Capítulo 2

Projeto de Aplicações Paralelas

Guilherme Galante
Universidade Estadual do Oeste do Paraná

Resumo

O objetivo deste minicurso é fornecer uma visão geral do processo de projeto de aplicações paralelas. São apresentadas duas abordagens: PCAM e Padrões de Projeto. Como se trata de um minicurso introdutório, espera-se que, ao final, o leitor tenha um entendimento inicial que sirva de base para continuar com seus estudos na área.

2.1. Introdução

Hoje em dia, todos os computadores são essencialmente paralelos. Ensinar e aprender programação paralela tornou-se cada vez mais importante devido à onipresença de processadores com algum grau de paralelismo em dispositivos portáteis, estações de trabalho e clusters de computação. Para utilizar totalmente os recursos de computação das arquiteturas de hardware atuais, é necessário que futuros cientistas e engenheiros da computação escrevam código altamente paralelizado. Além disso, o crescimento de áreas como aprendizagem de máquina e big data exige a adoção de processamento de alto desempenho como parte integrante de seu domínio de conhecimento. Adquirir habilidades de programação paralela é hoje em dia uma parte indispensável de muitos currículos de graduação e pós-graduação (Schmidt et al. 2018).

No entanto, o projeto de aplicações paralelas ainda é considerado uma ordem de magnitude mais difícil do que o projeto de algoritmos sequenciais e o desenvolvimento de programas sequenciais (Trobec et al. 2018). Projetar corretamente aplicações paralelas envolve conhecer bem o problema e como este pode ser trabalhado para se adaptar corretamente à arquitetura será adotada, seja um processador multicore, cluster ou GPU. Enfim, as escolhas na fase de projeto são fundamentais para se obter um bom desempenho e um uso eficiente dos recursos de processamento.

Nesse contexto, o objetivo deste minicurso é fornecer uma visão geral do processo de projeto de aplicações paralelas. São apresentadas duas metodologias: PCAM, proposta por (Foster 1995), e Padrões de Projeto, propostos por (Mattson et al. 2004). Como se

trata de um minicurso introdutório, espera-se que, ao final, o leitor tenha um entendimento inicial que sirva de base para continuar com seus estudos na área.

Este texto está organizado da seguinte forma. Seção 2.2 apresenta as etapas do desenvolvimento de aplicações paralelas. Seção 2.3 aborda a questão da detecção dos principais pontos de paralelismo (*hotspots*) em um programa paralelo. A Seção 2.4 apresenta a metodologia PCAM. A Seção 2.5 introduz alguns padrões de projeto para o desenvolvimento de aplicações paralelas. Por fim, a Seção 2.6 conclui o minicurso.

2.2. Do problema à solução paralela: etapas de desenvolvimento

De acordo com (Czarnul 2018), o desenvolvimento de código paralelo envolve quatro etapas:

- Formulação de um problema com a definição dos dados de entrada, operações necessárias e formato dos resultados de saída.

- Projeto da aplicação. Normalmente, uma aplicação sequencial pode ser paralelizada, ou seja, adaptada para executar mais rápido, dividindo cálculos/dados para executar em vários núcleos com comunicação/sincronização necessária, de modo que a saída correta seja produzida. Outra abordagem é produzir um algoritmo paralelo do zero, o que não é comum (ou trivial).

- Implementação do algoritmo. Nesse caso, uma ou mais interfaces de programação (API) podem ser usadas para codificar o algoritmo. Normalmente, uma API é usada de modo que sua implementação possa ser executada com eficiência em uma classe selecionada de hardware. Por exemplo, OpenMP para processadores multicore, MPI para clusters e CUDA para GPUs.

- Otimização de código. Esta etapa inclui a aplicação de técnicas de otimização no código que podem estar relacionadas a hardware específico. Tais técnicas podem incluir reorganização de dados, escalonamento, sobreposição de comunicações e cálculos, balanceamento de carga entre nós e núcleos de computação.

A paralelização de um código sequencial pode ser fácil, como é o caso, por exemplo, nos chamados problemas *trivialmente paralelizáveis*, caracterizados pelo esforço mínimo necessário para dividir a aplicação em uma coleção de tarefas que podem ser distribuídas entre um conjunto de componentes de computação para processamento em paralelo (Wilkinson and Allen 2005). Por outro lado, o processo é mais difícil quando o fluxo de controle é complexo, o algoritmo contém partes difíceis de paralelizar ou quando a proporção de cálculos para comunicação/sincronização é relativamente pequena. Por exemplo, em um cálculo direto dos números de Fibonacci:

```
f[0] = f[1] = 1;
for (i = 2; i <= n; i++)
     f[i] = f[i-1] + f[i-2];
```

essencialmente não há oportunidade para execução simultânea de instruções, considerando que a computação em uma iteração depende dos resultados de uma ou mais iterações anteriores (dependência de laço) (Pacheco 2011).

2.3. Identificando Hotspots

Um *hotspot* é uma seção de código que leva a maior parte do tempo de execução de um programa. Portanto, identificar os hotspots pode dar pistas de onde o esforço efetivo de paralelização deve ser concentrado. Seções do programa que representam pouco uso da CPU devem ser ignorados inicialmente. A maioria das aplicações científicos e técnicas, por exmeplo, geralmente realiza a maior parte de seu trabalho em alguns trechos bastante restritos. Geralmente laços de repetição e funções com uma alta porcentagem de contagem de instruções são identificados como hotspots.

A detecção de *hotspots* pode ser realizada usando várias ferramentas de análise de desempenho, que podem identificar as seções do código que estão consumindo a maior parte dos recursos computacionais ou do tempo de processamento.

2.3.1. Exemplo: Intel VTune Profiler

O Intel VTune Profiler[1] é uma ferramenta de análise de desempenho amplamente utilizada para detecção de hotspots em aplicações. O VTune é capaz de fornecer informações detalhadas sobre o desempenho da aplicação, permitindo que os desenvolvedores identifiquem áreas críticas do código que estão consumindo a maior parte dos recursos computacionais.

O VTune Profiler também fornece visualizações gráficas das informações coletadas, permitindo que os desenvolvedores compreendam facilmente os resultados e identifiquem os hotspots de maneira eficiente. As visualizações incluem gráficos de tempo de CPU, uso de memória e latência, bem como fluxogramas de chamada de função e tabelas de eventos.

