5
Aplicações de Alto Desempenho

Felipe Martins Müller[1] *(UFSM, felipe@inf.ufsm.br)*
Andréa Schwertner Charão[2] *(UFSM, andrea@inf.ufsm.br)*
Haroldo Gambini Santos[3] *(Sociedade Meridional de Educação, haroldo@inf.ufsm.br)*

Resumo:

Este curso apresenta duas classes de aplicações onde o processamento de alto desempenho tem auxiliado e viabilizado a resolução de problemas. A primeira parte do curso trata de metaheurísticas aplicadas a problemas de otimização combinatória utilizando estratégias de processamento paralelo e distribuído para melhorar o desempenho das mesmas. A segunda parte aborda a resolução de problemas de Equações Diferenciais Parciais, que constituem o cerne de muitas aplicações de simulação numérica de grande porte. Para cada classe de aplicações é feita uma caracterização do problema, seguida pela apresentação de soluções que visam a utilização eficiente de plataformas de alto desempenho.

[1]Doutor em Engenharia El´etrica pela UNICAMP. Diretor do Centro de Tecnologia/UFSM. Professor e Pesquisador no Programa de P´os-Graduaç ˜ao em Engenharia de Produç ˜ao (PPGEP/UFSM).

[2]Doutora em Ciência da Computaç ˜ao pelo INPG (Grenoble, Franç a). Professora da UFSM e Pesquisadora do Laborat´orio de Sistemas de Computaç ˜ao (LSC/UFSM).

[3]Mestre em Engenharia de Produç ˜ao pelo PPGEP/UFSM. Programador da Sociedade Meridional de Educaç ˜ao.

5.1. Introdução

O progresso notável nas áreas ligadas à micro-eletrônica fornece microprocessadores cada vez mais poderosos, em um ritmo muitas vezes superior às previsões mais otimistas. No entanto, as necessidades de certas aplicações "consumidoras" do processamento de alto desempenho estão habitualmente além dos recursos disponíveis. De fato, a conquista do poder computacional para resolver problemas "de ponta" é naturalmente seguida pelo lançamento de novos desafios.

O processamento paralelo e distribuído constitui uma resposta a estas demandas de alto desempenho, podendo ser visto como uma maneira de contornar as barreiras tecnológicas e econômicas que se impõem ao desenvolvimento de microprocessadores mais rápidos. Os avanços nesta área têm sido reforçados nos últimos anos pela pesquisa e o desenvolvimento em torno de aglomerados (*clusters*) de computadores e grades (*grids*) de computação. Este curso apresenta duas classes de aplicações relevantes neste contexto, caracterizadas por uma demanda de poder de processamento que ultrapassa os limites dos paradigmas computacionais "convencionais".

Os **problemas de otimização combinatória** têm sido o foco de inúmeras pesquisas nos ramos da matemática e da computação. Isso se deve, em parte, ao imenso conjunto de aplicações que esses problemas apresentam nas áreas da engenharia e da gestão empresarial, e em parte a grande dificuldade que se constitui a resolução eficiente desses. Mesmo para casos de pequenas dimensões, a enumeração completa das soluções possíveis não é viável devido à explosão de possibilidades. Na primeira parte desse curso (seção 5.2.) será apresentado um ambiente de execução e gerenciamento de métodos de resolução para esses problemas que permite a utilização de recursos computacionais distribuídos e heterogêneos para a resolução eficiente desses problemas.

A resolução de **problemas de Equações Diferenciais Parciais** também constitui um campo de pesquisa bastante ativo, tratando-se de um cálculo que consome grande parte do tempo de processamento de muitas aplicações de simulação computacional. A paralelização desta resolução não é uma tarefa trivial, especialmente em arquiteturas com memória distribuída. Em particular, as propriedades numéricas dos métodos empregados precisam ser respeitadas, o que se traduz em comunicações e pontos de sincronização que tendem a degradar o desempenho paralelo. A segunda parte deste curso (seção 5.3.) aborda estas questões em maior detalhe, fornecendo uma breve introdução aos métodos numéricos mais relevantes e à sua implementação em arquiteturas paralelas.

