Capítulo 2

Projetando e Construindo Programas Paralelos

Lucas Mello Schnorr[1], Lucas Leandro Nesi[2]
Instituto de Informática, Universidade Federal do Rio Grande do Sul
Porto Alegre, Brasil

Resumo

Apresentamos os conceitos de projeto e desenvolvimento de programas paralelos. Ele é fortemente baseado no livro de Ian Foster que porta o título Designing and Building Parallel Programs*, de 1995. Embora antigo, os conceitos apresentados continuam da atualidade. O enfoque será dado para o projeto metodológico chamado* `PCAM`*, oriundo das quatro fases de criação de um programa paralelo: Particionamento, Comunicação, Aglomeração e Mapeamento. Esses quatro eixos serão apresentados de forma a levar o participante a quebrar um problema potencialmente grande em pedações, pensar em como particioná-lo e as implicações do mapeamento em uma determinada máquina paralela. Serão utilizados exemplos práticos tais como simulação de equações físicas simples.*

2.1. Introdução

Grandes esforços são empregados por nações desenvolvidas para colocar a computação como elemento de base no currículo de escolas, complementando espaços como fazem já a matemática, a física, a filosofia e a língua oficial do país.

[1]Lucas Mello Schnorr possui graduação em Ciência da Computação pela Universidade Federal de Santa Maria (2003), mestrado em Computação pela Universidade Federal do Rio Grande do Sul (2005), doutorado em Computação pela Universidade Federal do Rio Grande do Sul com um acordo de cotutela com o Institut National Polytechnique de Grenoble (2009), pós-doutorado pelo Centre National de la Recherche Scientifique (2011) e pós-doutorado pelo Institut National de Recherche en Informatique et en Automatique (2017). Desde 2013 é Professor Adjunto da Universidade Federal do Rio Grande do Sul e orientador do Programa de Pós-Graduação em Computação. Conduz pesquisas em ambiente internacional. Tem experiência na área de Ciência da Computação, com ênfase em Sistemas de Computação e Processamento de Alto Desempenho.

[2]Lucas Leandro Nesi possui graduação em Ciência da Computação pela Universidade do Estado de Santa Catarina (2017) e atualmente persegue um mestrado em Computação pela Universidade Federal do Rio Grande do Sul com bolsa da Coordenação de Aperfeiçoamento de Pessoal de Nível Superior (CAPES). Realiza pesquisas na área de processamento de alto desempenho, especialmente em problemas que envolvem sistemas heterogêneos com múltiplas GPUs e CPUs.

Esses esforços são justificados pois o pensamento computacional, ou mais especificamente, a forma de se programar um computador, é sem dúvida fundamental para esta e a próxima geração de pessoas. Enquanto todos esses esforços devem ser encorajados, a sensação é que pouco se antecipa da importância da programação paralela para a próxima geração de estudantes. O que se observa hoje é que o futuro provavelmente será cada vez mais paralelo, com computadores compostos por múltiplos núcleos de processamento com capacidades diferentes.

No âmbito do paralelismo, o projeto de aplicações paralelas é uma etapa fundamental na resolução de problemas complexos, tais como simulações científicas que implementam equações matemáticas que modelam comportamentos físicos. Estas simulações em geral acabam por substituir experimentos reais tornando-as mais acessíveis para a comunidade de cientistas. Projetar corretamente aplicações paralelas envolve uma mistura de conhecer bem o problema e como este pode ser trabalhado para se adaptar corretamente a plataforma de execução que será adotada, seja ela um único computador ou um *cluster* computacional. Enfim, as escolhas na fase de projeto são fundamentais para se obter um bom desempenho e um uso eficiente dos recursos de processamento.

Este minicurso trata de maneira abstrata os conceitos fundamentais para o projeto e construção de programas paralelos. O minicurso está estruturado de maneira semelhante ao capítulo 2 do livro [2]. A metodologia de projeto baseia-se na divisão do processo de paralelização em quatro etapas: particionamento, comunicação, aglomeração e mapeamento. Esse projeto metodológico é conhecido como **PCAM**, devido as iniciais de cada uma das quatro etapas. Essa separação reflete uma forma que induz a pensar nos aspectos fundamentais de paralelização, permitindo a quebra de um problema em pedaços pequenos que possam ser justamente calculados ao mesmo tempo por diferentes núcleos de processamento. Ao invés de mergulhar nos detalhes arquiteturais dos computadores, certamente importantes, o minicurso aborda esses conceitos de maneira abstrata. Considera-se que tais conceitos são fundamentais para qualquer pessoa que trabalha na área de paralelismo de alto desempenho.

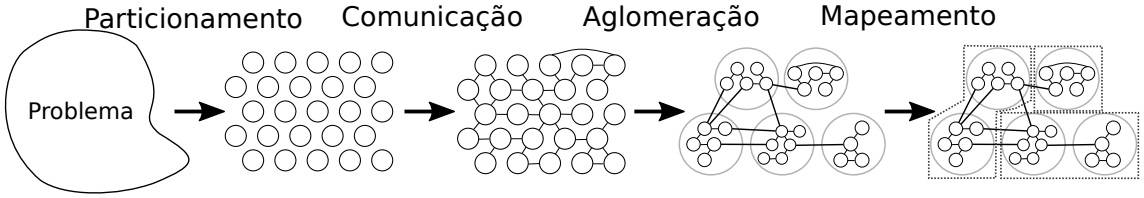

Figura 2.1. Modelo metodológico PCAM com suas quatro fases: particionamento, para quebrar o problema em pedaços menores; comunicação, para definir quais pedaços devem comunicar-se entre si; aglomeração, para agrupar pedaços com critérios de localidade; e mapeamento, para atribuir a unidades computacionais os grupos que devem ser processados.

O modelo **PCAM**, ilustrado na Figura 2.1, envolve portanto quatro etapas: particionamento, comunicação, aglomeração e mapeamento. As duas primeiras etapas tem um enfoque na escalabilidade de uma solução, ou seja, procuramos definir algoritmos capazes de resolver o problema de maneira mais paralela pos-

sível, com maior concorrência (ou seja, mais execução simultânea de processos) entre as unidades de processamento. Nas duas últimas etapas, de aglomeração e mapeamento, a preocupação do projetista se foca na preocupação com desempenho necessitando de um conhecimento mínimo da configuração da plataforma de execução. Segue uma breve descrição de cada uma das etapas:

- **Particionamento**: As operações que resolvem um determinado problema a devem ser quebradas em pedaços pequenos; o objetivo principal é detectar o paralelismo nestas operações.
- **Comunicação**: Definir quais são as atividades de comunicação necessárias para que a resolução de um problema, dividida em pedaços, funcione de maneira apropriada sem erros.
- **Aglomeração**: A conclusão do particionamento e comunicação são combinadas de maneira a avaliar se a solução respeita requisitos de desempenho computacional e custos de implementação.
- **Mapeamento**: A tarefas são atribuídas, estática ou dinamicamente, às unidades de processamento de uma maneira a maximizar o uso de recursos computacionais e minimizar atividades de comunicação.