Na Figura 2.1 apresenta-se um exemplo de análise de uma aplicação de multiplicação de matrizes. Observa-se que o tempo total de CPU para a aplicação é igual a 150.993 segundos. A seção *Top Hotspots* fornece dados sobre as funções mais demoradas (funções de *hotspot*) classificadas pelo tempo de CPU gasto em sua execução. Para a aplicação teste, a função *multiply0*, que levou 150.670 segundos para ser executada, aparece no topo da lista como a função mais ”quente”, e portanto, um possível ponto de partida para a paralelização. Inclusive, é possível visualizar no código-fonte quais as linha de código que demandam mais tempo de execução (Figura 2.2).

[1] ⟨https://www.intel.com/content/www/us/en/developer/tools/oneapi/vtune-profiler.html⟩

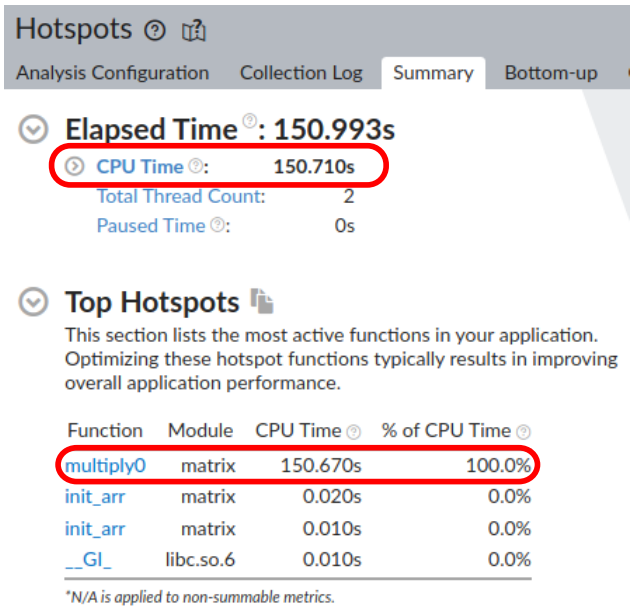


Figura 2.1. Detecção de hotspots usando Intel VTune.

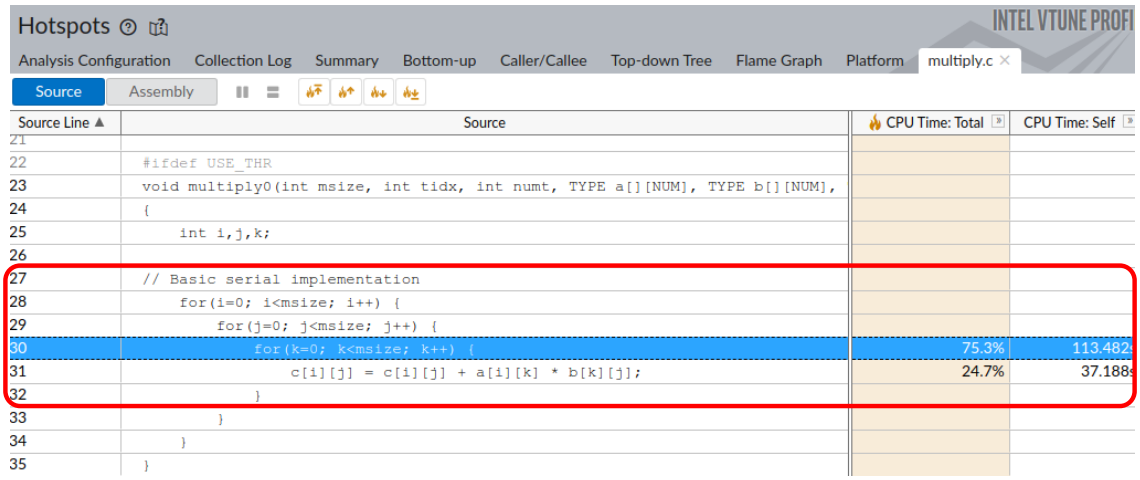


Figura 2.2. Trecho de código referente ao hotspot.

Mais detalhes sobre o uso da ferramenta pode ser obtido em sua documentação oficial[2].

[2] ⟨https://www.intel.com/content/www/us/en/developer/tools/oneapi/vtune-profiler-documentation.html⟩

2.4. Metodologia PCAM

Nessa seção, apresenta-se o método PCAM, apresentada por (Foster 1995). O nome deriva-se das iniciais das quatro fases que a compõem: (1) Particionamento, (2) Comunicação, (3) Agregação e (4) Mapeamento, conforme ilustrado na Figura 2.3.

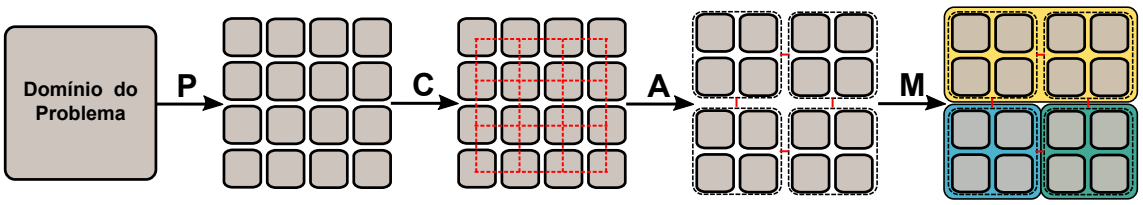


Figura 2.3. Fases da metodologia PCAM.

