] c[100,1000,100] \
                                                                        ngz[2,10,2]
```

a obtenção das amostras de tempo seria iniciada a partir de domínios de tamanho 1000×1000 com 9 processos. Para esse tamanho de entrada e número de processos, a aplicação seria executada com 2, 4, 6, 8 e 10 regiões de sobreposição. Ao terminar a execução com 9 processos, o *auto-tuner* testaria a aplicação com 8 e 6 processos. Caso seja observado ganho de desempenho, as amostras de tempo continuariam sendo coletadas para os números de processos menores (4, 2 e 1), até que fosse observada perda de desempenho. Assim que a obtenção das amostras fosse finalizada para o tamanho de entrada 1000×1000, a execução da aplicação seria feita para o próximo tamanho de entrada que, no caso do exemplo, seria 900×900.

Portanto, a condição de parada do *auto-tuner* para um dado tamanho de entrada é o número de processos. O *auto-tuner* diminui o número de processos enquanto for observado ganho de desempenho, e interrompe a execução de um determinado tamanho de entrada caso a aplicação não apresente ganho por até duas diminuições sucessivas do número de processos. Dessa forma, o *auto-tuner* busca o menor número de recursos computacionais pelo máximo de desempenho, conjuntamente com a variação do número de regiões sobrepostas. Além disso, pela variação do número de processos se dá condições da aplicação assumir diferentes *layouts* da decomposição de domínios, variando, indiretamente, os tamanhos dos subdomínios e das regiões de sobreposição entre processos.

4.6.2 Execução da aplicação pelo auto-tuner

Após a base de amostras ter sido construída, a aplicação pode ser executada por intermédio do *auto-tuner*, de forma a possibilitar a sintonia dos parâmetros de acordo com o tamanho da entrada. A execução é feita de forma semelhante à execução de uma aplicação MPI, porém informando os parâmetros cujos valores deverão ser escolhidos pelo *auto-tuner*, assim como demonstra a seguinte linha:

```
./mpiexec.rb -n p[] ./executavelParalelo 800 800 ngz[]
```

Os valores dos parâmetros p e ngz são escolhidos com base no tamanho do domínio. No caso do exemplo, o *auto-tuner* consultaria na base de amostras os valores de parâmetros para o tamanho 800×800. Caso seja informado um tamanho não existente na base de amostras, o *auto-tuner* recupera os valores de parâmetros do tamanho mais próximo ao informado. Por exemplo, caso existam as amostras para os tamanhos 800×800 e 850×850 e a aplicação for executada com um domínio de tamanho 820×820, o *auto-tuner* se basearia no tamanho 800×800 existente na base de amostras.

4.7 Considerações do capítulo

A classe `Mesh` desenvolvida possibilitou um meio de variar o número das regiões de sobreposição entre subdomínios, de forma que o número de regiões possa ser exposto para a aplicação na forma de um parâmetro, que possibilite ao *auto-tuner* tratá-lo como um fator da aplicação para explorar uma faixas de valores.

Além disso, os diferentes valores para a extensão das regiões sobrepostas podem ser sintonizados com os números de processos da aplicação paralela na arquitetura. Ambos níveis de fatores devem ser informados pelo usuário ao *auto-tuner* no momento da construção da base de amostras. Portanto, o usuário deve ter conhecimento dos recursos computacionais disponíveis na máquina paralela, de modo a informar intervalos de valores para o número de processos dentro do número de nós existentes.

A seguir, o Capítulo 5 apresenta os resultados experimentais obtidos através da execução do *auto-tuner* e da variação das regiões de sobreposição.

5 RESULTADOS EXPERIMENTAIS

O Capítulo 4 apresentou a classe `Mesh`, que possibilita o recálculo local das regiões de sobreposição, além do funcionamento do *auto-tuner* desenvolvido, que se baseia no princípio da adaptação parametrizada. O presente capítulo tem como objetivo avaliar o comportamento do *auto-tuner* por meio da execução da aplicação da Equação do Calor em um agregado de máquinas (*cluster*), variando o número de processos, a dimensão do domínio e a extensão das regiões de sobreposição. É avaliado também o impacto que o aumento da extensão das regiões causa no desempenho da aplicação nessa arquitetura. A avaliação é feita, em parte, com base em uma modelagem teórica do algoritmo da Equação do Calor, apresentada no decorrer do capítulo.

A Seção 5.1 apresenta a aplicação da Equação do Calor e o modelo teórico que será utilizado para avaliar o efeito do aumento da extensão das regiões de sobreposição. A Seção 5.2 apresenta a metodologia adotada para a execução dos experimentos. Os resultados obtidos são apresentados na Seção 5.3. Por fim, as considerações do capítulo são apresentadas na Seção 5.4.

5.1 Aplicação da Equação do Calor

A Equação do Calor, discutida na Seção 3.3, calcula as posições de um domínio bidimensional através de um estêncil de 5 pontos (Figura 3.3(a)). Os processos da aplicação determinam os pontos dos respectivos subdomínios até que o resultado global satisfaça um teste de convergência, ou um número máximo de iterações seja atingido. O algoritmo da aplicação é exposto na Seção 5.1.1 a seguir, e o custo de cálculo e comunicação do algoritmo é desenvolvido na Seção 5.1.2.

5.1.1 Algoritmo

O laço principal da aplicação da Equação do Calor é apresentado pelo Código 5.1 [1]. Internamente, a classe `Mesh` aloca 2 subdomínios A e B de forma a permitir o teste de convergência da solução. O laço principal calcula A e B em dois momentos: no primeiro momento, o estêncil é aplicado sobre B e o resultado é atribuído para os pontos de A (linhas 2 a 5); no segundo momento, o estêncil é aplicado sobre A e o resultado é atribuído para as células do subdomínio B (linhas 9 a 12). O método `swap` (linhas 7 e 14) é responsável por fazer a troca entre A e B.

Como o algoritmo pode levar um número elevado de iterações até convergir, o teste de convergência é feito a cada `iter_skip` iterações. A função `calc_diff` calcula localmente a diferença entre os subdomínios A e B, e a função `MPI_Allreduce` é uma

[1] Código-fonte disponível em https://github.com/lxalmeida/Poisson2D

operação coletiva entre os processos da aplicação, responsável por calcular a soma de todas as diferenças locais com o objetivo de determinar a convergência global do resultado.

O custo teórico de desempenho do Código 5.1 é apresentado na Seção 5.1.2. A modelagem será utilizada na Seção 5.3.1 como base para compreender o efeito do aumento das regiões de sobreposição na aplicação paralela.