Em um cenário ideal, espera-se que uma aplicação paralela criada no âmbito do processo metodológico PCAM seja capaz de explorar de maneira eficiente uma quantidade indeterminada de unidades de processamento. Mesmo assim, é comum aplicar as quatro etapas tendo em mente uma determinada configuração computacional. Criar uma aplicação com portabilidade de desempenho é uma tarefa bastante difícil pois envolve utilizar algoritmos adaptativos em função da execução e do ambiente. Este tópico ainda é objeto de investigação.

Este texto está organizado da seguinte forma. A Seção 2.2 apresenta tópicos relacionados a fase de particionamento, tais como particionamento de dados e funcional. A Seção 2.3 apresenta tópicos ligados a forma como a comunicação deve ser projetada, se preocupando com pontos como limitação de canais e distribuição de operações de comunicação. A Seção 2.4 apresenta questões sobre granularidade de tarefas e técnicas para tentar reduzir a comunicação. A Seção 2.5 apresenta enfim a quarta etapa do processo PCAM com tópicos relacionados ao mapeamento das tarefas agrupadas para as unidades de processamento. A Seção 2.6 traz um estudo de caso com a equação de transferência de calor, onde cada uma das fases do processo PCAM é discutido. Enfim, a Seção 2.7 conclui este texto com um sumário do que foi descrito e ponteiros para aprofundar os conceitos apresentados.

2.2. Particionamento

A fase de particionamento no projeto de uma aplicação paralela envolve a descoberta de oportunidades para execução paralela. O objetivo principal envolve em

definir o maior número possível de pequenas tarefas. Com isso, procura-se estabelecer qual a unidade de trabalho (conjunto de operações sequenciais) de menor tamanho possível para o problema que está sendo quebrado em pedaços. Como consequência, esse grão pequeno permitirá então uma maior flexibilidade para a criação de algoritmos paralelos com diferentes características.

Um tamanho de tarefa demasiadamente pequeno pode incutir em uma perda de desempenho no que diz respeito às comunicações e ao gerenciamento das tarefas pequenas. Isso leva naturalmente a uma junção das operações de várias tarefas pequenas, efetivamente mudando a *granularidade* das tarefas. No âmbito do modelo PCAM, esta reflexão é relegada para mais tarde, na fase de aglomeração. Um outro aspecto nesta fase de particionamento é evitar a replicação de dados e de operações. Embora importante, este tipo de requerimento pode ser revisitado nas fases de projeto seguintes com o objetivo de reduzir a quantidade de comunicações, por exemplo.

Existem dois tipos principais de particionamento: de dados e de operações. A primeira técnica, mais comum e com um enfoque nos dados do problema, é conhecida por **decomposição de domínio**. Nela, o enfoque do particionamento se dá exclusivamente sobre o conjunto de dados, que é então dividido em pedaços pequenos. Todas as operações que são realizadas sobre os dados são secundárias e não afetam o processo de particionamento de dados. A segunda técnica, menos comum, tem um enfoque nos tipos de operações (instruções) que devem ser computadas pela aplicação paralela, sendo conhecida por **decomposição funcional**. Neste caso, o projetista deve identificar partes no futuro código da aplicação que são funcionalmente independentes, prevendo sua execução de maneira concorrente. Ainda que idealmente o processo de particionamento possa se preocupar com a divisão dos *dados* e das *operações* conjuntamente, é comum adotar um ou outro tipo de decomposição de maneira independente. Abaixo se fornece detalhes específicos para cada uma dessas técnicas.

Decomposição de domínio (particionamento de dados)

A decomposição do domínio tem por objetivo quebrar o problema em pedaços suficientemente pequenos. Por simplicidade, esse processo é conduzido de forma a obter pedaços que sejam também de tamanhos idênticos, de forma a facilitar as etapas seguintes do método PCAM. Cada pedaço terá portanto os dados, resultante da partição pelo método, e as operações associadas. É importante quantificar o custo destas operações de forma que elas sejam consideradas, ainda que de maneira secundária, na definição do tamanho da partição de dados.

A Figura 2.2 demonstra um exemplo de decomposição de um problema tridimensional que já foi discretizado conforme ilustração na esquerda da figura. Esta discretização é representada pelos pontos do espaço tridimensional, com cinco coordenadas no eixo `x`, e quatro coordenadas nos eixos `y` e `z`. O problema foi portanto discretizado em 80 pontos. Esta discretização pode então ser particionada em (1D) uma dimensão, com planos ao longo do eixo `z`, ou através de (2D) colunas ao longo do eixo `y`, ou (3D) através de uma partição tridimensional que

envolve apenas um ponto. É esta última opção que permite a maior flexibilidade pois o tamanho da partição engloba um único ponto do domínio discretizado.

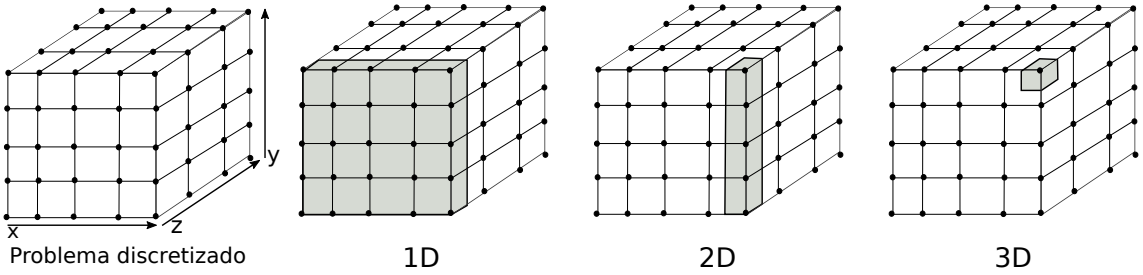

Figura 2.2. Três tipos de decomposição de domínio para um problema tridimensional (à esquerda), com uma (1D), duas (2D) e três (3D) dimensões. O tamanho do dado na abordagem tridimensional é o menor possível (representado pelo conjunto de pontos na grade), pois envolve apenas um ponto na grade.

Decomposição funcional (particionamento das operações)

A decomposição funcional representa uma maneira bastante alternativa para dividir o problema. Ao invés do foco ser nos dados do problema, divide-se as operações que podem ser executadas de maneira concorrente. Nos cenários onde a decomposição funcional permite uma execução paralela eficiente sobre dados disjuntos, pode-se concluir que o processo de particionamento está concluído. Pelo contrário, os cenários onde os mesmos dados são acessados por operações identificadas como paralelas resultantes da decomposição funcional implicam inevitavelmente em comunicações. Caso o conjunto de dados compartilhado pelas partições funcionais sejam demasiado grande, isso indica que é preferível uma abordagem via decomposição de domínio tradicional.