5.2. Problemas de Otimização Combinatória

A área da otimização combinatória (OC) trata da alocação eficiente de recursos limitados, de modo que certos objetivos sejam alcançados. Especificamente, esses problemas têm uma formulação matemática que restringe os valores das variáveis em um conjunto discreto de possibilidades. Assim, dado um conjunto possíveis soluções, a resolução desses problemas inclui a geração, avaliação e comparação de soluções, usualmente, respeitando-se limites de tempo. As aplicações das técnicas de busca de soluções são inúmeras, e incluem, por exemplo, problemas de seqüenciamento, com larga aplicação na engenharia, problemas de satisfação de restrições e mesmo jogos como *Rubic's cube*.

O uso da estrutura da Internet na ajuda ao desenvolvimento e teste de novos

métodos de otimização tem se alterado nos últimos anos. Inicialmente, diretórios de instâncias de teste como o TSPLib [REI 91] e OR-Library (http://mscmga.ms.ic.ac.uk/info.html) tornaram disponíveis bases comuns para avaliação do desempenho e robustez dos métodos de otimização combinatória. Baseados na mesma tecnologia, repositórios de códigos como o Netlib (www.netlib.org) permitem o acesso a arquivos de softwares matemáticos.

Passos importantes estão sendo dados no sentido de substituir-se a simples transferência de arquivos estáticos por provedores de serviços de otimização. Ferramentas de programação matemática como o *NEOS Solver* [CZY 97] aceitam a submissão de instâncias, juntamente com a modelagem matemática que permite a sua resolução remota.

Para que se ofereçam melhores ferramentas de OC através da Internet, dois principais tipos de usuários devem ser considerados. Primeiramente, pesquisadores da área da OC que desejam avaliar comparativamente os métodos disponíveis, ou combiná-los, na tentativa de criação de métodos mais robustos. Segundo, usuários de métodos de OC que desejam resolver problemas sem ter que instalar e configurar pacotes específicos de sofware.

CORE (*Combinatorial Optimization Resource Management Environment*) é um ambiente baseado na estrutura da Internet, que utiliza recursos computacionais distribuídos e heterogêneos para a resolução remota de problemas de otimização combinatória. Ainda, um repositório de instâncias guarda casos de teste, com os melhores valores de solução encontrados para as instâncias em questão e, quando disponíveis, os valores ótimos de solução. Com a ajuda de assistentes, usuários podem criar planos de otimização, navegar em um repositórios de instâncias, enviar planos de otimização para servidores remotos e receber os resultados computados e sumarizados. O ambiente foi especialmente desenvolvido para métodos heurísticos e metaheurísticos. No entanto, métodos exatos também são suportados.

Nas seções subseqüentes, será dada uma breve visão das técnicas de solução para problemas de otimização combinatória, seguida pela apresentação dos trabalhos relacionados ao provimento de serviços remotos de otimização utilizando recursos distribuídos, com a posterior apresentação da arquitetura do ambiente *CORE* e um caso de estudo para o problema de seqüenciamento de tarefas em máquinas paralelas com detalhamento da implementação da metaheurística GRASP. Finalmente, são discutidas as estratégias de paralelização dessa metaheurística e as implicações para a implementação no ambiente *CORE*.

5.2.1. Visão geral das técnicas de otimização

Resolver problemas de otimização combinatória, ou seja, encontrar a solução ótima para esses problemas, pode ser uma tarefa difícil. De fato, a modelagem de aspectos complexos de problemas encontrados no mundo real geralmente leva à classes de problemas (do tipo *NP*-Árduo), para os quais algoritmos eficientes provavelmente não existem [GAR 79].

Os métodos de resolução podem, de maneira simplificada, serem classificados em heurísticos e exatos. Uma vez que o uso de algoritmos heurísticos aumenta e eficiência do processo de busca, evitando a exploração de todas as alternativas, analistas de pesquisa operacional têm, freqüentemente, considerado o uso de heurísticas para a obtenção de boas soluções (sem garantia de otimalidade) para problemas nos quais os algoritmos exatos correntes são incapazes de prover soluções em tempo razoáveis.