```
1    while(k < max_iter && diff_norm > 1.0E-5){
2        mesh.send_borders();
3        inner_jacobi_iter(mesh);
4        mesh.sync();
5        outer_jacobi_iter(mesh);

7        mesh.swap();

9        mesh.send_borders();
10       inner_jacobi_iter(mesh);
11       mesh.sync();
12       outer_jacobi_iter(mesh);

14       mesh.swap();

16       if((k%iter_skip) == 0){
17           diff = calc_diff(mesh);
18           MPI_Allreduce(&diff, &diff_norm, 1, MPI_DOUBLE, MPI_SUM,
19                                                   mesh.topology->get_comm());
20       }

22       k++;
23   }
```

Código 5.1: Laço principal da aplicação da Equação do Calor

5.1.2 Custo de computação e comunicação

As equações que apresentam os custos de cálculo e comunicação assumem as variáveis a seguir, que são também ilustradas na Figura 5.1:

- M: tamanho do domínio da aplicação;
- s: extensão das regiões de sobreposição;
- w: largura do estêncil;
- g: vazão de rede;
- α: tempo necessário para o processador calcular 1 ponto através do estêncil;
- P: número de processos;
- t: número de *threads*.

O cálculo do subdomínio é feito enquanto a atualização das regiões de sobreposição está em curso. Na linha 2 do Código 5.1 é iniciada a atualização das regiões, que pode ocorrer ou pela rede ou por recálculo, e na linha 3 é iniciado o cálculo do miolo do subdomínio. O tempo de recebimento de uma região de sobreposição é definido como H_{com} na Equação (5.1).

$$H_{com} = s \cdot w \cdot \frac{M}{g\sqrt{P}} \tag{5.1}$$

Em contrapartida à atualização das regiões de sobreposição pela rede, que ocorre a cada s iterações, a atualização por recálculo acontece durante as $s-1$ iterações restantes. A cada iteração i, onde $i < s$, as classes `Halos` calculam uma linha a menos da região, até que a atualização seja possível apenas por meio do recebimento das bordas dos subdomínios vizinhos, quando $i = s$. O custo de recálculo de uma região de sobreposição durante $s-1$ iterações segue o comportamento da Equação (5.2).

$$H_{proc} = \frac{M}{\sqrt{P}} \cdot \frac{s(s-1)}{2} \cdot \alpha. \tag{5.2}$$

Em relação ao cálculo do miolo do subdomínio na linha 3 e das fronteiras na linha 5, o processamento de ambos conjuntos de dados é realizado por *threads* POSIX. O tempo de processamento do miolo e das fronteiras são definidos por M_{proc} e B_{proc}, respectivamente, através das Equações (5.3) e (5.4).

$$M_{proc} = \frac{\left(\frac{M^2}{P} - \frac{4M \cdot w}{\sqrt{P}} + 4\right)}{t} \cdot \alpha \ . \tag{5.3}$$

$$B_{proc} = \frac{\left(\frac{4M \cdot w}{\sqrt{P}} - 4\right)}{t} \cdot \alpha \ . \tag{5.4}$$

Para cada ciclo de s iterações, durante $s-1$ iterações incorre os tempos de M_{proc}, B_{proc} e H_{proc}; já na iteração s, estão presentes os custos de M_{proc}, B_{proc} e H_{com}. Para o caso da iteração s, M_{proc} e H_{com} ocorrem em paralelo. Portanto, deve ser considerado o máximo entre ambos os valores. No caso das $s-1$ iterações, como H_{proc} já considera o cálculo desse intervalo de iterações, ele é somado ao custo total. Essa relação é apresentada pela Fórmula (5.5).

$$(s-1)(M_{proc} + B_{proc}) + (Max(M_{proc}, H_{com}) + B_{proc}) + H_{proc} \tag{5.5}$$

A Fórmula (5.5) define o custo de um ciclo de s iterações. Dessa forma, o tempo total de processamento do laço do Código 5.1 é obtido multiplicando-se a Fórmula (5.5) pelo número total de ciclos, que é definido como $\frac{k}{s}$, sendo k igual ao número de iterações. Pelo fato do laço calcular os subdomínios A e B, se considera que cada iteração do laço seja igual a duas iterações. A Equação (5.6) define T_p como a estimativa de tempo total de processamento.

$$T_p = \frac{2k}{s} \cdot ((s-1)(M_{proc} + B_{proc}) + (Max(M_{proc}, H_{com}) + B_{proc}) + H_{proc}) \tag{5.6}$$

5.2 Metodologia

Os experimentos foram realizados no *cluster* Xiru do Instituto de Informática da Universidade Federal do Rio Grande do Sul (II-UFRGS). O *cluster* Xiru é formado por 14 nós

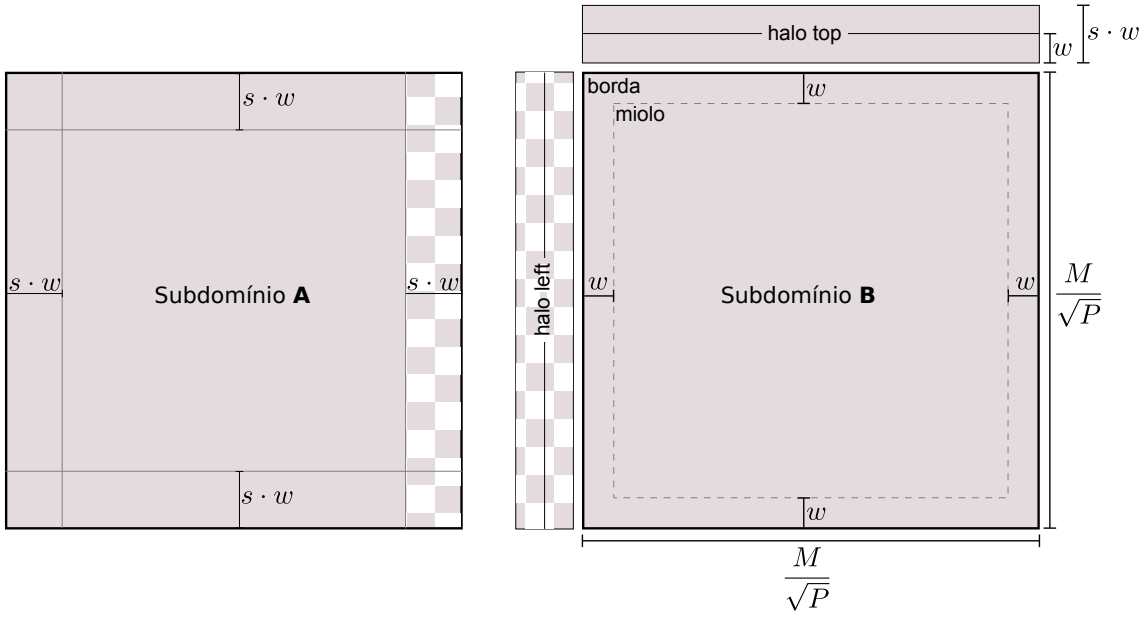


Figura 5.1: O subdomínio B está sobreposto com o subdomínio A por intermédio de `Halo_Left`, que gerencia no exemplo $s = 2$ regiões de sobreposição, cada uma de tamanho w.