Um exemplo de particionamento funcional é a simulação computacional de um modelo climático. Esse modelo pode ser visto como um conjunto de equações independentes que resolvem parte físicas da simulação, tais como a atmosfera, a hidrologia, a superfície da terra e do oceano. Cada uma dessas partes operam sobre um conjunto de dados independentes, mas as bordas – o contato entre o oceano e a terra firma – exigem comunicação extra. Nesse contexto, uma decomposição funcional pode ser aplicada no alto nível, enquanto que cada parte do modelo possa operar internamente com o outro tipo de decomposição.

– Verificações Habituais sobre Particionamento –

- □ A partição (e o código que a cria) contém mais tarefas que a quantidade de núcleos de processamento em pelo menos uma ordem de magnitude.
- □ A partição evita cálculo redundante e é capaz de ser mantida em memória principal (sem uso de recursos de disco, por exemplo).
- □ As tarefas são de tamanho compatível, ou seja, o custo computacional das tarefas é parecido (idealmente idêntico).

- □ O aumento do tamanho do problema aumenta a quantidade de tarefas e não o tamanho delas.

- □ São verificadas e comparadas várias alternativas de particionamento possíveis, através do código que cria as partições.

2.3. Comunicação

Em um programa paralelo, é comum que as tarefas necessitem trocar informações para realizar suas operações. Em PCAM, a fase de comunicação envolve justamente o projeto destas atividades de troca de dados. Nos cenários onde inexiste a necessidade de comunicação, os problemas são chamados de *trivialmente paralelizáveis*, pois basta realizar o particionamento e as fases de aglomeração e mapeamento de PCAM. O enfoque deste texto é portanto naquelas aplicações mais complexas que incluem comunicações entre as tarefas.

Uma das principais preocupações com as atividades de comunicação é que elas possam acontecer da maneira mais concorrente possível. Esse estado ideal pode ser atingido de diferentes formas, através de variados padrões de comunicação. Uma classificação destes padrões pode seguir os seguintes eixos aproximadamente ortogonais: comunicação local ou global, estrutura ou não, estática ou dinâmica, e síncrona ou não. Estes eixos são detalhados abaixo.

2.3.1. Local e Global

Uma comunicação local é obtida quando a operação de computação de uma determinada tarefa (um ponto no domínio de problema discretizado) necessita dados de um pequeno número de tarefas (pontos) vizinhas. A operação é conhecida por *stencil*, podendo ser configurada para um ambiente de variadas dimensões, de acordo com o domínio do problema. Por exemplo, na Figura 2.2 a quantidade de vizinhos imediatos no domínio tridimensional é de seis, quatro de cada lado, um acima e outro abaixo. Neste caso, seis comunicações são necessárias antes da operação de cálculo. Quando o particionamento é bidimensional, como representado na ilustração 2D da Figura 2.3, a comunicação de todos os vizinhos pode ser necessária para um passo de simulação. Neste caso, existem oito operações de comunicações necessárias antes de efetuar as operações de cálculo da tarefa central. A localidade das comunicações podem variar bastante em função da complexidade da aplicação. Alguns cálculos podem exigir, por exemplo, dados de vizinhos de segunda ordem, conforme a ilustração 3D da Figura 2.3.

Uma operação de comunicação global pode envolver muitas tarefas, potencialmente todas aquelas que participam da aplicação paralela. Operações frequentemente reconhecidas como globais são aquelas de difusão massiva (*broadcast*) onde uma tarefa envia um dado para todas as demais, ou uma tarefa de redução, onde os dados de todas as tarefas são reduzidos por intermédio de um operador binário até uma única tarefa. Uma comunicação global pode ser implementada através de um algoritmo mestre-trabalhador. Neste caso, uma determinada tarefa fica responsável por recepcionar os dados de todas as outras que

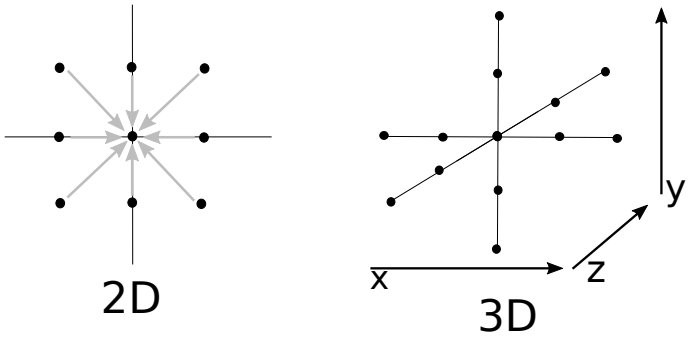

Figura 2.3. Comunicação local em particionamentos 2D (esquerda) e 3D (direita).

participam da computação. Este algoritmo tem duas características que tornam-no incapaz de atingir uma boa escalabilidade. Ele é centralizado e sequencial, uma vez que a tarefa mestre recebe as informações em uma determinada ordem. Uma forma mais eficiente de obter a mesma funcionalidade deste algoritmo é empregar uma árvore N-ária para difundir o dado mais rapidamente. Neste caso, as comunicações nas folhas e nós intermediários da árvore podem acontecer simultaneamente. O resultado é uma estrutura de comunicação regular na qual cada tarefa se comunica com poucos vizinhos próximos.

2.3.2. Estrutura ou não estruturada

As situações apresentadas até o momento são exemplos de uma estrutura de comunicação estática, ondes as tarefas tem vizinhos claramente definidos e imutáveis em função do particionamento estabelecido na fase anterior. Neste contexto as comunicações são frequentemente fixas, ditas estruturadas, e não evoluem ao longo da execução da aplicação paralela. Em outros casos, a grade de discretização pode seguir padrões mais complexos, conhecidos como não estruturados e irregulares. Por exemplo, a grade de um objeto irregular tal como o pulmão de uma pessoa pode ser melhor modelado por uma grade composta por formas simples, tais como triângulos, tetraedros, etc. Estas grades podem ser descritas por grafos, onde os vértices representam as tarefas e as arestas representam comunicação. Nestes casos, as atividades de comunicação entre as tarefas são igualmente mais complexas, envolvendo por vezes mais vizinhos em determinadas regiões da discretização quando comparada com outras.

2.3.3. Estática ou dinâmica

Grades de discretização podem ser regulares (veja exemplo na Figura 2.2) ou irregulares (exemplo na Figura 2.4), tais como um objeto modelado por formas como triângulos, etc. Uma diferença fundamental que pode afetar as demais fases do projeto PCAM é se tais grades são estáticas ou dinâmicas. No caso de grades estáticas, a discretização é fixa desde a concepção na fase de particionamento do projeto até a execução do código. No caso de grades dinâmicas, a discretização pode mudar em função da execução da aplicação paralela. Programas complexos podem aumentar a fidelidade de simulação em torno de objetos móveis em uma

simulação.