Os blocos básicos para a construção de heurísticas são os algoritmos construtivos e de melhoramento. Algoritmos construtivos montam, passo a passo, uma solução, de modo que ao final do processo, uma solução válida é encontrada. Um exemplo clássico de heurística construtiva são os algoritmos gulosos que, a cada passo, selecionam a alternativa que fornece o maior lucro imediato.

Algoritmos de melhoramento iniciam com uma solução válida e através da busca local (movimentos na solução inicial), tentam melhorar a qualidade de solução.

Metaheurísticas são um passo à frente desses procedimentos construtivos / melhoramento, e tornaram-se populares nos anos 80, quando algoritmos que permitiam movimentos de piora na qualidade de solução foram propostos, em uma tentativa de evitar a parada prematura da busca em uma solução sub-ótima. *Simulated annealing* [KIR 83], Algoritmos Genéticos [GOL 89] e Busca Tabu [GLO 97] são exemplos de metaheurísticas.

Uma metaheurística que tem se destacado pela facilidade de implementação e qualidade dos resultados obtidos é *GRASP* [FEO 95], que se constitui de um algoritmo construtivo semi-guloso e uma fase de melhoramento, como pode ser visto no algoritmo 1.

```
função GRASP (iterações) {
        melhorCusto = MAX;
        para i de 1 até iterações faça {
                x = construtivoSemiGuloso;
                x = buscaLocal(x);
                se (custo(x) < melhorCusto) então {
                        x* = x;
                        melhorCusto=custo(x);
                }
        }
        retorne x*;
}
```

Algoritmo 1 - Metaheurística GRASP.

A diversificação das soluções se dá através do construtivo semi-guloso (algoritmo 2), que contém um componente aleatório que perturba a seleção em um algoritmo construtivo guloso. Mais especificamente, ao invés de, a cada iteração, selecionar-se o componente com maior lucro imediato, se constrói uma lista restrita de opções (*RCL*) e escolhe-se, de maneira aleatória entre esses. O grau de aleatoriedade é controlado pelo parâmetro *alpha*, variando entre 1 (completamente guloso) e 0 (completamente aleatório). Ao concluir a solução, aplica-se a busca local (algoritmo 3). O processo é repetido por n iterações e, ao final se retorna a melhor solução encontrada.

```
função ConstrutivoSemiGuloso ( alpha ) {
        solução ← Ø;
        repita {
                avalie o custo incremental de cada elemento;
                construa a lista de candidatos restrita (RCL), de acordo com alpha;
                selecione o elemento s da RCL de modo aleatório;
                solução ← solução U {s} ;
        } enquanto solução incompleta;
        retorne solução;
}
```

Algoritmo 2 - Construtivo semi-guloso.

```
função BuscaLocal ( solução ) {
        enquanto (solução não for ótimo local) faça {
                encontre solução' ;
                solução = solução';
        }
        retorne solução;
}
```

Algoritmo 3 - algoritmo de busca local.

5.2.2. Trabalhos relacionados

Muitos projetos têm se dedicado a oferecer serviços de otimização combinatória através da estrutura da Internet [FOU 2000]. A maioria oferece serviços baseados em programação matemática. O acesso a esses serviços é feito através do envio de um modelo de resolução, juntamente com a instância para um servidor remoto.

O sistema *Progress* [BEC 98] é um sistema baseado na arquitetura cliente / servidor, juntamente com uma linguagem orientada a objetos e uma ferramenta para a construção de planos que provê acesso a algoritmos distribuídos. O principal propósito do sistema *Progress* é oferecer uma camada de interação entre implementações distribuídas.