de processamento, onde cada nó possui 2 processadores *quad-core* Intel Xeon E5310 1,6 GHz, com 16 GB de memória principal. Os nós são interconectados por uma rede Gigabit Ethernet através de um *switch*.

Para verificar o impacto no aumento das regiões de sobreposição, a aplicação da Equação do Calor foi executada para domínios de tamanho 10.000×10.000, 15.000×15.000, 20.000×20.000 e 25.000×25.000, variando a extensão das regiões em 1 a 20 ao passo de 2 (1, 2, 4, ..., 18, 20), e mantendo fixo o número de processos em 9. A aplicação foi lançada para a execução com 8 *threads*, e cada processo foi alocado em um nó do *cluster*. Os tempos de processamento obtidos correspondem à execução do laço apresentado no Código 5.1, de forma a ser possível comparar a modelagem teórica com os tempos observados.

A execução do *auto-tuner* foi feita no mesmo ambiente para o mesmo executável da aplicação, e a construção da base de amostras foi realizada para dois intervalos de domínios. A primeira chamada assumiu o intervalo r[100,1000,100] c[100,1000,100], e a segunda chamada foi feita para o intervalo r[1000,15000,1000] c[1000,15000,1000]. Os níveis para o fator do número de processos e para a extensão das regiões de sobreposição foram definidos como p[1,2,4,6,8,9] e ngz[2,10,2], respectivamente. Destaca-se que os tempos obtidos pelo *auto-tuner* são relativos ao tempo de toda a aplicação paralela, e não apenas o laço principal da Equação do Calor.

Para todos os casos, o número máximo de iterações da aplicação foi fixado em 200, de forma a ser possível comparar os tempos de processamento, uma vez que o número de iterações até a convergência tende a variar de acordo com a entrada. No caso dos experimentos sem o uso do *auto-tuner*, foram coletadas 50 amostras de tempo para cada cenário de execução. No caso do *auto-tuner*, cada amostra de tempo coletada é a média aritmética de 2 execuções.

5.2.1 Ajuste do modelo teórico

Os parâmetros g e α do modelo teórico apresentado na Seção 5.1.2 são dependentes da arquitetura onde a aplicação será executada e, portanto, devem ser medidos para que seja possível estimar os tempos de processamento.

O valor de α foi medido para cada um dos domínios de tamanhos 10.000×10.000, 15.000×15.000, 20.000×20.000 e 25.000×25.000, decompostos entre 9 processos. Como α representa o tempo necessário para o processador calcular 1 ponto do subdomínio através do estêncil, foram obtidas 100 amostras de tempo apenas do processamento do miolo em um determinado processo. A aplicação foi lançada com 8 *threads* por processo. O valor de α foi determinado dividindo a média aritmética das amostras de tempo pelo número total de pontos do subdomínio por *thread*. A Tabela 5.1 apresenta os valores de α utilizados no modelo.

Tamanho do subdomínio	Média (s)	Desv. pad. (%)	α (s)
8.334×8.334	0,312	1,760	3,591E-08
6.667×6.667	0,205	2,336	3,695E-08
5.000×5.000	0,142	4,745	4,538E-08
3.334×3.334	0,092	5,435	6,653E-08

Tabela 5.1: Valores do tempo de processamento de 1 ponto do subdomínio (α)

O valor da vazão de rede, representado por g, foi determinado através da média de 100 amostras de tempo de recebimento relativo a um conjunto de 16.384.000 valores de ponto flutuante de precisão dupla, cada qual com 8 *bytes* de tamanho. A média dos tempos foi de 1,136 segundo, com desvio padrão de 1,785%, o que resulta em $g = 110,04$ MB/s.

Ainda em relação à vazão de rede, como os nós do *cluster* possuem apenas 1 canal de comunicação *full-duplex*, dividiu-se o valor da vazão pelo número de comunicações em curso, que no caso é 4.

5.3 Resultados obtidos

O impacto no aumento das regiões de sobreposição é apresentado na Seção 5.3.1. As amostras de tempo coletadas nesse grupo de experimentos são representadas por meio de gráficos do tipo *boxplot*. O *boxplot* representa, para cada extensão das regiões de sobreposição, o valor mínimo e o valor máximo obtido. Entre esses valores, uma caixa retangular representa o quartil inferior, a mediana, e o quartil superior das amostras. Os pontos anômalos obtidos (*outliers*) são representados por círculos em cada grupo de amostras. Os tempos estimados pelo modelo teórico foram desenhados nos mesmos gráficos, na forma de séries com linhas seccionadas.

A média aritmética e o respectivo desvio padrão são apresentados de forma tabular. Os valores da média são representados nos gráficos por meio de uma linha não seccionada.

Os resultados da execução do *auto-tuner* são apresentados na Seção 5.3.2 sob a forma de gráficos produzidos a partir da base de amostras gerada pelo *auto-tuner*. As séries de dados dos gráficos representam os números de processos que o *auto-tuner* utilizou para as dimensões de domínio.

5.3.1 Impacto na variação das regiões de sobreposição

A Figura 5.2 e a Tabela 5.2 apresentam os resultados obtidos para domínios de tamanho 10.000×10.000 e 15.000×15.000; os resultados para os domínios de tamanho 20.000×20.000 e 25.000×25.000 são apresentados pela Figura 5.3 e pela Tabela 5.3.

Os experimentos demonstraram que o aumento das regiões de sobreposição não reduz o tempo de execução da aplicação na arquitetura disponível. A perda de desempenho pode ser explicada pela concorrência por recursos entre o recálculo das regiões de sobreposição e os fluxos de execução que processam o miolo do subdomínio.

Pelo modelo teórico, a atualização das regiões por recálculo seria justificada quando o tempo de processamento do miolo do subdomínio fosse inferior ao tempo de transferência das bordas pela rede de comunicação. Nesse caso, o desempenho da aplicação seria comprometido pelo bloqueio da execução nos pontos de sincronismo do Código 5.1, presentes nas linhas 4 e 11.

A Inequação 5.7 relaciona o custo de cálculo do miolo do subdomínio com o custo de recebimento de 1 região de sobreposição na arquitetura utilizada. O tempo de cálculo do subdomínio é ditado pelo valor de α, que define o tempo necessário para o processador calcular 1 ponto por meio do estêncil. O tempo de transferência da região de sobreposição pela rede é definido por g, que denota a vazão de rede.