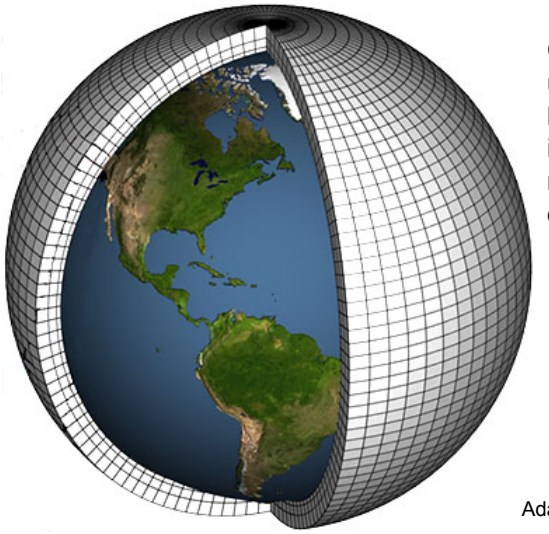
De acordo com o método, o projeto de um programa paralelo deve começar a partir de uma solução algorítmica existente (possivelmente sequencial) para um problema computacional, particionando-o em (muitas) pequenas tarefas e identificando dependências entre elas que podem resultar em comunicação e sincronização, para as quais deve-se selecionar as estruturas mais apropriadas. A granularidade da tarefa deve ser a mais fina possível para não restringir artificialmente as fases posteriores do projeto. O resultado dessas duas fases iniciais é um algoritmo paralelo escalável em um modelo de programação abstrato que é amplamente independente de um determinado computador paralelo e do número de processadores. Em seguida, as tarefas são aglomeradas em macrotarefas (processos) para reduzir a comunicação interna e as relações de sincronização dentro de uma macrotarefa a acessos à memória local. Por fim, mapeia-se as tarefas para os processadores, geralmente com o objetivo de minimizar o tempo total de execução. Técnicas de balanceamento de carga e/ou escalonamento de tarefas podem ser usadas para melhorar a qualidade do mapeamento. As duas últimas etapas, aglomeração e mapeamento, são mais dependentes da máquina porque usam informações sobre o número de processadores disponíveis, a topologia da rede, o custo de comunicação, etc.

2.4.1. Particionamento

Nesta primeira etapa, identificam-se fontes potenciais de paralelismo particionando (ou decompondo) o problema em pequenas tarefas. É a base de toda programação paralela, de uma forma ou de outra (Wilkinson and Allen 2005). O foco está em definir um grande número de pequenas tarefas para produzir uma decomposição refinada de um problema. A decomposição do problem em uma granularidade mais fina fornece flexibilidade em termos de possíveis algoritmos paralelos. Posteriormente, a avaliação dos requisitos de comunicação, a arquitetura da máquina ou questões de engenharia de software nos levam a abrir mão de muitas oportunidades identificadas para execução paralela neste estágio (Stanimirović 2020).

O particionamento pode ser aplicado aos dados do programa, dividindo-os e realizando o processamento sobre as partes simultaneamente. Essa abordagem de particionamento é chamada de *decomposição de domínio*. Diferentes partições podem ser possíveis, com base em diferentes estruturas de dados e modos de particionamento. Uma boa prática é focar na maior estrutura de dados ou naquela que é acessada com frequência.

A Figura 2.4 ilustra a decomposição de domínio em um modelo climático envolvendo uma grade tridimensional, composto por um conjunto de células. No estágio de particionamento, a ideia é decompor da forma mais agressiva possível, por exemplo, definindo uma tarefa para cada celula da grade. Nesse tipo de modelo, a computação é executado repetidamente, sendo que o cálculo referente a cada célula pode (potencialmente) ser resolvido em paralelo.



Os modelos climáticos dividem a Terra em uma grade com intervalos verticais e horizontais. Quanto menores os intervalos, mais fina a grade e melhor a resolução do modelo, ou seja, mais detalhes o modelo pode produzir.

Adaptado de: https://str.llnl.gov/december-2017/bader

Figura 2.4. Decomposição de domínio.

O particionamento também pode ser aplicado às funções de um programa, ou seja, dividindo-o em funções independentes e executando-as simultaneamente. Isso chama-se *decomposição funcional*. A ideia de executar uma tarefa dividindo-a em várias tarefas menores que, quando concluídas, completarão a tarefa geral é, obviamente, bem conhecida e pode ser aplicada em muitas situações, quer as tarefas menores operem em partes dos dados ou sejam funções concorrentes separadas. A Figura 2.5 ilustra um exemplo de decomposição funcional em um modelo climático.

Enquanto cada componente pode ser paralelizado mais naturalmente usando técnicas de decomposição de domínio, o algoritmo paralelo como um todo pode ser decomposto usando técnicas de decomposição funcional, mesmo que esse processo não produza um grande número de tarefas. Observe que o a decomposição de domínio geralmente tem uma granularidade mais fina do que o paralelismo de tarefas.

Embora a decomposição de domínio seja a base para a maioria dos algoritmos paralelos, a decomposição funcional é valiosa como uma maneira diferente de pensar sobre os problemas. Por esse motivo, deve ser considerado ao explorar possíveis algoritmos paralelos. Um foco nos cálculos a serem executados pode, às vezes, revelar a estrutura de um problema e, portanto, oportunidades de otimização, que não seriam óbvias apenas com o estudo dos dados.

Antes de avaliar as necessidades de comunicação, a checklist a seguir pode ser usada para garantir que o projeto não tenha falhas óbvias. Em geral, todas essas perguntas devem ser respondidas afirmativamente:

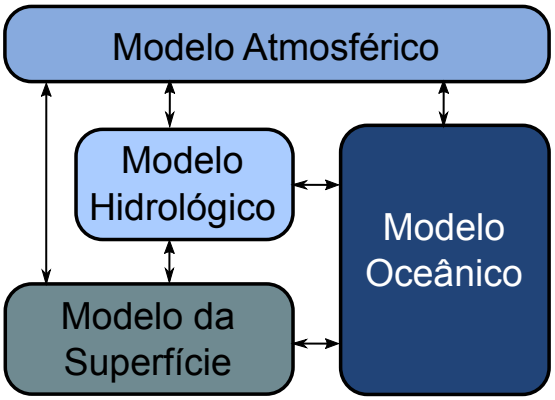


Figura 2.5. Decomposição funcional.

1. O particionamento do problema tem mais tarefas do que o número de processadores disponíveis? Se não tem, há pouca flexibilidade nos estágios de projeto subsequentes.

2. A partição evita computação e armazenamento redundantes?

3. As partições tem tamanho ou custo computacional semelhante? Quanto mais próximas, melhor, evitando problemas de balanceamento de carga.

4. O número de tarefas aumenta com o tamanho do problema? Idealmente, um aumento no tamanho do problema deve aumentar o número de tarefas em vez do tamanho das tarefas.

5. Foi possível identificar várias partições alternativas? Pode-se maximizar a flexibilidade nos estágios de projeto subsequentes considerando as diferentes alternativas. Lembre-se de verificar as decomposições de domínio e funcional.

2.4.2. Comunicação

Nesta etapa, determina-se a comunicação necessária entre as tarefas identificadas na primeira etapa. Especificam-se os dados que devem ser transferidos entre duas tarefas em termos de um canal que liga essas tarefas. Em um canal, uma tarefa pode enviar mensagens (produtor) e a outra pode receber (consumidor) (Schmidt et al. 2018).