O projeto *NEOS* (*Network-Enabled Optimization Server*) [CZY 97] permite o acesso a grupos de servidores de ferramentas matemáticas registradas em um servidor central. Novas ferramentas podem ser registradas dinamicamente e, uma vez que existe um grande número de servidores disponíveis, mecanismos de tolerância a falhas foram implementados. As requisições podem ser enviadas para as ferramentas através de formulários *web*, *e-mail*, ou através de um aplicativo baseado em *sockets* TCP-IP. Como os tempos de resolução podem demandar um esforço computacional impraticável, grande parte das ferramentas impõe restrições no envio de requisições, em especial no tamanho da instância ou no tempo gasto na resolução do problema. Uma questão em aberto nesses serviços é a padronização no formato das instâncias e modelos, o que torna a utilização de diferentes ferramentas uma tarefa trabalhosa.

A ferramenta *NetSolve* [ARN 2001] oferece *interfaces* cliente / servidor que ajudam na construção de sistemas distribuídos para a resolução de problemas de otimização combinatória. *Interfaces* para C, Fortran, Matlab e outras linguagens de programação encontram-se disponíveis.

5.2.3. Arquitetura do ambiente CORE

O ambiente CORE foi desenvolvido através de uma arquitetura de objetos distribuídos, com uma clara distinção entre os componentes de dados, processamento e visualização. Enquanto muitos módulos do sistema foram projetados utilizando-se padrões existentes [GAM 95], o pacote de software que trata especificamente de otimização combinatória exigiu um novo conjunto de padrões, que serão descritos na seqüência.

A divisão arquitetural (figura 5.1) básica utiliza o modelo de computação cliente / servidor. No lado servidor, tem-se o **Repositório**, que armazena métodos, instâncias de dados e usuários, bem como o **Servidor de Otimização**, que oferece o poder de processamento de um nodo para o ambiente.

Na camada cliente, existem as ferramentas administrativas, como **Gerenciador do Repositório**, utilizado na manutenção dos recursos, e o **Gerenciador de Usuários**, que

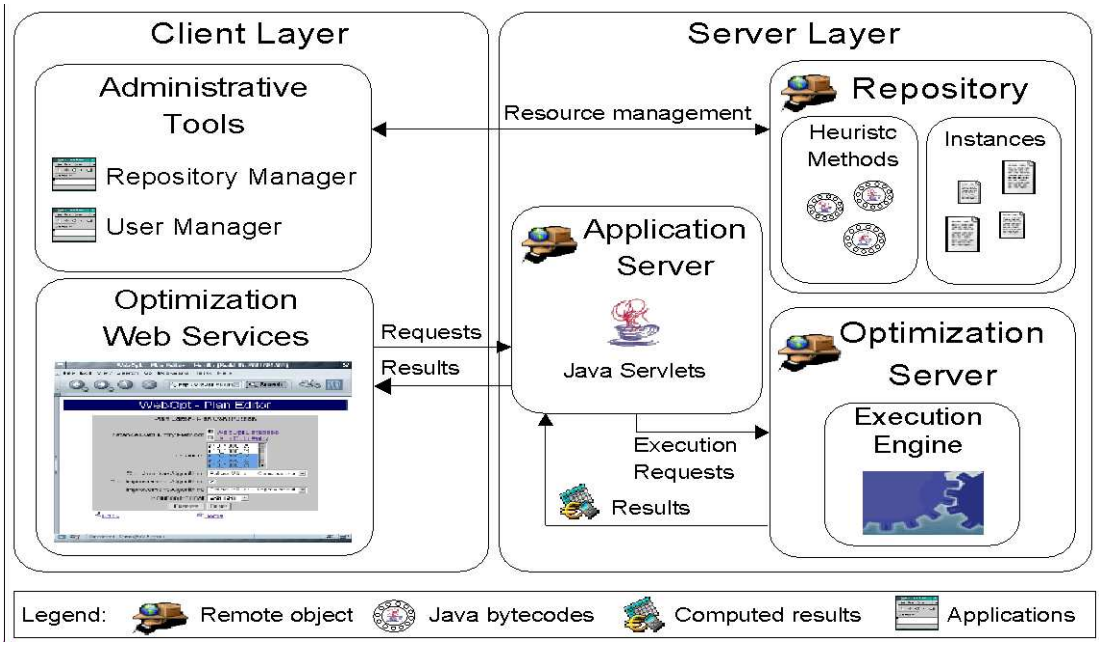


Figura 5.1: Arquitetura do ambiente *CORE*.

restringe o acesso aos recursos administrativos do ambiente. Ainda na camada cliente, o **Editor de Planos** é a principal ferramenta de acesso aos serviços de otimização.