$$\frac{\left(\frac{M^2}{P} - \frac{4M}{\sqrt{P}} + 4\right)}{t} \cdot \alpha < \frac{8bytes \cdot M}{0,25 \cdot g\sqrt{P}} \tag{5.7}$$

Para que o aumento da extensão das regiões de sobreposição seja justificado através dessa relação, ou o valor de α ou o valor de g deve ser pequeno o suficiente tal que satisfaça a desigualdade.

Considerando o cenário de testes com um domínio de tamanho 25.000×25.000, um valor de α que satisfizesse a Inequação 5.7 seria de 2,793E-10. Em comparação, na arquitetura utilizada foi medido um α igual a 3,591E-08 para o mesmo tamanho de domínio, conforme a Tabela 5.1. Levando em conta que o poder de processamento tende a evoluir mais rapidamente que a taxa de transmissão das redes de computador, futuramente o aumento da extensão das regiões de sobreposição pode ser justificado na arquitetura utilizada.

De uma forma mais geral, os processos da aplicação extrairiam ganho de desempenho através do processamento das regiões de sobreposição sempre que o custo de acesso aos dados do subdomínio vizinho tiver um alto custo em termos de tempo. Um exemplo dessa situação é o processamento das regiões de sobreposição em GPGPUs, onde o custo de acesso à memória compartilhada é muito superior ao custo de acesso à *cache* local, assim como apresentado pelo trabalho descrito na Seção 3.5.2.4.

Como consequência dos resultados obtidos pela variação das regiões de sobreposição, o ganho de desempenho obtido pelo *auto-tuner* acaba sendo decorrente da variação do número de processos da aplicação. Os resultados da execução do *auto-tuner* na arquitetura são apresentados na Seção 5.3.2 a seguir.

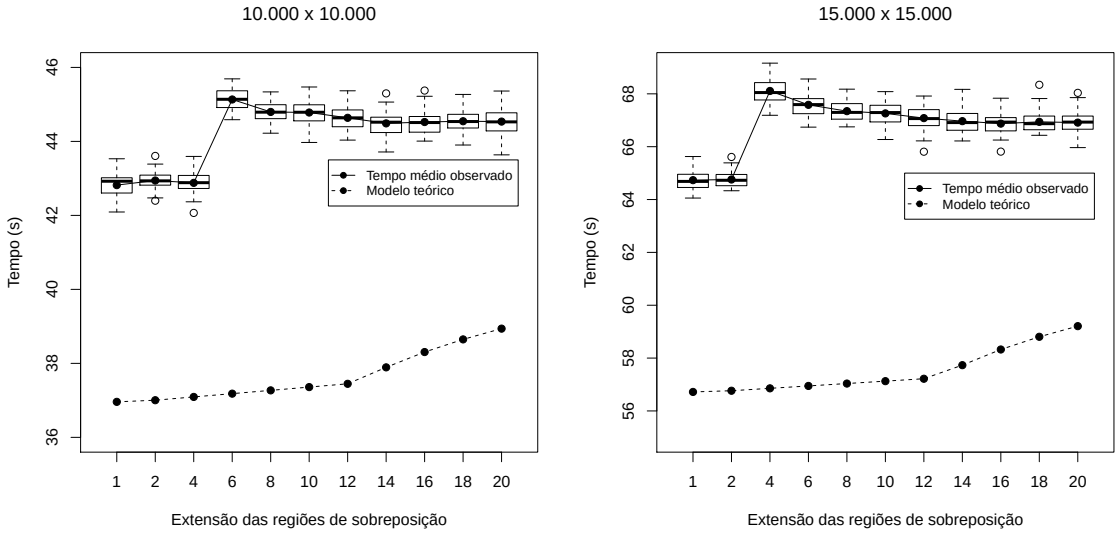


Figura 5.2: Resultados experimentais para domínios de tamanho 10.000×10.000 e 15.000×15.000

	10.000×10.000		15.000×15.000	
Extensão das regiões	Média (s)	Desv. pad. (%)	Média (s)	Desv. pad. (%)
1	42,819	0,682	64,734	0,557
2	42,943	0,586	64,759	0,430
4	42,879	0,741	68,109	0,649
6	45,134	0,608	67,585	0,649
8	44,797	0,602	67,346	0,553
10	44,781	0,718	67,260	0,593
12	44,638	0,707	67,082	0,702
14	44,487	0,709	66,967	0,626
16	44,524	0,704	66,874	0,552
18	44,546	0,727	66,941	0,560
20	44,536	0,768	66,912	0,652

Tabela 5.2: Média aritmética e desvio padrão das amostras para domínios de tamanho 10.000×10.000 e 15.000×15.000

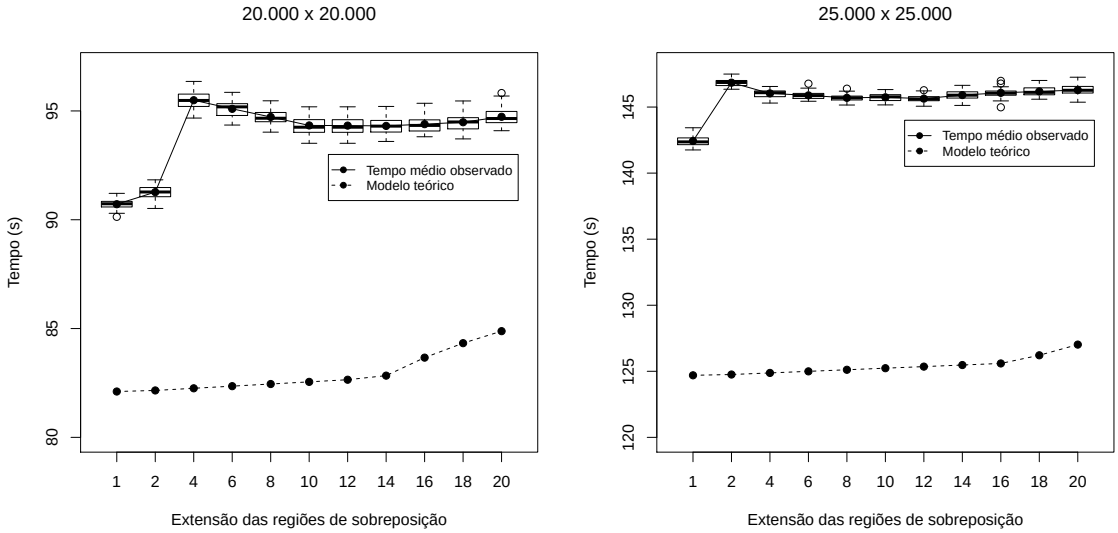


Figura 5.3: Resultados experimentais para domínios de tamanho 20.000×20.000 e 25.000×25.000

	20.000×20.000		25.000×25.000	
Extensão das regiões	Média (s)	Desv. pad. (%)	Média (s)	Desv. pad. (%)
1	90,714	0,236	142,430	0,263
2	91,270	0,294	146,846	0,200
4	95,493	0,421	146,033	0,196
6	95,097	0,409	145,885	0,182
8	94,724	0,334	145,698	0,192
10	94,334	0,405	145,739	0,203
12	94,327	0,396	145,639	0,194
14	94,312	0,387	145,908	0,252
16	94,393	0,376	146,076	0,220
18	94,481	0,424	146,184	0,238
20	94,732	0,434	146,288	0,241

Tabela 5.3: Média aritmética e desvio padrão das amostras para domínios de tamanho 20.000×20.000 e 25.000×25.000

5.3.2 Execução do auto-tuner

A Figura 5.4 apresenta as coletas realizadas pelo *auto-tuner* para domínios de tamanho 100×100, 200×200, 300×300 e 400×400. As séries de dados dos quatro gráficos pertencem ao grupo de amostras coletadas pela chamada que explorou o intervalo de domínios r[100,1000,100] c[100,1000,100]. A Figura 5.5 ilustra as coletas para domínios de tamanho 8000×8000, 9000×9000, 10000×10000 e 11000×11000, que fazem parte do intervalo r[1000,15000,1000] c[1000,15000,1000].