Considerando o exemplo do modelo climático da seção anterior, as setas na Figura 2.5 representam trocas de dados entre componentes durante a computação: o modelo atmosférico gera dados de velocidade do vento que são usados pelo modelo oceânico, o modelo oceânico gera dados de temperatura da superfície do mar que são usados pelo modelo atmosférico e assim por diante.

Ainda no modelo climático, outro tipo de comunicação também pode ser necessário no caso da decomposição de domínios. Assumindo uma decomposição refinada na qual cada tarefa encapsula uma única célula da grade, e que o modelo numérico necessita de dados das 8 células vizinhas (estêncil de 9 pontos) na dimensão horizontal e duas células na vertical (estêncil de 3 pontos), o padrão de comunicação entre as células é como ilustrado na Figura 2.6.

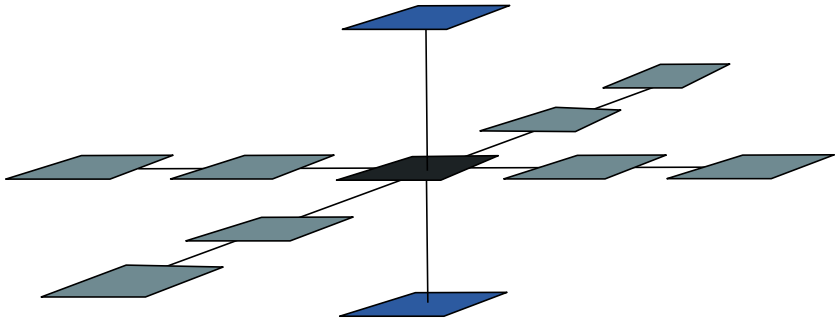


Figura 2.6. Para um único ponto de grade o estêncil de nove pontos é usado para troca de dados horizontal e o estêncil de três pontos usado para trocas de dados na vertical.

Na prática, existem muitos padrões de comunicação diferentes, como local ou global, estruturado ou não estruturado, estático ou dinâmico, síncrono ou assíncrono. Na comunicação local, cada tarefa se comunica com um pequeno conjunto de outras tarefas (seus ”vizinhos”); já a comunicação global exige que cada tarefa se comunique com muitas tarefas. Na comunicação estruturada, uma tarefa e seus vizinhos formam uma estrutura regular; em contraste, as redes de comunicação não estruturadas podem ser grafos arbitrários. Na comunicação estática, a identidade dos parceiros de comunicação não muda com o tempo; em contraste, a identidade dos parceiros de comunicação em estruturas de comunicação dinâmica pode ser determinada por dados computados em tempo de execução e pode ser altamente variável. Na comunicação síncrona, produtores e consumidores executam de forma coordenada, com pares produtor/consumidor cooperando em operações de transferência de dados; em contraste, a comunicação assíncrona pode exigir que um consumidor obtenha dados sem a cooperação do produtor.

A fase de comunicação também contém um checklist:

1. Todas as tarefas executam o mesmo número de operações de comunicação? Comunicação desequilibrada gera baixa escalabilidade.

2. Cada tarefa se comunica apenas com um pequeno número de vizinhos? Em caso necgativo, pode ser necessário reformular a comunicação global em termos de estruturas de comunicação locais.

3. As comunicações podem prosseguir em paralelo?

4. Os cálculos associados a diferentes tarefas podem ocorrer simultaneamente? Não: pode ser necessário reordenar cálculos/comunicações.

2.4.3. Aglomeração

No terceiro estágio, altera-se a granularidade do projeto obtida nos estágios anteriores, combinando uma série de pequenas tarefas em tarefas maiores. A aplicação particionada com granularidade muito fina pode não ser eficiente ao ser implementada diretamente em uma máquina real (em uma arquitetura de memória distribuída, por exemplo). Uma razão para isso é que a execução de um grande número de pequenas tarefas em paralelo usando diferentes processos/threads pode ser altamente ineficiente devido à sobrecarga

de comunicação. Para reduzir essa sobrecarga, pode ser benéfico agrupar várias tarefas pequenas em uma única tarefa maior no mesmo processador. Isso geralmente melhora a *localidade dos dados* e, portanto, reduz a quantidade de dados a serem comunicados entre as tarefas.

Na Figura 2.7, apresenta-se um recorte de 8×8 células da malha horizontal do modelo climático (a) e duas possibilidades de aglomeração (b) e (c). Note que o número de partições é distinto, bem com as necessidades de comunicação entre essas partições. Portanto, para um determinado problema pode haver diversas formas de aglomerar os dados, e cada uma delas pode resultar em diferentes resultados de desempenho.

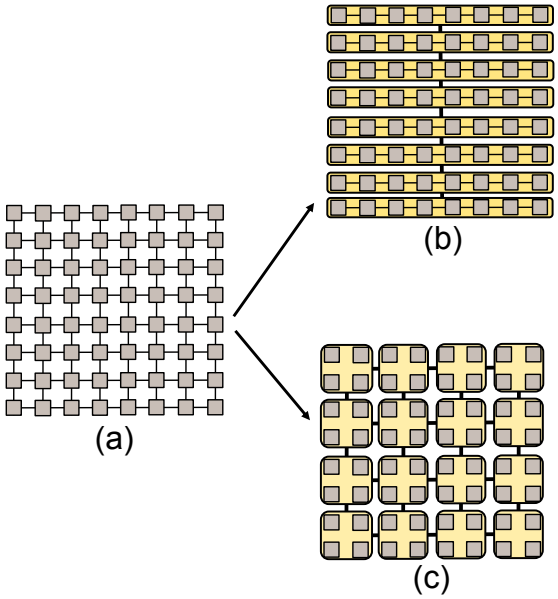


Figura 2.7. Diferentes abordagens de aglomeração. Partindo da mesma partição e padrão de comunicação (a), na primeira abordagem aglomera-se todas as tarefas ao longo da mesma linha (b) e na segunda abordagem aglomera-se uma grade quadrada de tarefas (c).

Nesta seção revisitou-se as decisões de particionamento e comunicação desenvolvidas nos dois primeiros estágios do projeto, aglomerando tarefas e operações de comunicação. A seguir, apresenta-se o checklist par esta fase, no qual as enfatizam a análise quantitativa do desempenho, que se torna mais importante à medida que passamos do abstrato para o concreto:

1. A aglomeração reduziu os custos de comunicação? Em caso negaitvo, deve ser usado uma estratégia alternativa de aglomeração.