A maior parte dos módulos da camada servidora aceita diferentes tipos de acesso da camada cliente. Notavelmente as ferramentas administrativas foram desenvolvidas com a utilização de *Applets Java*. Por outro lado, ferramentas de acesso público como o **Editor de Planos**, foram desenvolvidas com o servidor de aplicações de código aberto Tomcat (http://jakarta.apache.org/tomcat), que implementa a API de *servlets* Java (http://java.sun.com/products/servlet/). Dessa forma, o Editor de Planos pode ser acessado através de qualquer navegador, sem a necessidade de *plugins* especiais.

O ambiente foi inteiramente escrito em Java, e a tecnologia de objetos distribuídos utilizada é RMI (*Remote Method Invocation*), que permite uma integração fácil das chamadas remotas no código da camada cliente. Além disso, o projeto da arquitetura RMI permite a configuração fina no modo de passagem de mensagens, permitindo a solução de possíveis gargalos (como o tempo de serialização de objetos) [MAT 2001].

Repositório: O repositório consiste em um catálogo de recursos de otimização combinatória. Os objetos armazenados no repositório são algoritmos, instâncias e usuários. Os algoritmos inseridos no repositório são ponteiros para classes concretas que estendem classes definidas por um *framework* [GAM 95] específico para o domínio da otimização combinatória. O código para a classe do algoritmo em questão pode residir em qualquer servidor da Internet. Uma vez que os algoritmos vêm de fontes heterogêneas, existe um processo automático de validação que confirma se um dado arquivo compilado de uma classe Java realmente herda das classes definidas pelo *framework* e implementa os métodos necessários para a sua execução.

O repositório é um módulo chave no ambiente, uma vez que permite que grupos de desenvolvimento independentes incluam recursos no ambiente sem a necessidade de uma administração centralizada, ao mesmo que define um nível mínimo de conformidade

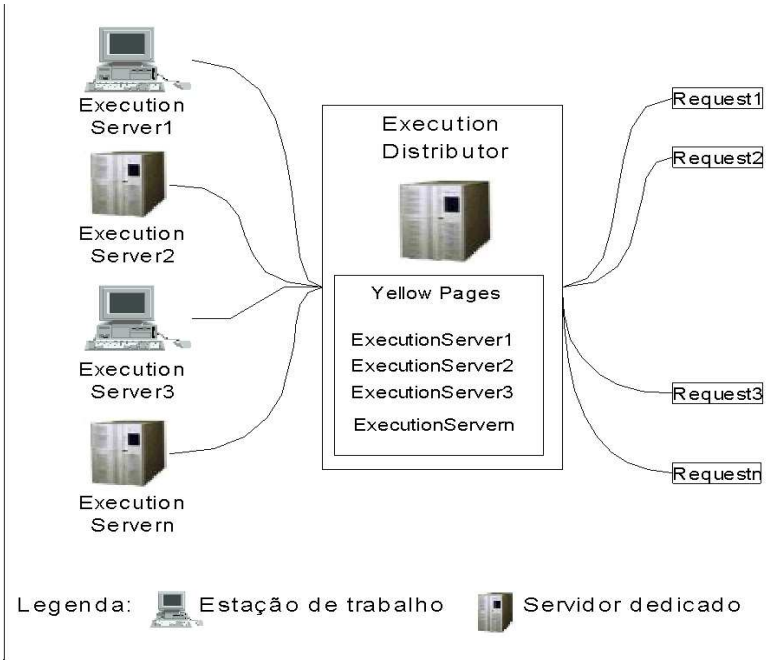


Figura 5.2: Modelo de distribuição de execuções.

dos métodos disponíveis em relação ao *framework* desenvolvido.