O comportamento do *auto-tuner* é melhor evidenciado pela Figura 5.4, onde toda a faixa de número de processos é explorada para os domínios de tamanho 100×100 e

200×200. Esses dois casos mostram que ambas as dimensões possuem um tamanho que não justifica o sobrecusto de execução com o valor máximo possível de processos.

No caso de 100×100, a execução com 1 processo foi o caso que apresentou melhor desempenho. É importante destacar que o aumento da região de sobreposição para esse caso não tem efeito algum, uma vez que não existe paralelismo. Portanto, o aparente ganho de desempenho representado pela série é meramente uma variação dos tempos obtidos.

Para o domínio de tamanho 300×300, o *auto-tuner* identificou 8 processos e 2 regiões de sobreposição como um valor ótimo. Ainda assim, o *auto-tuner* executou a aplicação com 6 e 4 processos. Não observando ganho de desempenho pela diminuição, a coleta foi interrompida para esse tamanho de domínio. A partir de 400×400, o número ótimo de processos foi sempre identificado como 9 e a extensão das regiões de sobreposição como igual a 2.

A execução com os intervalos r[1000,15000,1000] c[1000,15000,1000] segue um comportamento semelhante às execuções a partir do domínio 400×400, como observado pela Figura 5.5. Isso é, o número ideal de processos foi identificado como 9, verificando sempre dois números de processos "para trás".

Em relação ao tempo de *auto-tuning*, o *auto-tuner* levou em torno de 5 minutos para construir a base de amostras para o intervalo r[100,1000,100] c[100,1000,100]. Para este caso, o espaço de busca foi constituído de 300 linhas de execução, dentre as quais 185 foram executadas na arquitetura. No caso do intervalo r[1000,15000,1000] c[1000,15000,1000], o tempo foi de, aproximadamente, 5 horas e 30 minutos. O espaço de busca gerado foi de 450 linhas de execução, dentre as quais 225 foram executadas para a obtenção das amostras de tempo.

5.4 Considerações do capítulo