2. A aglomeração produziu tarefas com custos de computação e comunicação semelhantes? Quanto maiores as tarefas criadas pela aglomeração, mais importante é que tenham custos semelhantes. Se criamos apenas uma tarefa por processador, essas tarefas devem ter custos quase idênticos.

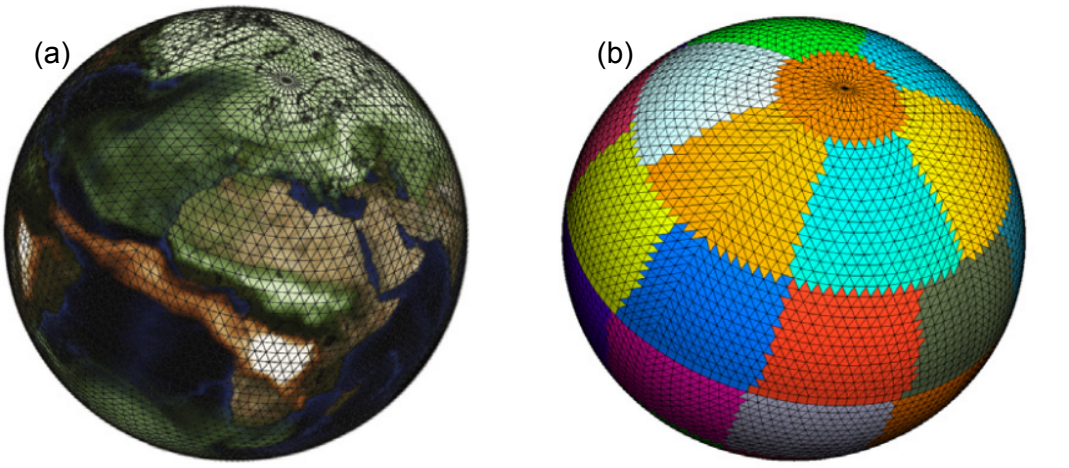
3. O número de tarefas ainda aumenta com o tamanho do problema? Caso contrário, seu algoritmo não será mais capaz de resolver problemas maiores em computadores paralelos maiores.

4. O número de tarefas pode ser reduzido ainda mais, sem introduzir desbalanceamento de carga ou reduzir a escalabilidade? Algoritmos que criam menos tarefas granulares são geralmente mais simples e mais eficientes do que aqueles que criam muitas tarefas refinadas.

2.4.4. Ferramentas de particionamento de grafos

Como visto, escolher o modo de aglomeração mais apropriado não é trivial principalmente quando temos domínios irregulares ou não-estruturados. Por exemplo, simulações numéricas de grande escala em computadores paralelos, requerem a distribuição dos elementos da malha entre os processadores. Essa distribuição deve ser feita de forma a balancear as computações entre os processadores e minimizando a comunicação (Barlas 2014).

Em muitos casos, a aglomeração pode ser mapeada para um problema de particionamento de grafos. O problema de particionamento de grafos consiste na divisão do conjunto de nós de um grafo em k blocos de tamanho igual, de modo que o número de arestas que correm entre os blocos seja minimizado (Buluç et al. 2016). Um exemplo de domínio computacional particionado é apresentado na Figura 2.8. Cada partição (conjunto de células) é representado por uma cor e possuem um número muito próximo de elementos.



Adaptada de Deconinck et al. (2017).

Figura 2.8. Malha esférica contendo 5248 células, agrupadas em 32 partições.

Na decomposição funcional, as tarefas geralmente dependem umas das outras, pois calculam dados que são usados por outras tarefas. As dependências entre as tarefas são descritas por meio de um grafo de dependência acíclico (DAG). Os vértices do grafo representam tarefas, e os arcos representam dependências entre tarefas decorrentes da transferência de dados ou sincronização de seu trabalho garantindo a ordem adequada de execução das tarefas. A aglomeração das tarefas com base na decomposição funcional

consiste em particionar o conjunto de vértices do grafo de dependência em subconjuntos que são atribuídos aos respectivos processadores (Czech 2016). A Figura 2.9 ilustra a aglomeração de um conjunto de tarefas mapeadas como um DAG. Cada cor representa um grupo de tarefas.

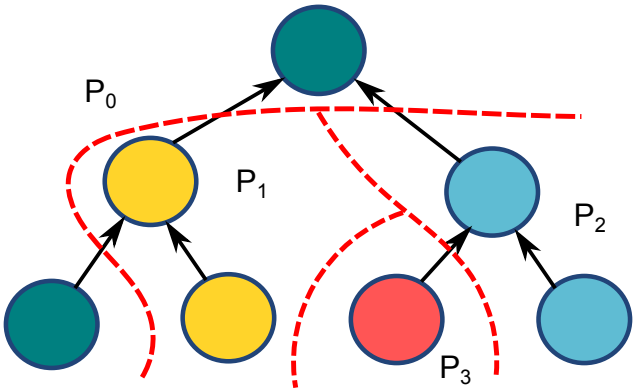


Figura 2.9. Aglomeração de tarefas baseada no particionamento de grafos. Adaptada de (Czech 2016)

Por se tratar de um problema NP-Difícil, ferramentas de particionaemnto que implementam heurísticas podem auxiliar nessa tarefa. Exemplos são os softwares Metis[3] e Zoltan[4] . O funcionamento dessas ferramentas está fora do escopo desse minicurso. Mais detalhes sobre as ferramentas podem ser encontradas em suas documentações oficiais.

2.4.5. Mapeamento

Para que a aplicação seja executada, os grupos de tarefas produzidos pela terceira etapa devem ser atribuídos/mapeados aos nós/processadores/cores disponíveis na arquitetura alvo. Os objetivos que precisam ser alcançados nesta etapa são (a) balancear a carga dos nós, ou seja, todos eles devem ter aproximadamente a mesma quantidade de trabalho medido pelo tempo de execução, e (b) minimizar a comunicação entre processadores atribuindo tarefas com interações frequentes ao mesmo processador (ou mais próximos).