Servidor de Otimização: O Servidor de otimização é responsável pela interpretação e execução de **Planos de Otimização** enviados por clientes. Nesse caso, um plano de otimização é uma estratégia de resolução (seleção de métodos e sua respectiva configuração de parâmetros) que serão executados sobre um conjunto de instâncias de um determinado problema. Planos de otimização podem ser simples combinações de heurísticas construtivas ou de melhoramento, como também estratégias metaheurísticas como *GRASP* [FEO 95] que, de uma maneira geral, aplicam estratégias mais complexas de coordenação sobre grupos de heurísticas simples.

O Servidor de Otimização é um objeto remoto, acessível através de interfaces remotas Java. O acesso de usuários se dá, geralmente, através de interfaces gráficas *web*. Entretanto, as interfaces de programação do servidor de otimização podem ser usadas para a comunicação com sistemas de informação, por exemplo, atuando como um componente distribuído de software.

Outro recurso do servidor de otimização é o mecanismo de distribuição de execuções (figura 5.2), que é implementado na classe *Execution Distributor*. Objetos dessa classe apresentam as mesmas interfaces que o Servidor de Otimização real. No entanto, objetos do tipo *Execution Distributor*, ao invés de realizarem a execução de planos, distribuem as requisições entre uma lista de servidores registrados dinamicamente. Servidores, dedicados ou não, podem oferecer seus recursos computacionais para o ambiente. Atualmente, a política de distribuição de tarefas é bastante simples, baseada em uma lista circular de servidores disponíveis, sendo que a requisição é sempre enviada para o próximo servidor disponível.

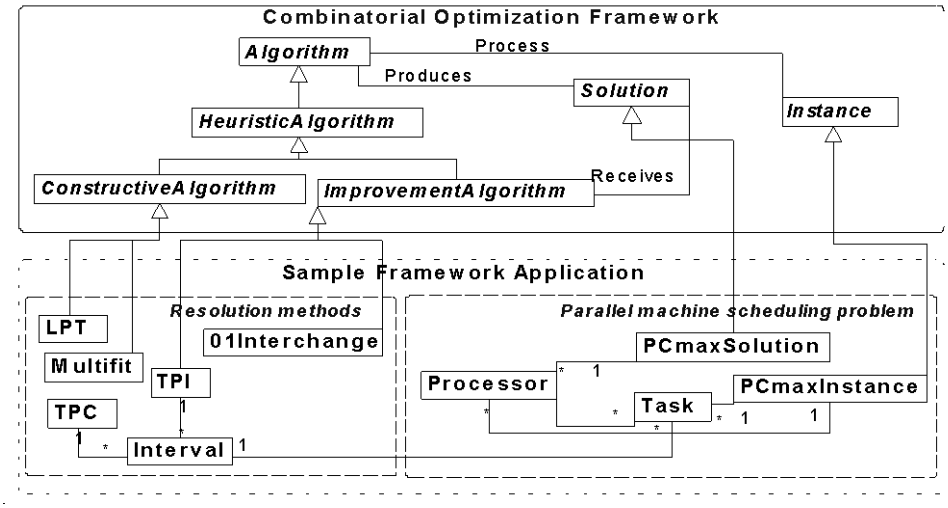


Figura 5.3: Framework para otimização combinatória e sua extensão para o P||Cmax.

5.2.4. O problema de seqüenciamento em máquinas paralelas: estudo de caso

Uma vez que o *framework* desenvolvido nesse projeto trata de aspectos que são independentes de problema e método de resolução, classes abstratas como *Instance*, *Solution*, *Constructive Algorithm* e *Improvement Algorithm* definem as bases para a construção de métodos heurísticos/metaheurísticos mais complexos (figura 5.3). A incorporação de novos métodos ao *framework* é feita de duas formas: primeiramente, através da extensão de classes do *framework* e da construção de métodos completamente orientados a objetos. Outra opção é o encapsulamento de métodos legados em classes especializadas. Enquanto a primeira opção tem a vantagem de um acoplamento seguro nos padrões definidos pelo *framework*, a segunda opção pode ser uma alternativa para o aumento do desempenho e/ou diminuição do tempo de incorporação de um método no *framework*, com o custo da perda da portabilidade do código compilado.