Como exposto na Seção 5.3.1, o recálculo das regiões de sobreposição seria adequado em uma arquitetura com um poder de processamento tal que o tempo de cálculo fosse inferior ao tempo de comunicação dos dados dos subdomínios vizinhos, assim como aconteceria em GPGPUs ou em máquinas paralelas largamente distribuídas (*cluster of clusters* ou *grids*).

Embora o aumento da extensão das regiões de sobreposição tenha se mostrado inadequado para a arquitetura paralela utilizada, ainda assim a execução do *auto-tuner* buscou o número mínimo de processos necessário para a aplicação apresentar ganho de desempenho em um *cluster* de computadores. Com isso, o *auto-tuner* possibilitou identificar casos em que a execução com o número total de processos causa prejuízo no desempenho da aplicação. Posteriormente, o usuário seria capaz de executar a aplicação informando apenas o tamanho da entrada, de forma que o *auto-tuner* se encarregue de lançar a aplicação com o número ideal de processos por meio do MPI.

O Capítulo 6 a seguir apresenta as conclusões gerais da dissertação.

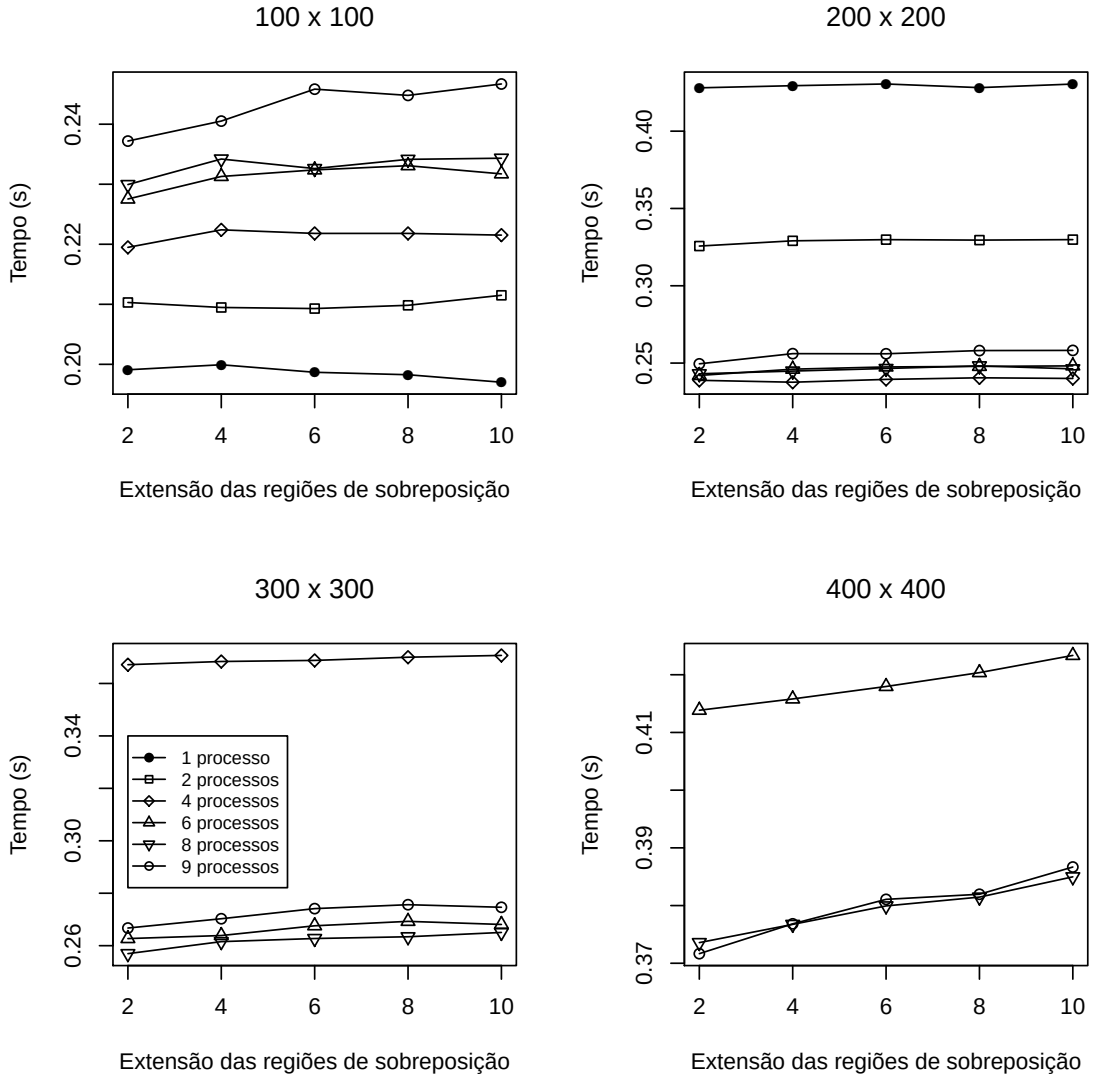


Figura 5.4: Coletas realizadas pelo *auto-tuner* para domínios de tamanho 100×100, 200×200, 300×300 e 400×400

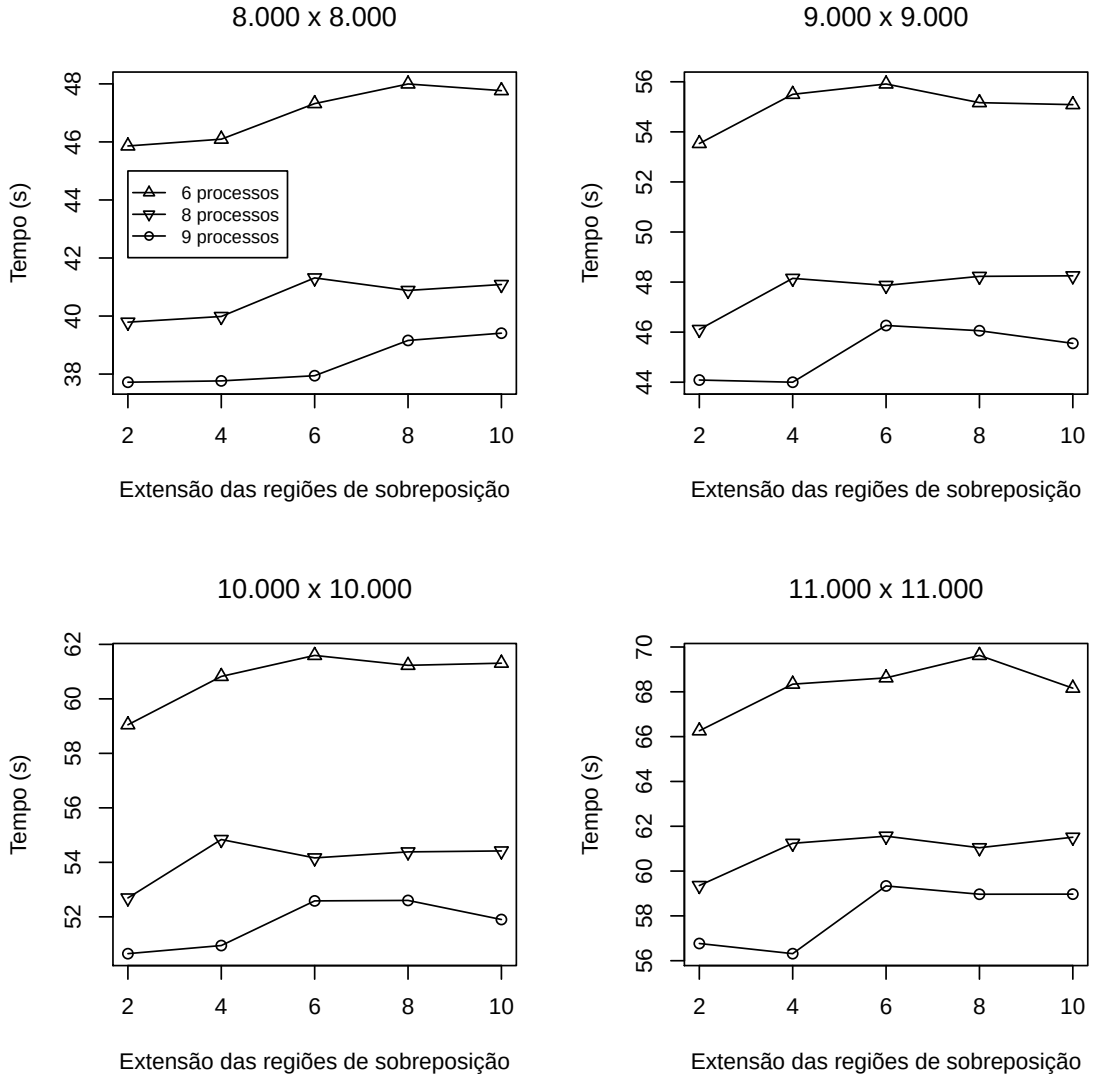


Figura 5.5: Coletas realizadas pelo *auto-tuner* para domínios de tamanho 8000×8000, 9000×9000, 10000×10000 e 11000×11000

6 CONCLUSÃO

Dentro do escopo de aplicações paralelas que processam domínios bidimensionais utilizando estênceis, este trabalho explorou a técnica de extensão e atualização por recálculo das regiões sobrepostas entre subdomínios, de forma a expor o nível de extensão como um fator e, consequentemente, possibilitar a adaptação de uma aplicação através de *auto-tuning* em um agregado de máquinas. A extensão das regiões de sobreposição foi facilitada pela criação da classe `Mesh`, que auxiliou na abstração da decomposição de domínios da aplicação paralela.