A complexidade do mapeamento pode ser bastante distinta, dependendo da arquitetura e das características da aplicação. Por exemplo, muitos algoritmos desenvolvidos usando técnicas de decomposição de domínio apresentam um número fixo de tarefas de tamanho igual e comunicação local e global estruturada. Nesses casos, um mapeamento eficiente ocorre de modo direto. Por outro lado, em algoritmos baseados em decomposição de domínio mais complexos com quantidades dinâmicas de trabalho por tarefa e/ou padrões de comunicação não estruturados, estratégias eficientes de aglomeração e mapeamento podem não ser óbvias.

A Figura 2.10 mostra dois cenários relacionados ao mapeamento de 16 grupos de tarefas. No primeiro cenário (a), as 16 tarefas são mapeadas em quatro nós de processamento com a mesma configuração de hardware, e portanto, com mesma capacidade de computação. Nesse caso, o grupo é divido igualmente entre os nós. No cenário (b), as

[3] ⟨https://github.com/KarypisLab/ParMETIS⟩
[4] ⟨https://sandialabs.github.io/Zoltan/⟩

tarefas são distribuídas entre 3 nós heterogêneos, sendo que 2 deles possuem mais capacidade de computação que o terceiro. Aqui, as duas máquinas devem receber mais grupos para processamento que a terceira máquina.

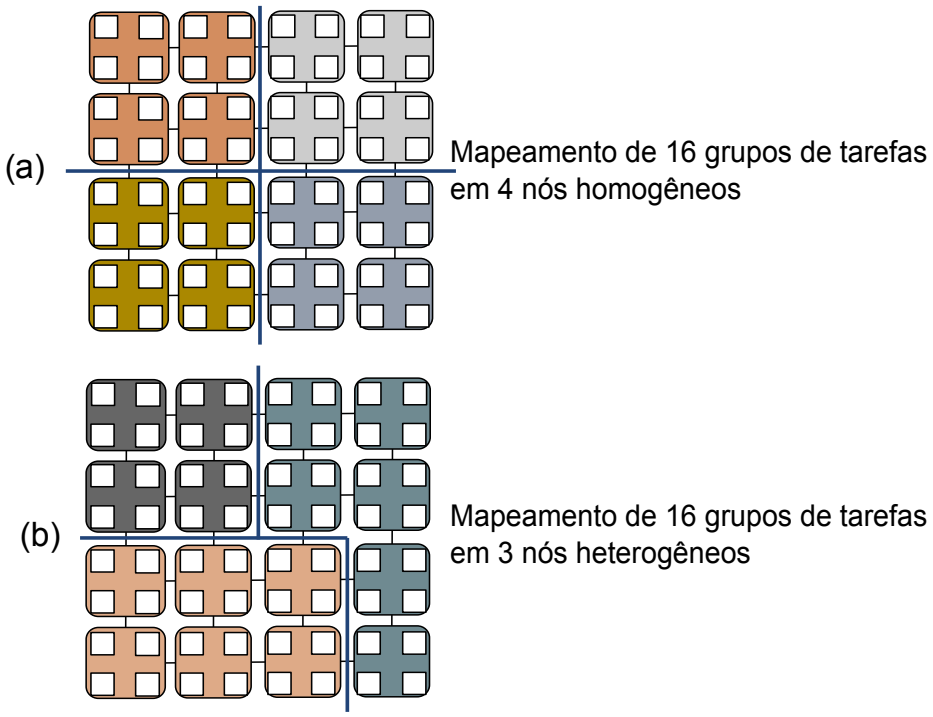


Figura 2.10. Diferentes mapeamentos de tarefas.

O mapeamento em uniprocessadores ou em computadores de memória compartilhada é mais simples, geralmente realizado pelo próprio sistema operacional ou por uma platarforma de execução (runtime). No entanto, o mapeamento em máquinas de memória distribuída deve ser feito por escalonadores e balanceadores de carga apropriados. Os detalhes dos escalonadores e balanceadores de carga está além do contexto desse minicurso. Sugestões e leitura sobre o assunto são as referências (Czech 2016) e (Barlas 2014).

2.5. Padrões de Projeto para Aplicações Paralelas

Um padrão de projeto descreve uma boa solução para um problema recorrente em um contexto particular. A ideia é registrar a experiência dos especialistas de uma forma que possa ser usada por outros que enfrentem um problema semelhante.

(Mattson et al. 2004) propõem um conjunto de padrões organizado em quatro espaços de projeto: (1) Encontrando o Paralelismo, (2) Estrutura do Algoritmo, (3) Estruturas de Apoio e (4) Mecanismos de Implementação. O fluxo descrito na Figura 2.11 sugere que o programador realize a modelagem do sistema seguindo passos que permitem explorar o paralelismo de maneira adequada. Os passos consistem em definir a melhor maneira para encontrar porções paralelizáveis, estruturar o sistema para que as partes paralelizáveis possam ser executadas de maneira concorrente, escolher as estruturas de suporte à execução paralela que mais condizem com o algoritmo definido e por fim, es-

colher quais os mecanismos de implementação que permitem transformar estas estruturas de suporte em programas executáveis.

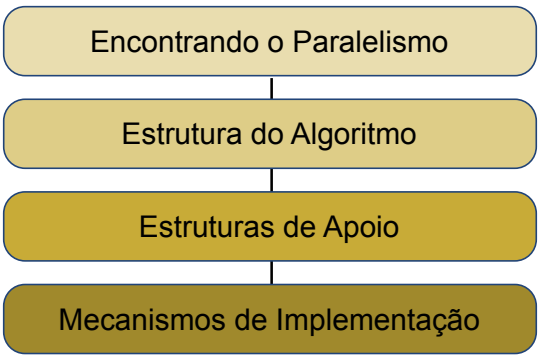


Figura 2.11. Espaços de projeto propostos por (Mattson et al. 2004).

2.5.1. Encontrando o Paralelismo

Nesta primeira etapa, o projetista precisa entender quais partes do problema são mais computacionalmente intensivas, porque o esforço para paralelizar o problema deve ser focado nessas partes (Hotspots, já tratados na Seção 1.3 deste minicurso).

Após a conclusão dessa análise, os padrões no espaço de projeto *Encontrando o Paralelismo* podem ser usados para iniciar o projeto de um algoritmo paralelo. Os padrões neste espaço de design podem ser organizados em dois grupos: (1) Decomposição, (2) Análise de dependências.

Basicamente, os padrões de decomposição *Decomposição de tarefas* (ou funcional) e *Decomposição de dados* (ou de domínio), são usados para decompor o problema em partes que podem ser executadas simultaneamente. Note que essa etapa relaciona-se à etapa de Particionamento da metodologia PCAM (ver Seção 1.4.1).