É importante ressaltar que em ambos os casos, o fluxo de execução é controlado pelo *framework*, de modo que a adição de um novo método não requer nenhuma alteração na estrutura do ambiente.

O exemplo que será descrito trata do processo envolvido na inserção de métodos heurísticos para o seqüenciamento em máquinas paralelas [BAK 95]. Esse problema é definido como a alocação de n tarefas em m processadores com igual produtividade, na tentativa de minimização do *makespan* (tempo total do início da execução até o momento em que todas as tarefas se completaram), sem preempção. Dessa forma, uma vez que uma tarefa é alocada em uma máquina, não pode ser removida até o final de seu processamento. Todas as máquinas iniciam o processo ao mesmo tempo. A classificação de 3 campos [LAW 89] define esse problema como P $\mid\mid$ C_{max}.

Os métodos inicialmente escolhidos foram: Algoritmo de Três Fases (TP) [FRA 94] *Longest Processing Time* (LPT) [GRA 79], Multifit [COF 78] e 0/1 Interchange [FIN 79]. Os algoritmos LPT e Multifit são heurísticas construtivas e o método 0/1 Interchange é um algoritmo de melhoramento. O algoritmo TP tem uma fase construtiva e duas de melhoramento, que foram implementados como dois métodos independentes: o primeiro

(TPC) com a parte construtiva e o segundo (TPI) com duas fases de melhoramento.

Como pode ser visto na figura 5.3, classes como *PCmaxInstance* e *PCmaxSolution* que especificam, respectivamente, os dados do problema e a solução, são comuns para todas as implementações dos algoritmos desenvolvidos para o P | | C_{max}. Uma vez que essas classes definem apenas métodos de acesso aos dados, sem especificar as estruturas de armazenamento, cada algoritmo pode utilizar as estruturas de dados mais apropriadas para o seu caso, sem comprometer a compatibilidade com os outros métodos. O algoritmo TP, por exemplo, utiliza uma estrutura de intervalos para facilitar as buscas nas fases de melhoramento.

Uma vez que as classes estejam codificadas nas especificações do *framework*, podem ser referenciadas pelo Repositório e utilizadas no Editor de Planos. Dessa forma, avaliações dos métodos disponíveis podem ser realizadas. Comparações que indiquem, por exemplo, quais as implicações em tempo de resolução/qualidade de solução existem em diferentes combinações de heurísticas podem ser feitas através do Editor de Planos. Nesse sentido, o ambiente *CORE* tem sido ferramenta de apoio ao Grupo de Otimização da UFSM (http://glover.ce.ufsm.br) para o desenvolvimento de trabalhos na área da otimização combinatória [MÜL 2002].

5.2.5. A metaheurística GRASP no ambiente CORE

Os métodos anteriormente descritos constituem heurísticas simples, determinísticas. Como já foi mostrado, a construção de um método *GRASP* exige a existência de um método semi-guloso e um algoritmo de melhoramento.

Como exemplo de heurística semi-gulosa, se utilizará o algoritmo *GRLPT*, disponível no ambiente *CORE*. O algoritmo *GRLPT (Greedy Randomized LPT)*, constitui uma extensão ao algoritmo *LPT* clássico, sendo que, a cada iteração, constrói uma lista (*RCL*) com os k processadores menos carregados, de acordo com o parâmetro *alpha*, e escolhe, de forma aleatória, entre esses para a alocação da tarefa.

Uma vez que uma instância do $P||Cmax$ consiste em uma lista de n tarefas com seus tempos de processamento J ($j_1, j_2, ..., j_n$) e o número de processadores m, o algoritmo *GRLPT* (algoritmo 4) , a cada iteração, verifica o custo $C(c_1, c_2, ..., c_m)$ que cada processador apresenta para a construção da *RCL*. Assim, considerando que o $P||Cmax$ é um problema de minimização, os limitantes para inclusão na *RCL* h^{max} e h^{min} são calculados utilizando-se o processador que apresentar maior e menor carga, respectivamente, de forma que h^{max} = *max*($C_{p=1..m}$) e h^{min} = *min*($C_{p=1..m}$).