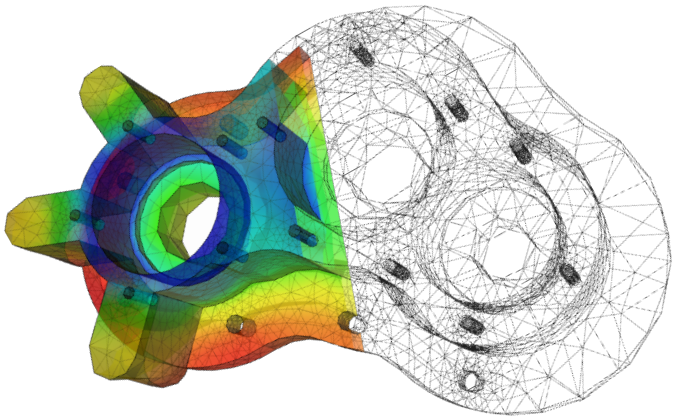

Figura 2.4. Grade de particionamento irregular tridimensional, representada por um grafo onde cada nó é uma tarefa e cada aresta é uma operação de comunicação. (Artigo da Wikipedia em Alemão sobre Elementos Finitos).

Padrões irregulares de comunicação normalmente não afetam as fases de particionamento. No exemplo da Figura 2.4, pode-se observar que algumas regiões tem uma intensidade de tarefas maior (pela proximidade física) que outras. O particionamento de um grafo como este pode implicar que cada nó de um grafo se torne uma tarefa e suas arestas se tornem comunicações. No entanto, uma grade irregular pode complicar bastante a condução das fases de aglomeração e mapeamento. Por exemplo, ainda que a grade seja estática, a fase de aglomeração pode ser complicada pois envolve antecipar o custo computacional das tarefas e a quantidade de dados da comunicação, de forma a criar grupos de tarefas que tenham por um lado um peso similar e que minimizem as comunicações. No caso da grade ser dinâmica, os algoritmos que realizam a aglomeração podem ser necessários durante a execução do programa incutindo em sobrecargas que devem ser pesados contra os benefícios trazidos por um melhor agrupamento de tarefas.

2.3.4. Síncrona ou Assíncrona

Todos os conceitos sobre esta fase consideram comunicação síncrona, onde as duas tarefas envolvidas na troca de dados estão cientes quando a operação acontece. Na comunicação assíncrona, por outro lado, as tarefas que possuem os dados, e que são responsáveis pelo envio, não estão cientes do momento quando as tarefas receptoras precisarão efetivamente dos dados da comunicação. Sendo assim, as tarefas receptoras devem registrar a necessidade de um dado que eventualmente será satisfeito, de maneira assíncrona, pela tarefa responsável por enviar o dado.

A grande vantagem de comunicações assíncronas é que elas podem ser utilizadas para esconder as comunicações. Em *middlewares* sofisticados de comunicação, a troca de dados de maneira assíncrona pode ser implementada de uma forma que a aplicação não fica bloqueada em nenhum momento esperando a conclusão do envio/recepção. Sendo assim, a aplicação paralela pode antecipar o registro da necessidade de um dado de forma que quando ele for necessário

esse dado já tenha sido recebido. Esse conceito serve também do lado do envio, onde o gerador do dado registra o envio para quem precisa do dado assim que ele for gerado, potencializando a comunicação assíncrona.

– Verificações Habituais sobre Comunicação –

- ☐ As tarefas realizam uma quantidade similar de operações de comunicação.
- ☐ Cada tarefa se comunicam com um número pequeno de vizinhos.
- ☐ As operações de comunicação independentes podem ser concorrentes.
- ☐ O cálculo computacional das tarefas pode acontecer de maneira concorrente.

2.4. Aglomeração

As fases precedentes da metodologia PCAM permitem o particionamento e a definição das comunicações necessárias para a resolução paralela de um problema. O resultado destas fases é um algoritmo abstrato que contém potencialmente muitas tarefas, tendo em vista que o objetivo é identificar a menor operação possível que possa ser executada concorrentemente com as demais. Esse algoritmo abstrato, distante da realidade, é normalmente ruim por ter tarefas demais. O gerenciamento dessa quantidade enorme de tarefas é prejudicial para o desempenho. A fase de *aglomeração* tem por objetivo tornar o algoritmo abstrato das fases precedentes em algo mais realista, de acordo com os limites impostos pela configuração da plataforma de execução alvo. O objetivo principal é obter um programa eficiente nesta plataforma. Para atingir tal objetivo, é importante avaliar, analítica ou experimentalmente, (a) o benefício da aglomeração através do seu impacto em diretivas de comunicação e no tempo de execução, (b) o benefício de se replicar dados e operações, e (c) a necessidade de uma redução drástica para uma tarefa por unidade de processamento (seguindo o modelo SPMD: *Single Problem Multiple Data*).

Outro ponto relevante na fase de aglomeração é que a decisão sobre a granularidade das tarefas não deve ser fixa. O programa deve ser capaz de permitir uma certa adaptabilidade tendo em vista a evolução dos computadores (avanços nas unidades de processamento e na tecnologia de interconexão) que podem tornar obsoleta uma determinada decisão de aglomeração. Abaixo são apresentados tópicos relacionados a granularidade de tarefas, a relação entre superfície e volume no particionamento, e uma discussão sobre replicação de cálculo e formas de evitar a comunicação.

2.4.1. Granularidade de tarefas

Na fase de particionamento o objetivo é baseado na premissa de quanto mais tarefas melhor, ou seja, deve-se encontrar o menor conjunto de operações que possa ser executada sequencialmente. No entanto, observa-se que esse tipo de particionamento fino pode levar a elevados custos de comunicação que prejudicam o

desempenho da aplicação, tendo em vista que a unidade de processamento para de executar código útil para se ocupar de envios e recepção de dados. A aglomeração permite então tornar as tarefas maiores, e na medida que isso ocorre, pode haver um efeito positivo da redução do custos de comunicação.

A Figura 2.5 mostra exemplos de aglomeração de tarefas a partir do particionamento original com uma tarefa por ponto, ilustrada na esquerda da figura. As quatro opções de aglomeração, ilustradas no centro esquerda, ilustram uma aglomeração de pontos horizontal e vertical (na parte superior da figura), e dois planos possíveis (na parte inferior). Enfim, uma aglomeração mais efetiva é a aquela tridimensional, como na ilustração do centro direita onde o bloco aglomera pontos em todas as três dimensões, reduzindo o perímetro do bloco, ou seja, as bordas que exigem comunicação com as tarefas vizinhas.