O grupo de Análise de Dependência contém três padrões que ajudam a agrupar as tarefas e analisar as dependências entre elas: Agrupamento de Tarefas, Ordenação de Tarefas e Compartilhamento de Dados. O ponto de partida em uma análise de dependência é agrupar tarefas com base em restrições entre elas e, em seguida, determinar quais restrições de ordenação se aplicam a grupos de tarefas. O próximo passo é analisar como os dados são compartilhados entre os grupos de tarefas, para que o acesso aos dados compartilhados seja gerenciado corretamente. Esse padrão equivale à fase de Comunicação e Aglomeração da metodologia PCAM, apresentada na Seção 1.4.3.

2.5.2. Estrutura do Algoritmo

Objetivo do espaço de Estrutura do Algoritmo é refinar o projeto e aproximá-lo de um programa que pode executar tarefas simultaneamente mapeando a simultaneidade em várias unidades de execução executando em um computador paralelo.

(Mattson et al. 2004) listam uma série de padrões de decomposição que podem abranger as formas básicas pelas quais uma carga de trabalho pode ser decomposta para eventual distribuição aos nós de uma plataforma paralela/multicore. A Figura 2.12 mostra a árvore de decisão que leva a um dos seis padrões possíveis. Em geral, um problema pode

ser decomposto de várias maneiras diferentes. A variedade de padrões nos permite pensar sobre o problema de paralelização de diferentes perspectivas.

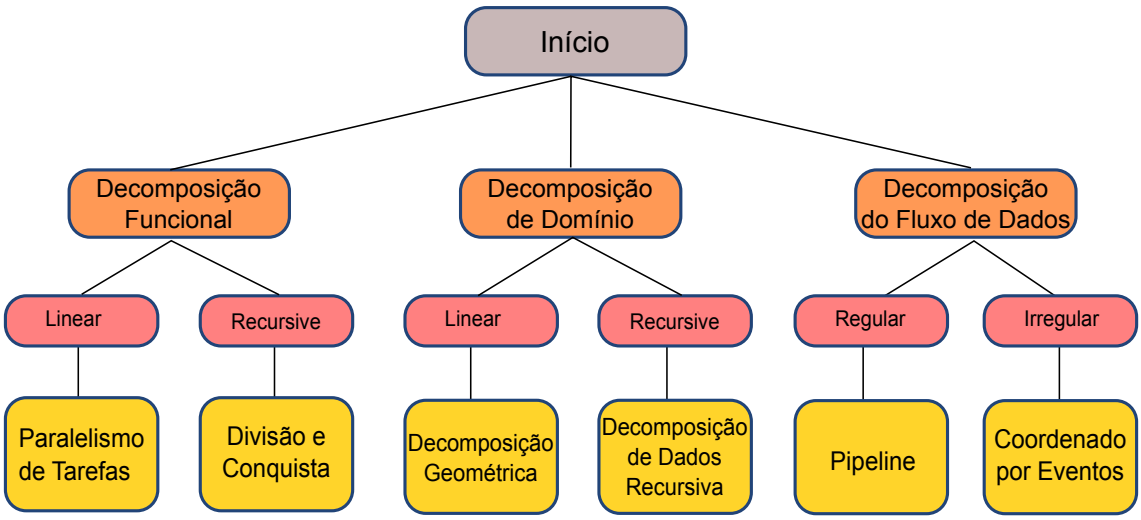


Figura 2.12. Padrões para Estrutuda do Algoritmo.

A Decomposição Funcional deve ser escolhida quando a execução das próprias tarefas for o melhor princípio de organização. Em seguida, determine como as tarefas são enumeradas. Se eles puderem ser reunidos em um conjunto linear em qualquer número de dimensões, escolha o padrão Paralelismo de Tarefas. Este padrão inclui tanto situações em que as tarefas são independentes umas das outras (trivialmente paralelizáveis) quanto situações em que existem algumas dependências entre as tarefas na forma de acesso a dados compartilhados ou necessidade de troca de mensagens. Se as tarefas forem enumeradas por um procedimento recursivo, escolha o padrão Dividir e Conquistar. Nesse padrão, o problema é resolvido dividindo-o recursivamente em subproblemas, resolvendo cada subproblema independentemente e, em seguida, recombinando as subsoluções na solução do problema original.

A Decomposição de Domínio é mais apropriada quando o particionamento dos dados for o principal forma de organizar o paralelismo. O padrão de Decomposição Geométrica pode ser escolhido quando o espaço do problema for decomposto em partições discretas e o problema for resolvido computando soluções para estas partições, com a necessidade de troca de dados com outras partições. A escolha do padrão Dados Recursivos é mais apropriada quando o problema for definido em termos de uma estrutura de dados recursiva (por exemplo, uma árvore binária).

A Organizar por Fluxo de Dados pode ser usada quando o princípio de organização é baseado em como o fluxo de dados impõe uma ordem nos grupos de tarefas. Os dados podem seguir uma sequência predeterminada de etapas que levam ao uso do padrão Pipeline, ou podem seguir padrões irregulares, como no caso de simulações de eventos discretos, levando ao uso do padrão Coordenado por Eventos.

2.5.3. Estruturas de Apoio

Os padrões no espaço de projeto de Estruturas de Apoio abordam a fase do processo de projeto de programa paralelo, representando um estágio intermediário entre os padrões orientados a problemas do espaço de projeto de Estrutura de Algoritmo e os mecanismos de programação específicos descritos em Mecanismos de Implementação. Alguns padrões de estrutura são resumidos a seguir (Barlas 2014).

SPMD Em um programa SPMD (Single Program, Multiple Data) todos as unidades de processamento da plataforma de execução executam o mesmo programa, mas podem aplicar as mesmas operações em dados diferentes e/ou seguir diferentes caminhos de execução dentro do programa. Geralmente, essa diferenciação é baseada em um identificador de processo/thread.

MPDM O padrão MPMD (Multiple-Program, Multiple-Data) permite que diferentes executáveis componham uma aplicação. Cada nó de computação é livre para executar sua própria lógica de programa e processar seu próprio conjunto de dados.