Os resultados experimentais demonstraram que a otimização da decomposição de domínios pelo aumento da extensão das regiões de sobreposição se mostrou imprópria para a arquitetura paralela utilizada. Como consequência, o ganho de desempenho pelo *auto-tuner* ocorreu unicamente pela variação do número de processos da aplicação paralela. Entretanto, a futura elevação do poder computacional pode tornar a atualização das regiões de sobreposição por recálculo vantajosa, uma vez que o processamento tenderia a finalizar antes da comunicação, causando perda de desempenho decorrente da espera dos dados pela rede. Ainda assim, o papel do *auto-tuner* foi importante por variar fatores da aplicação independentes entre si, de forma a tentar descobrir empiricamente possíveis interações entre os parâmetros variados.

Em relação à decomposição de domínios, que foi outra área abordada no trabalho, constatou-se que o estêncil de uma aplicação é uma entidade central que define determinadas características da aplicação paralela. Pela forma modular como a classe `Mesh` foi concebida, uma descrição em alto nível do estêncil poderia adaptá-la para outros tipos de formatos, a partir de um sintetizador de código que alterasse pontos específicos da classe. A descrição do estêncil permitiria ainda a geração de um *kernel* para o processamento do subdomínio, que fosse otimizado para uma determinada arquitetura (por exemplo, um *kernel* em CUDA em um agregado com GPGPUs).

A respeito da área de *auto-tuning*, de uma forma geral, os métodos e definições encontrados na literatura sobre a esta área ainda são focados na adaptação de *kernels* de bibliotecas numéricas, principalmente por essa técnica ter nascido dentro da classe de software numérico. Entretanto, um software paralelo possui comumente outros aspectos que necessitariam ser adaptados às características de uma arquitetura, além de um *kernel* numérico. Por exemplo, enquanto aplicações numéricas devem adaptar um ou mais *kernels* específicos, uma aplicação paralela recursiva (*e.g.*, uma ordenação paralela) possui um determinado limite do nível de recursão (*threshold*), a partir do qual os processos devem resolvem os respectivos subproblemas sequencialmente. Nesse caso, o conceito de *kernel* deixa de existir, dando lugar a outros aspectos da aplicação que, da mesma forma, poderiam ser adaptados ao hardware.

6.1 Contribuições

Este trabalho teve como objetivo aplicar *auto-tuning* em aplicações paralelas que lidam com decomposição de domínios estruturados, sendo o *auto-tuning* focado no aumento da extensão das regiões de sobreposição entre os subdomínios. A principal contribuição do trabalho foi a disponibilização da classe `Mesh` ao grupo de pesquisa, e de um módulo *auto-tuner* que lida com a adaptação parametrizada de uma aplicação paralela em MPI. Outras contribuições do trabalho foram:

- avaliar o efeito do aumento da extensão das regiões de sobreposição no *cluster* Xiru;
- evidenciar o estêncil de uma aplicação que trabalha com decomposição de domínios como uma entidade de alto nível;
- constatar que *auto-tuning* é uma área que deveria sofrer uma formalização, de forma a torná-la geral para a otimização de outras classes de software;
- trabalhar com a biblioteca MPI como um *backend*, abstraindo detalhes técnicos especificados pelo padrão MPI através da Orientação a Objetos.

Um artigo foi submetido e aceito na Conferência Latinoamericana de Computação de Alto Desempenho (CLCAR'2011) (ALMEIDA; MAILLARD, 2011), além de um resumo na Escola Regional de Alto Desempenho (ERAD'2010) (ALMEIDA; MAILLARD, 2010).

6.2 Trabalhos futuros

Como continuidade do trabalho, o *auto-tuner* poderá ser avaliado em uma arquitetura paralela amplamente distribuída como *cluster of clusters* ou em *grid*, onde a latência ou vazão de rede tenderiam a comprometer a comunicação entre processos. Em relação à classe `Mesh`, assim como exposto na Seção 4.5, a criação de uma ferramenta para adaptá-la a diferentes formatos de estênceis poderia ser desenvolvida, de forma a estendê-la a outras geometrias.

Transcendendo aplicações que trabalham sobre domínios estruturados, o uso das técnicas de aumento da extensão das regiões de sobreposição e recálculo poderiam ser avaliadas em aplicações que trabalham sobre domínios não estruturados, como na aplicação de modelagem climatológica OLAM (WALKO et al., 2000).

REFERÊNCIAS

AHO, A. V.; SETHI, R.; ULLMAN, J. D. **Compiladores**: princípios, técnicas e ferramentas. Rio de Janeiro, RJ: LTC, 1995.

ALLEN, G. et al. Supporting Efficient Execution in Heterogeneous Distributed Computing Environments with Cactus and Globus. In: SUPERCOMPUTING, ACM/IEEE 2001 CONFERENCE. **Anais...** [S.l.: s.n.], 2001. p.52–52.

ALMEIDA, A.; MAILLARD, N. Auto-tuning de Regiões de Sobreposição Heterogêneas para Domínios Estruturados em Ambientes Paralelos. In: ERAD'2010: 10ª ESCOLA REGIONAL DE ALTO DESEMPENHO, Passo Fundo, RS. **Anais...** [S.l.: s.n.], 2010. p.93–94.

ALMEIDA, A.; MAILLARD, N. Abstração de Decomposição de Domínios para Aplicações Paralelas com MPI. In: CLCAR'2011: CONFERENCIA LATINOAMERICANA DE COMPUTACIÓN DE ALTO RENDIMIENTO, Colima, Mexico. **Anais...** [S.l.: s.n.], 2011.

Argone National Laboratory. **MPICH2 Home Page**. MPICH2: High-performance and Widely Portable MPI, disponível em: http://www.mcs.anl.gov/research/projects/mpich2, acesso em: 31 ago. 2011.

ASANOVIC, K. et al. A view of the parallel computing landscape. **Commun. ACM**, New York, NY, USA, v.52, p.56–67, October 2009.

BALAY, S. et al. Efficient Management of Parallelism in Object Oriented Numerical Software Libraries. In: MODERN SOFTWARE TOOLS IN SCIENTIFIC COMPUTING. **Anais...** Birkhäuser Press, 1997. p.163–202.

BALAY, S. et al. **PETSc Users Manual**. [S.l.]: Argonne National Laboratory, 2008. (ANL-95/11 - Revision 3.0.0).

BILMES, J. et al. **The PHiPAC v1.0 Matrix-Multiply Distribution**. [S.l.]: University of California at Berkeley, 1998. TR-98-35.

CHRISTEN, M.; SCHENK, O.; BURKHART, H. Patus: a code generation and autotuning framework for parallel iterative stencil computations on modern microarchitectures. In: IEEE INTERNATIONAL PARALLEL DISTRIBUTED PROCESSING SYMPOSIUM, 25., Anchorage, Alaska, USA. **Anais...** [S.l.: s.n.], 2011.