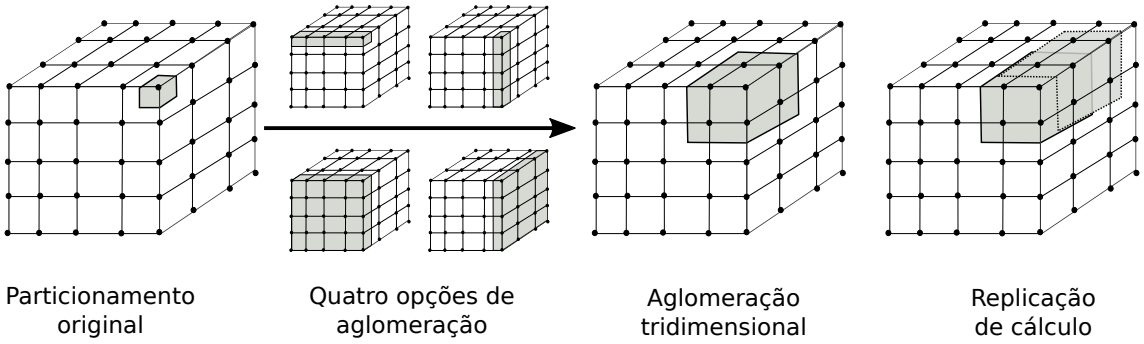

Figura 2.5. Exemplos de aglomeração de tarefas a partir do particionamento original (esquerda), e replicação de cálculo através da sobreposição de dois conjuntos aglomerados na decomposição de domínio (direita).

A aglomeração serve sobretudo para escolher o nível certo de concorrência que extrai o máximo de desempenho da plataforma de execução. Isso envolve então a redução das comunicações mas também pode ser influenciada por outras estratégias. Por exemplo, pode-se agregar dados a serem comunicados de forma que o envio seja feito com uma única operação ao invés de múltiplos envios. Isso permite evitar a latência da rede de interconexão. Pode-se também encontrar a menor quantidade de tarefas que maximize o desempenho, tendo em vista que que o excesso de tarefas pode ser penalizado pelo custo de gerenciamento da criação e manutenção das mesmas.

2.4.2. Relação entre superfície e volume

A fase de aglomeração traz o benefício de poder reduzir a quantidade de comunicação, como ilustrado no exemplo da Figura 2.6. Do lado esquerdo nós temos um particionamento fino de 8×8, com um total de 64 tarefas (representadas pelos círculos). Considerando que cada tarefa deve enviar 1 dado para cada um dos quatro vizinhos imediatos (conforme ilustrado nas tarefas em tons de cinza mais escuro), nós temos um total de 256 operações de comunicação cada uma com 1 dado. Ao realizar uma aglomeração bidimensional com um fator de 16 para 1, obtemos uma grade como aquela ilustrada na direita da figura, com quatro tarefas. Nesta configuração, são reduzidas não somente a quantidade de operações

de comunicação para apenas 16 (pois cada uma das quatro tarefas se comunica com quatro vizinhos), mas também a quantidade de dados comunicados, pois envolve apenas o perímetro dos dados bidimensionais gerenciados por uma tarefa (os quadrados em tons cinza escuro na figura). Essa relação entre superfície e volume, ilustrada na figura através de um exemplo 2D, reduz portanto as necessidades de comunicação.

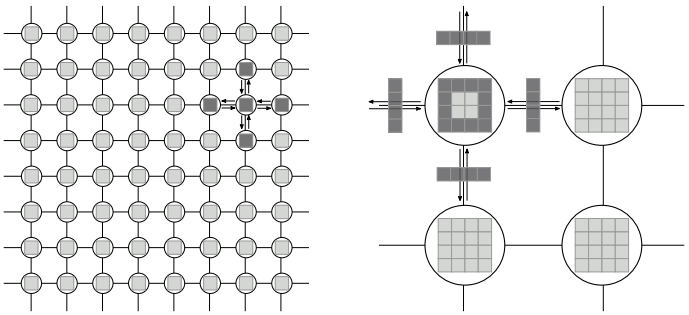

Figura 2.6. Efeito do aumento da granularidade nos custos de comunicação: em uma grade 8×8 (esquerda), o custo total de comunicação é de 256 mensagens (cada tarefa realiza 4 comunicação) cada uma com 1 dado (256 dados no total); em uma grade 2×2 com 4 tarefas (direita), apenas 16 comunicações são necessárias, cada uma com 4 dados para um total de 64 dados.

O efeito da fase de aglomeração em grades não-estruturadas, como aquela exemplificada na Figura 2.4, são mais complexas de serem realizadas. Existem técnicas especializadas que tentam equalizar o peso das partições ao mesmo tempo que reduzem as bordas de comunicação. Essas técnicas permitem portanto o balanceamento da carga computacional e são rapidamente apresentadas na Seção 2.5 sobre Mapeamento.

2.4.3. Replicando computação

Em determinado cenários pode ser benéfico replicar cálculo computacional para se obter vantagens como a redução da quantidade/volume de comunicações necessárias. Um exemplo de replicação de cálculo acontece quando as partições de dados da decomposição de domínio tem uma área/volume que se intersecta. A ilustração na direita da Figura 2.5 traz um exemplo onde duas partições compartilham pontos do domínio no centro do cubo na dimensão `z` (profundidade).

2.4.4. Evitando comunicação

A aglomeração certamente será benéfica se o entendimento do projetista estabelecer que algumas tarefas não podem ser executadas de maneira concorrente. O exemplo da esquerda na Figura 2.7 representa uma operação de redução de dados através de um operador de soma. Conforme sobe-se na estrutura hierárquica (a partir dos nós folha na parte inferior da ilustração), realiza-se a soma dos valores filhos até chegarmos com a soma de todos os valores na raiz da árvore (nó superior). Percebe-se que o paralelismo é natural em um determinado nível da árvore, por exemplo, quando se deve obter os dados das folhas para realizar a primeira soma. No entanto, podemos ver que as tarefas da folha são

incapaz de se executaram ao mesmo tempo que as tarefas intermediárias no primeiro nuvem pois existe uma clara dependência de dados no grafo. Isso indica uma oportunidade de aglomeração entre os nós intermediários de primeiro nível e seus filhos (que são folhas). A ilustração na direita exemplifica como pode-se tirar proveito desta situação através do encadeamento de operações de redução. Neste caso, duas operações independentes de redução podem ser aglomeradas, estabelecendo uma forma de *pipeline* dentro da tarefa aglomerada.