Master/Worker Este padrão distingue dois tipos de atores: um mestre e um conjunto de escravos. O mestre coordena a execução da aplicação atribuindo e escalonando unidades de trabalho a cada um dos escravos para processamento. Os escravos recebem as tarefas, as processam e retornam os resultados. Assume-se que o trabalho pode ser dividido em um conjunto de tarefas independentes que podem ser processadas independentemente pelos escravos. Aqui, o número de tarefas pode ser igual ao número de escravos ou o trabalho pode ser dividido em um número mais significativo de tarefas

Loop Parallelism Este padrão aborda o problema de transformar um programa serial, cujo tempo de execução é dominado por um conjunto de laços, em um programa paralelo onde as diferentes iterações do loop são executadas em paralelo.

Fork/Join O padrão fork/join é empregado quando o algoritmo paralelo solicita a criação dinâmica (fork) de tarefas em tempo de execução. Essas tarefas filhas (processos ou threads) geralmente precisam terminar (join) antes que o processo/thread pai possa retomar a execução.

Qual é a melhor forma de estruturar o programa? A resposta é altamente dependente da aplicação. Alguns padrões podem ser utilizados de forma isolada, mas também podem ser combinados de maneiras diferentes para atender às necessidades de um determinado problema. Várias plataformas de execução podem impor um padrão específico de estrutura de programa aos desenvolvedores. Por exemplo, o MPI usa o padrão SPMD/MPMD, enquanto o OpenMP promove o padrão de paralelismo de laço (Barlas 2014). Além disso, dado um padrão de decomposição específico, certos padrões de estrutura de programa são mais adequados para fins de implementação. A Tabela na Figura 2.5.3 resume essas combinações.

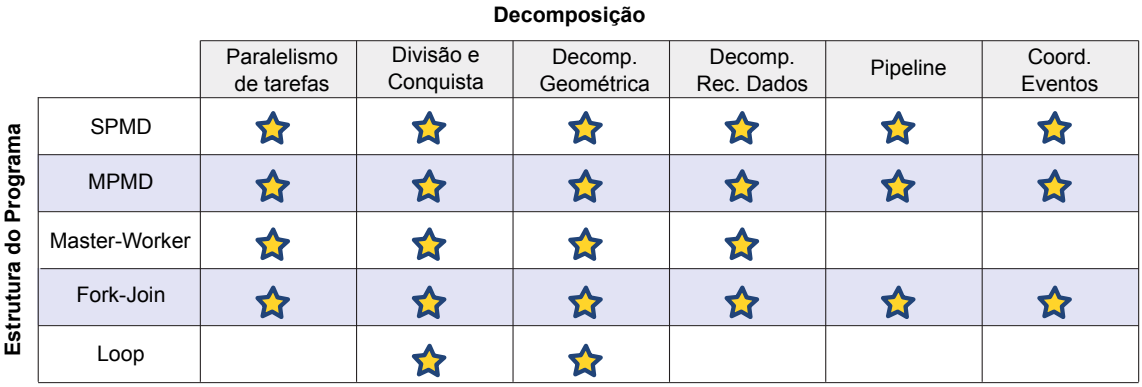


Figura 2.13. Padrões de Decomposição e os Padrões de Estrutura de Programa Mais Adequados para Implementá-los.

2.5.4. Mecanismos de Implementação

Os mecanismos de implementação definem a gestão de unidades de execução, comunicação e sincronização. Neste nível, os padrões são estabelecidos pelo modelo de programação, de forma que cabe ao programador escolher adequadamente este modelo.

Dependendo da arquitetura alvo, diferentes unidades de execução (UE). Para ambientes de software baseados em memória distribuída, como MPI, as UEs são mapeadas em processos. Ambientes baseados em memória compartilhada, como OpenMP, utilizam threads.

Na maioria dos algoritmos paralelos, as UEs precisam se comunicar para trocar informações à medida que a computação prossegue. Os ambientes de memória compartilhada fornecem esse recurso por padrão. Em sistemas de memória distribuída, como não há uma memória comum entre os processadores, a comunicação é feita por troca de mensagens, que pode ser feita de modo coletivo (broadcast, multicast) ou ponto-a-ponto (send/recv).

No gerenciamento de threads e processos, a sincronização ocorre quando se deseja manter uma ordem na execução dos eventos. Para sincronização, os padrões utilizados são: Cercas (Fences), Barreiras e Exclusão Mútua.

2.6. Considerações Finais

O objetivo deste minicurso foi apresentar uma visão introdutória sobre o projeto de aplicações paralelas, de modo a servir como um ponto de partida para os iniciantes na área. Foram duas abordagens diferentes, PCAM e Padrões de Projeto, cada uma com suas particularidades mas que podem ser utilizadas de forma complementar.

Para um aprofundamento deste tópico, recomenda-se a leitura dos livros indicados nas referências.

Referências

Barlas 2014 Barlas, G. (2014). *Multicore and GPU Programming: An Integrated Approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.

Buluç et al. 2016 Buluç, A., Meyerhenke, H., Safro, I., Sanders, P., and Schulz, C. (2016). *Recent Advances in Graph Partitioning*, pages 117–158. Springer International Publishing, Cham.

Czarnul 2018 Czarnul, P. (2018). *Parallel Programming for Modern High Performance Computing Systems*. Chapman, 1st edition.

Czech 2016 Czech, Z. J. (2016.). *Introduction to parallel computing /*. Cambridge University Press,, Cambridge:. Includes index.

Foster 1995 Foster, I. T. (1995). *Designing and building parallel programs - concepts and tools for parallel software engineering*. Addison-Wesley.

Mattson et al. 2004 Mattson, T., Sanders, B., and Massingill, B. (2004). *Patterns for Parallel Programming*. Addison-Wesley Professional, first edition.

Pacheco 2011 Pacheco, P. (2011). *An Introduction to Parallel Programming*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1st edition.

Schmidt et al. 2018 Schmidt, B., González-Domínguez, J., Hundt, C., and Schlarb, M. (2018). *Parallel Programming*. Morgan Kaufmann.

Stanimirović 2020 Stanimirović, I. (2020). *Parallel Programming*. Arcler Press, first edition.

Trobec et al. 2018 Trobec, R., Slivnik, B., Bulic, P., and Robic, B. (2018). *Introduction to Parallel Computing - From Algorithms to Programming on State-of-the-Art Platforms*. Undergraduate Topics in Computer Science. Springer.

Wilkinson and Allen 2005 Wilkinson, B. and Allen, M. (2005). *Parallel programming - techniques and applications using networked workstations and parallel computers (2. ed.)*. Pearson Education.