CORMEN, T. H. et al. **Algoritmos**: teoria e prática. 2.ed. Rio de Janeiro: Campus, 2002.

DATTA, K. **Auto-tuning Stencil Codes for Cache-Based Multicore Platforms**. 2009. Tese (Doutorado em Ciência da Computação) — EECS Department, University of California, Berkeley. (UCB/EECS-2009-177).

DATTA, K. et al. Stencil Computation Optimization and Auto-tuning on State-of-the-Art Multicore Architectures. In: SC '08: PROCEEDINGS OF THE 2008 ACM/IEEE CONFERENCE ON SUPERCOMPUTING, Piscataway, NJ, USA. **Anais...** IEEE Press, 2008. p.1–12.

DONGARRA, J. Preface: Basic Linear Algebra Subprograms Technical (Blast) Forum Standard I. **The International Journal of High Performance Computing Applications**, [S.l.], v.16, n.1, p.1–111, Spring 2002.

FORUM, M. P. I. **MPI**: A Message-Passing Interface Standard. Knoxville, Tennessee, Estados Unidos: University of Tennessee, 2009. Disponível em: http://www.mpi-forum.org/docs/docs.html, acesso em: jul. de 2011.

FOSTER, I. T. Globus Toolkit Version 4: software for service-oriented systems. In: NPC. **Anais...** Springer, 2005. p.2–13. (Lecture Notes in Computer Science, v.3779).

FRIGO, M. A Fast Fourier Transform Compiler. In: ACM SIGPLAN CONF. ON PROGRAMMING LANGUAGE DESIGN AND IMPLEMENTATION, 1999. **Proceedings...** ACM, 1999. v.34, n.5, p.169–180.

FRIGO, M. et al. Cache-Oblivious Algorithms. In: FOCS '99: PROCEEDINGS OF THE 40TH ANNUAL SYMPOSIUM ON FOUNDATIONS OF COMPUTER SCIENCE, Washington, DC, USA. **Anais...** IEEE Computer Society, 1999. p.285.

FRIGO, M.; JOHNSON, S. G. The Design and Implementation of FFTW3. **Proceedings of the IEEE**, [S.l.], v.93, n.2, p.216–231, 2005. Special issue on "Program Generation, Optimization, and Platform Adaptation".

GALANTE, G. **Métodos multigrid paralelos em malhas não estruturadas aplicados à simulação de problemas de dinâmica de fluidos computacional e transferência de calor**. 2006. Dissertação de Mestrado — Universidade Federal do Rio Grande do Sul.

GARDNER, M. Mathematical Games: The fantastic combinations of John Conway's new solitaire game 'Life'. **Scientific American**, [S.l.], v.223, n.4, p.120–123, Oct. 1970.

GOODALE, T. et al. The Cactus Framework and Toolkit: Design and Applications. In: VECTOR AND PARALLEL PROCESSING – VECPAR '2002, 5TH INTERNATIONAL CONFERENCE. **Anais...** Springer, 2003.

GROPP, W.; LUSK, E.; SKJELLUM, A. **Using MPI**: Portable Parallel Programming with the Message-Passing Interface. 2.ed. United States: The MIT Press, 1999.

HUANG, W. et al. Compact thermal modeling for temperature-aware design. In: DAC '04: PROCEEDINGS OF THE 41ST ANNUAL DESIGN AUTOMATION CONFERENCE, New York, NY, USA. **Anais...** ACM, 2004. p.878–883.

KAMIL, S. et al. A Generalized Framework for Auto-tuning Stencil Computations. In: Proceedings of Cray User Group Conference. **Anais...** [S.l.: s.n.], 2009. LBNL-2078E.

LEISERSON, C. E. **Stencil Computing (Lab 3)**. Concepts in Multicore Programming (Lecture), Massachusetts Institute of Technology, disponível em: http://courses.csail.mit.edu/6.884/spring10/, acesso em: 22 abr. 2011.

MENG, J.; SKADRON, K. Performance Modeling and Automatic Ghost Zone Optimization for Iterative Stencil Loops on GPUs. In: ICS '09: PROCEEDINGS OF THE 23RD INTERNATIONAL CONFERENCE ON SUPERCOMPUTING, New York, NY, USA. **Anais...** ACM, 2009. p.256–265.

RIPEANU, M.; IAMNITCHI, A.; FOSTER, I. T. Cactus Application: Performance Predictions in Grid Environments. In: EURO-PAR. **Anais...** Springer, 2001. p.807–816. (Lecture Notes in Computer Science, v.2150).

RIZZI, R. L. **Modelo Computacional Paralelo para a Hidrodinâmica e para o Transporte de Substâncias Bidimensional e Tridimensional**. 2002. Tese de Doutorado — Universidade Federal do Rio Grande do Sul.

SAAD, Y. Data Structures And Algorithms For Domain Decomposition And Distributed Sparse Matrices. **IN PREPARATION, ARMY HIGH PERFORMANCE COMPUTING RESEARCH**, [S.l.], 1994.

SAWDEY, A. et al. The Design, Implementation, and Performance of a Parallel Ocean Circulation Model. In: WORKSHOP ON THE USE OF PARALLEL PROCESSORS IN METEOROLOGY. **Anais...** [S.l.: s.n.], 1994. p.523–550.

SMITH, B.; BJØRSTAD, P.; GROPP, W. **Domain Decomposition**: Parallel Multilevel Methods for Elliptic Partial Differential Equations. Cambridge: Cambridge University, 1996.

STROUSTRUP, B. **The C++ Programming Language**. 3.ed. Massachusetts, USA: Addison-Wesley, 1997.

The Open MPI Project. **Open MPI Home Page**. Open MPI: Open Source High Performance Computing, disponível em: http://www.open-mpi.org/, acesso em: 31 ago. 2011.

WALKO, R. L. et al. Coupled Atmosphere-Biophysics-Hydrology Models for Environmental Modeling. **Journal of Applied Meteorology**, [S.l.], v.39, n.6, p.931–944, 2000.

WHALEY, R. C.; PETITET, A. Minimizing development and maintenance costs in supporting persistently optimized BLAS. **Softw. Pract. Exper.**, New York, NY, USA, v.35, n.2, p.101–121, 2005.

WHALEY, R. C.; PETITET, A.; DONGARRA, J. J. Automated empirical optimizations of software and the ATLAS project. **Parallel Computing**, [S.l.], v.27, n.1-2, p.3 – 35, 2001.

WILLIAMS, S. W. **Auto-tuning Performance on Multicore Computers**. 2008. Tese (Doutorado em Ciência da Computação) — University of California at Berkeley.