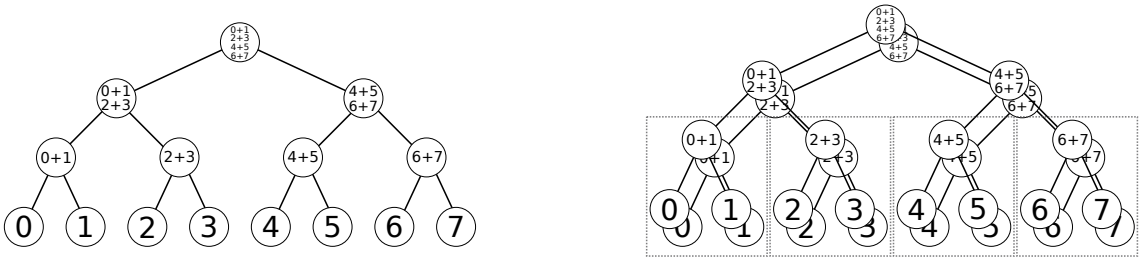

Figura 2.7. Algoritmo em árvore binária para uma operação de redução (esquerda) e exemplo do encadeamento de operações de redução e as vantagens de realizar aglomeração de níveis da árvore (direita).

– Verificações Habituais sobre Aglomeração –

- ☐ A aglomeração reduz o custo das comunicações ao aumentar a localidade.
- ☐ O benefício da replicação de operações de cálculo justifica seu emprego.
- ☐ O benefício da replicação de dados não afeta a escalabilidade do algoritmo.
- ☐ A aglomeração gera tarefas similares em custos de cálculo e comunicação.
- ☐ A quantidade de tarefas aumenta com o tamanho do problema.
- ☐ Há portabilidade de desempenho se a aglomeração limita a concorrência.
- ☐ O número de tarefas pode ser reduzido sem afetar o balanceamento de carga.

2.5. Mapeamento

O quarto estágio da metodologia PCAM, chamado mapeamento, consiste em definir onde cada tarefa será executada. O mapeamento em si em um problema difícil pois precisa ser explícito em supercomputadores de alto desempenho. Inexiste um método automático para realizar o mapeamento, embora soluções simples possam ser aplicadas, sem que o desempenho seja o melhor possível. Os requisitos fundamentais na atividade explícita de mapeamento envolve (a) colocar tarefas concorrentes em unidades de processamento diferentes, de forma que tais tarefas sejam de fato executadas em paralelo e (b) alocar tarefas que se comunicam frequentemente em locais próximos na topologia de interconexão, tanto

física quanto lógica. Estas duas condições podem entrar em conflito. Por exemplo, se considerarmos apenas a localidade podemos ser levados a colocar todas as tarefas em uma unidade de processamento.

Em alguns casos a fase de mapeamento é simples. Por exemplo, em algoritmos paralelos que usam decomposição de domínio com tarefas cujo custo computacional é estático (ao longo do tempo) e homogêneo (entre as partições) são apropriados para mapear em supercomputadores com capacidade homogênea (todos as unidades de processamento são idênticas). Nestes casos, pode-se inclusive aglomerar as tarefas de maneira que tenhamos apenas uma tarefa por processador, reduzindo ao máximo a necessidade de comunicação.

Por outro lado, a fase de mapeamento se torna mais complexa quando as partições tem custos computacionais diferentes, ainda que estes custos sejam estáticos ao longo do tempo. Nestes cenários, algoritmos de *balanceamento de carga* podem ser úteis para equilibrar os custos entre os recursos computacionais. Métodos descentralizados de balanceamento de carga tem mais chance de se adaptar a evolução dos supercomputadores, especialmente no quesito de escalabilidade, pois não há uma única entidade centralizada. O cenário mais complexo para a fase de mapeamento ocorre quando a carga computacional é heterogênea tanto entre as partições quanto ao longo do tempo. Neste caso, deve-se aplicar preferencialmente algoritmos de *balanceamento de carga dinâmicos*, capazes de monitorar a evolução da carga computacional ao longo do tempo. Algoritmos que exigem apenas um conhecimento local são preferíveis pois não requerem nenhum tipo de conhecimento global (possível com comunicações coletivas globais entre todas as tarefas).

Enfim, os algoritmos oriundos de decomposição funcional tem uma abordagem diferente de mapeamento. Eles podem ser mapeados preferencialmente por algoritmos de escalonamento de tarefas, antecipando tempo de ociosidade em processadores.

2.5.1. Balanceamento de carga

Algoritmos de balanceamento de carga são também conhecidos por algoritmos de particionamento. Eles tem por objetivo aglomerar tarefas finas (oriundas da fase de particionamento) de uma partição inicial até encontrar uma tarefa cujo tamanho seja apropriado para uma determinada plataforma de execução. Existem quatro técnicas principais de balanceamento de carga: métodos baseados em bisseção recursiva, algoritmos locais, métodos probabilistas e mapeamento cíclicos.

Os métodos baseados em *bisseção recursiva* particionam o domínio do problema de maneira iterativa, com informações globais, sempre levando-se em conta o custo de um subdomínio e a minimização da comunicação entre as partições. Esses métodos se enquadram na classe de algoritmos de divisão e conquista e um exemplo é o algoritmo de Barnes-Hut [1]. Existem várias variantes dos métodos de bisseção recursiva. A forma mais simples, que não considera o custo e a quantidade das comunicações, consista em realizar a bisseção recursiva unicamente baseada nas coordenadas do domínio: sempre se divide a coordenada mais larga.

Uma segunda variante do método de bisseção se chama de método desbalanceado pois tem um enfoque unicamente no controle das comunicações, reduzindo o perímetro das partições. Enfim, uma terceira variante mais sofisticada e mais geral é a bisseção recursiva de grafo, útil para grades não-estruturadas (veja Figura 2.4). Esta variante usa a informação de conectividade do grafo para reduzir o número de arestas que cruzam a fronteira entre dois subdomínios.

Uma técnica alternativa com menor intrusão consiste nos algoritmos de *balanceamento de carga locais*. Eles são relativamente baratos pois necessitam apenas de informações da tarefa em questão e de seus vizinhos. Isso possibilita também uma execução paralela, ou seja, todas as tarefas podem executar o algoritmo local simultaneamente. No entanto, a falta de coordenação global leva em geral a um particionamento pior daquele obtido por particionadores globais.

O terceiro tipo de método de balanceamento de carga consiste em *métodos probabilistas*. Com um custo baixo de execução e uma boa escalabilidade, ao mesmo tempo que ignora completamente o custo e quantidade de comunicações, algoritmos probabilistas alocam as tarefas nos recursos computacionais de maneira aleatória. Essa abordagem funciona melhor quando há muitas tarefas, pois as chances são maiores de fazer com que os recursos computacionais recebam carga de trabalho similar.

Enfim, os *mapeamentos cíclicos* são uma quarta forma de realizar o mapeamento da carga nos recursos computacionais. Baseado em um particionamento já definido na primeira fase do método PCAM, a técnica distribui aos recursos, de maneira cíclica, as partições. Parte-se do princípio que existem bastante tarefas, que os custos com comunicação são baixos através de uma localidade de operações e dados reduzidas. Um mapeamento cíclico pode ser realizado utilizando como entrada os blocos de tarefas, criados na fase de aglomeração.

2.5.2. Escalonamento de tarefas

Os algoritmos de escalonamento de tarefas podem ser usados em situações com requisitos de localidade fracos. Em geral, eles consideram as tarefas como um conjunto de problemas que devem ser resolvidos, sendo colocados em uma "piscina" de problemas. Uma heurística de escalonamento deve então decidir, durante a execução e de maneira dinâmica, em qual recurso computacional um determinado problema, recuperado da piscina, será alocado. O desenvolvimento de uma heurística que englobe de um lado a necessidade de reduzir os custos de comunicação e de outro o conhecimento global do sistema (para efetuar um bom balanceamento de carga) é o principal desafio. Embora heurísticas centralizadas tem um bom conhecimento da plataforma, em geral eles não são escaláveis para centenas de unidades de processamento. Para mitigar esse problema, existem heurísticas que criam uma estrutura hierárquica de gerenciadores, permitindo uma alternativa mais escalável com uma visão semi-global do estado da plataforma. Enfim, no outro extremo existem heurísticas totalmente descentralizadas: cada processador mantém uma "piscina" de tarefas e trabalhadores podem requisitar mais tarefas quando se tornam ociosos. Heurísticas probabilistas, poten-

cialmente hierárquicas de acordo com a topologia da plataforma computacional, e associadas a roubo de tarefas se enquadram nesta classe de algoritmos.

– Verificações Habituais sobre Mapeamento –

- ☐ Ao considerar uma abordagem tradicional SPMD (com uma única tarefa por unidade de processamento), realiza-se uma comparação com métodos baseados na criação dinâmica de tarefas, com heurísticas de escalonamento.
- ☐ Pelo contrário, ao considerar uma abordagem com criação dinâmica de tarefas, com heurísticas de escalonamento, compara-se com a abordagem tradicional SPMD.
- ☐ Ao usar um esquema de balanceamento de carga centralizado, verifica-se os limites de escalabilidade da abordagem.
- ☐ Foram avaliadas diferentes heurísticas de escalonamento ao empregar um algoritmo de balanceamento dinâmico. Os custos das heurística são conhecidos.
- ☐ Ao considerar técnicas simplistas para o mapeamento, tais como métodos probabilistas ou cíclicos, há uma quantidade suficiente de tarefas para assegurar um bom balanceamento.

2.6. Estudo de caso com a equação de Transferência de Calor

Um problema clássico de simulação computacional é a equação de calor para um determinado ambiente ou material. A equação de calor para um espaço bidimensional (x, y) é dada pela Equação 1.

$$\frac{\delta u}{\delta t} - \alpha\left(\frac{\delta^2 u}{\delta x^2} + \frac{\delta^2 u}{\delta y^2}\right) = 0 \tag{1}$$

Onde u é a temperatura, t é o tempo, e α é o coeficiente de difusão térmica. Esta equação pode ser resolvida pelo método das diferenças finitas, onde o espaço bidimensional é discretizado em uma grade com células. Cada célula é a menor unidade possível de representação das propriedades (temperatura) para um momento específico do tempo. Para simplificar nosso caso, assumimos que todas as células tem tamanho contante de Δx e Δy e que $\Delta x = \Delta y$, como mostra a Figura 2.8, além disso, assumimos que o tempo sempre avança de forma constante com o valor de Δt.

Esta discretização gera um sistema de equações lineares para ser resolvido. Um dos métodos possíveis para solucionar este sistema é o método de Jacobi. Este espaço discretizado da equação de calor, considerando várias simplificações, pode ser resolvida aplicando a Equação 2. Deve-se calcular, para cada célula, as propriedades físicas no próximo instante tempo. Onde $u_{i,j}^{k+1}$ representa a temperatura da iteração $k+1$ na coordenada i, j.

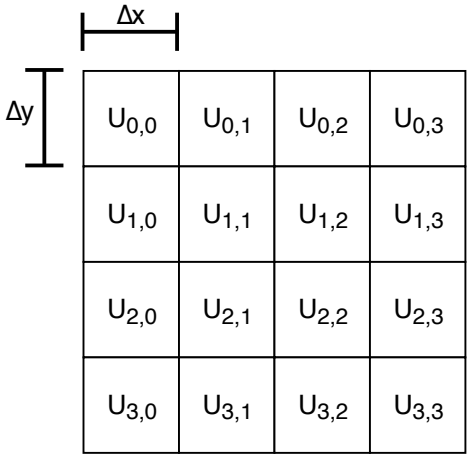

Figura 2.8. Discretização de um espaço 2D de tamanho 4×4 em células de tamanho Δx e Δy, onde cada célula tem a propriedade de temperatura u.

$$u_{i,j}^{k+1} = \frac{u_{i+1,j}^{k} + u_{i-1,j}^{k} + u_{i,j+1}^{k} + u_{i,j-1}^{k}}{4} \tag{2}$$

Desta forma, o problema computacional está em calcular as temperaturas de cada célula após um número de iterações K, dado um estado inicial do espaço, uma grade $nx \times ny$ de células com suas respectivas temperaturas iniciais $u_{i,j}^{0}$. Basicamente deve-se aplicar a equação 2 em todas as células K vezes, onde para cada célula devemos acessar os vizinhos dela na iteração anterior. Como existe uma dependência de dados entre as iterações, as células $u_{i+1,j}^{k}$, $u_{i-1,j}^{k}$, $u_{i,j+1}^{k}$ e $u_{i,j-1}^{k}$ sempre deveram ser calculadas antes que $u_{i,j}^{k+1}$. Além disso, deve-se determinar como as bordas do problema devem se comportar. Para simplificação, considera-se que as células da borda do espaço discretizado tem um comportamento especial, onde são iguais ao valor da temperatura da sua célula vizinha mais central. Um algoritmo sequencial que resolve este problema é dado na Listagem 2.1, a seguir.

```
for k from 1 to K:
   solve_borders();
   for j from 1 to ny-1:
     for i from 1 to nx-1:
        u[k][j][i] = (u[k-1][j][i+1] + u[k-1][j][i-1] +
                      u[k-1][j+1][i] + u[k-1][j-1][i]) / 4
```

Listagem 2.1. Algoritmo sequencial que resolve o problema da Equação 2.

Uma versão paralela deste problema pode ser concebida aplicando os conceitos anteriormente discutidos do PCAM. As próximas subseções discutem passo a passo algumas das várias opções de particionamento, comunicação, aglomeração e mapeamento possíveis neste problema.

Particionamento

Primeiramente, é necessário avaliar quais operações podem ser realizados em paralelo. Os dados do problema são uma matriz $nx \times ny$ para cada instante de

tempo k. Umas das primeiras e mais simples abordagens é calcular cada tempo k isoladamente, onde a equação pode ser resolvida em cada célula separadamente e paralelamente. Desta forma, apenas duas matrizes de tamanho $nx \times ny$ seriam necessárias, uma contendo um instante de tempo já calculado, e uma aonde seria colocados os resultados. Assim, o menor particionamento possível seria calcular cada célula do instante $k+1$ baseado na mesma célula e suas vizinhas do instante k.

Comunicação

A comunicação está diretamente ligada com o particionamento de dados. No problema em questão, existe uma dependência entre as células vizinhas, caracterizando uma comunicação local. Desta forma, utilizando a atual discretização do problema, a comunicação seria estruturada e estática, podendo haver comunicação assíncrona, já que diferentes recursos podem calcular células que todos os dados estão presentes enquanto enviam dados de células que outros recursos vão precisar.

Aglomeração

Para aglomerar estas células, pode-se escolher um conjunto de células contíguas. Maneiras tradicionais são apresentados pela Figura 2.9, aglomerando as células em conjunto de linhas (a), em conjunto de colunas (b) e em blocos (c).

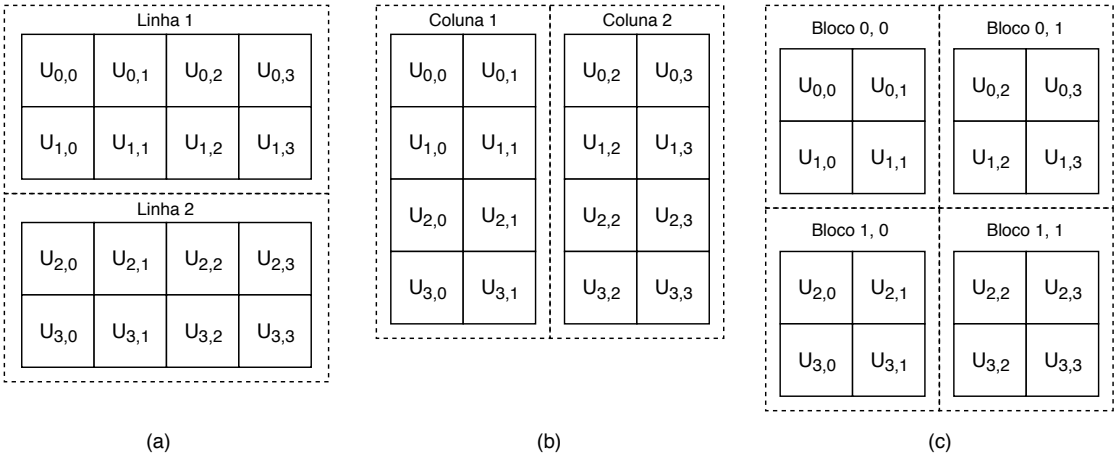

Figura 2.9. Possíveis aglomerações no problema bidimensional da equação de temperatura. (a) Por linhas. (b) Por Colunas. (c) Por blocos.

Em cada um destes tipos de aglomeração, deve-se entender o impacto tanto na programação quanto na execução do programa. Uma aglomeração em linhas, por exemplo, é implementada mais facilmente se a matriz é alocada de forma contígua na memória. Entretanto, tal tipo de aglomeração, por linhas ou por colunas, faz com que a borda entre grupos poderá ficar muito grande, aumentado a comunicação. Por exemplo, considere uma matriz de tamanho 30×30, caso cada grupo deva ter no máximo 100 células, 10 linhas de 3×30 deverão ser criadas. Desta maneira, a borda entre as linhas seria de 30 células de cada lado, totalizando 60 células. Neste mesmo caso, utilizando a estratégia de blocos, po-

deriam ser utilizados blocos de 10×10 células, gerando um total de nove blocos. Resultando em 40 células de borda que deverão ser transferidas. Em um cenário ideal, os blocos retangulares devem ter o menor perímetro possível para minimizar as comunicações, ou seja, devem se assemelhar ao máximo a quadrados.

Dependendo dos recursos a serem utilizados para realizar o processamento, grupos de diferentes tamanhos poderão ser gerados. Por exemplo, considere a utilização de recursos heterogêneos que tem variações no poder de processamento. Considere que parte dos dados seriam computados em um recurso com maior poder computacional, uma GPU por exemplo, e o resto em uma CPU. A criação de aglomerações de tamanhos diferentes seria interessante para satisfazer os diferentes poderes de computação, movendo as maiores aglomerações para os recursos com maior poder computacional.

Mapeamento

Considerando a aglomeração discutida anteriormente, grupos de partições maiores poderiam ser estaticamente mapeados para recursos com maior poder computacional. Por exemplo, considere um único nó de processamento com uma CPU e uma GPU, o problema em questão pode ser portando para o processamento vetorial da GPU e o número de células que este recurso processaria seria maior que as CPUs. Um programador poderia calcular a razão de processamento e dividir estaticamente o problema, criando grupos de células de tamanhos diferentes para cada recurso ou simplesmente arbitrando quantos grupos deveriam ir para a GPU. Entretanto, mapeamos dinâmicos poderiam ser realizados considerando estas arquiteturas heterogêneas, a fim de tentar balancear as cargas entre os recursos de melhor maneira possível. Utilizando um *middleware* de balanceamento, como os *runtimes* de programação orientada a tarefas, o processamento dos grupos de células seria mapeado para os recursos dinamicamente em tempo de execução conforme os algoritmos de balanceamento decidirem.

2.7. Conclusão

Este minicurso trata os conceitos fundamentais para o projeto de aplicações paralelas. O minicurso tem estrutura similar ao capítulo 2 do livro [2], apresentando a metodologia PCAM com quatro fases: particionamento, comunicação, aglomeração e mapeamento). As verificações habituais listadas em cada uma destas fases (veja Seções específicas anteriores) é fundamental para uma correta exploração de todos os algoritmos possíveis para o paralelismo natural que pode ser extraído de um problema. Somente a avaliação das diferentes variantes de algoritmos paralelos, pelo menos no momento do projeto, permite bons resultados de desempenho na escolha do algoritmo mais apropriado para um problema.

Como previamente apresentado na introdução, o texto deste minicurso é uma versão compacta e adaptada para o público iniciante em processamento de alto desempenho. Sugere-se fortemente a leitura do livro do Foster para um detalhamento mais profundo deste tópico em especial, inclusive com outros estudos de caso, mas também do restante do conteúdo apresentado.

Referências

[1] Josh Barnes and Piet Hut. A hierarchical o (n log n) force-calculation algorithm. *nature*, 324(6096):446, 1986.

[2] Ian Foster. *Designing and Building Parallel Programs: Concepts and Tools for Parallel Software Engineering*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995.