Como retorno, obtém-se uma lista T (t_1, t_2, ..., t_n), onde t_i indica em qual processador a tarefa i foi alocada. Assim como no *LPT*, a lista J encontra-se ordenada de forma não crescente. O pseudocódigo para o algoritmo *GRLPT* é:

função GRLPT(J, n, m, α)
  faça $C_{p=1..m} \leftarrow 0$;
  para *i = 1, ..., n* **faça**
    h^{max} = *max* $\{C_{p=1..m}\}$;
    h^{min} = *min* $\{C_{p=1...m}\}$;
    RCL =$\{p \in C | C_p \leq h^{\min} + \alpha.(h^{\max} - h^{\min})\}$;
    p = RAND(RCL);
    $C_p \leftarrow C_p + t_i$;
    $T_i \leftarrow p$;

fim para;
retorne T;
fim GRLPT;

Algoritmo 4 - *Greedy Randomized LPT*

Uma vez codificado nos padrões do framework e validado pelo repositório, o algoritmo *GRLPT* pode constituir parte de um plano de otimização *GRASP*, de modo que sua execução pode ser requisitada através do Editor de Planos, como será visto a seguir.

A utilização da metaheurísticas *GRASP* no ambiente *CORE*, através da interface do **Editor de Planos** (figura 5.4), permite um ajuste fino dos parâmetros de execução. Na seção ***Resolution Method***, encontra-se a seleção do algoritmo construtivo (*Greedy Randomized Constructive*) e do método de busca local (*Improvement Algorithm*). O parâmetro alpha (*Alpha Value*), controla o grau de aleatoriedade das execuções do guloso construtivo. Pode-se selecionar também o número de iterações que serão processadas (*Iterations*). Como o *GRASP* é um método com componentes estocásticos, é importante que se observe a semente aleatória (*Random Seed*) enviada para o mesmo. Esse parâmetro permite também, a reprodução de resultados anteriormente obtidos.

A seção ***Presentation*** especifica os detalhes que devem ser incluídos nos resultados da execução e como esses serão exibidos. Pode-se optar pelo acompanhamento em tempo real das execuções através do navegador, ou selecionar o envio por e-mail dos resultados, quando todo o processo estiver concluído. É possível também, a conferência de todas as iterações realizadas pelo método, ou apenas aquelas que apresentaram melhora na qualidade da solução, através das opções de *Execution History*. Ainda, pode-se escolher entre a visualização completa da solução ou apenas a exibição do valor da função objetivo.

Os resultados da execução de um plano de otimização sobre um grupo de instâncias (figura 5.5) incluem informações relativas a resolução de cada instância, como o valor da função objetivo e o tempo de cada fase da heurística quanto informações sumarizadas do processo, como o desvio médio do valor ótimo de solução e o tempo médio de resolução.

As instâncias disponibilizadas no ambiente CORE representam a base comum de testes sobre a qual novos métodos de resolução são submetidos. Execuções de planos de otimização com variações nos parâmetros e combinações de diferentes métodos alimentam o repositório do sistema, indicando quais estratégias são mais indicadas para cada tipo de instância.

A figura 5.6 apresenta a janela de consulta de uma instância, apresentando o seu conteúdo, bem como o melhor custo de solução encontrado até o momento e o plano de otimização que o gerou.

5.2.6. Considerações sobre distribuição e paralelização de métodos no ambiente CORE

A arquitetura do ambiente *CORE* permite a utilização de recursos computacionais distribuídos e heterogênos para a execução de algoritmos. Atualmente, requisições de execução concomitantes são automaticamente distribuídas entre uma lista de computadores registrados e executadas em paralelo.

A inclusão de metaheurísticas como *GRASP* no ambiente *CORE* coloca a necessidade de um outro nível de granularidade na distribuição de tarefas. Uma vez que a

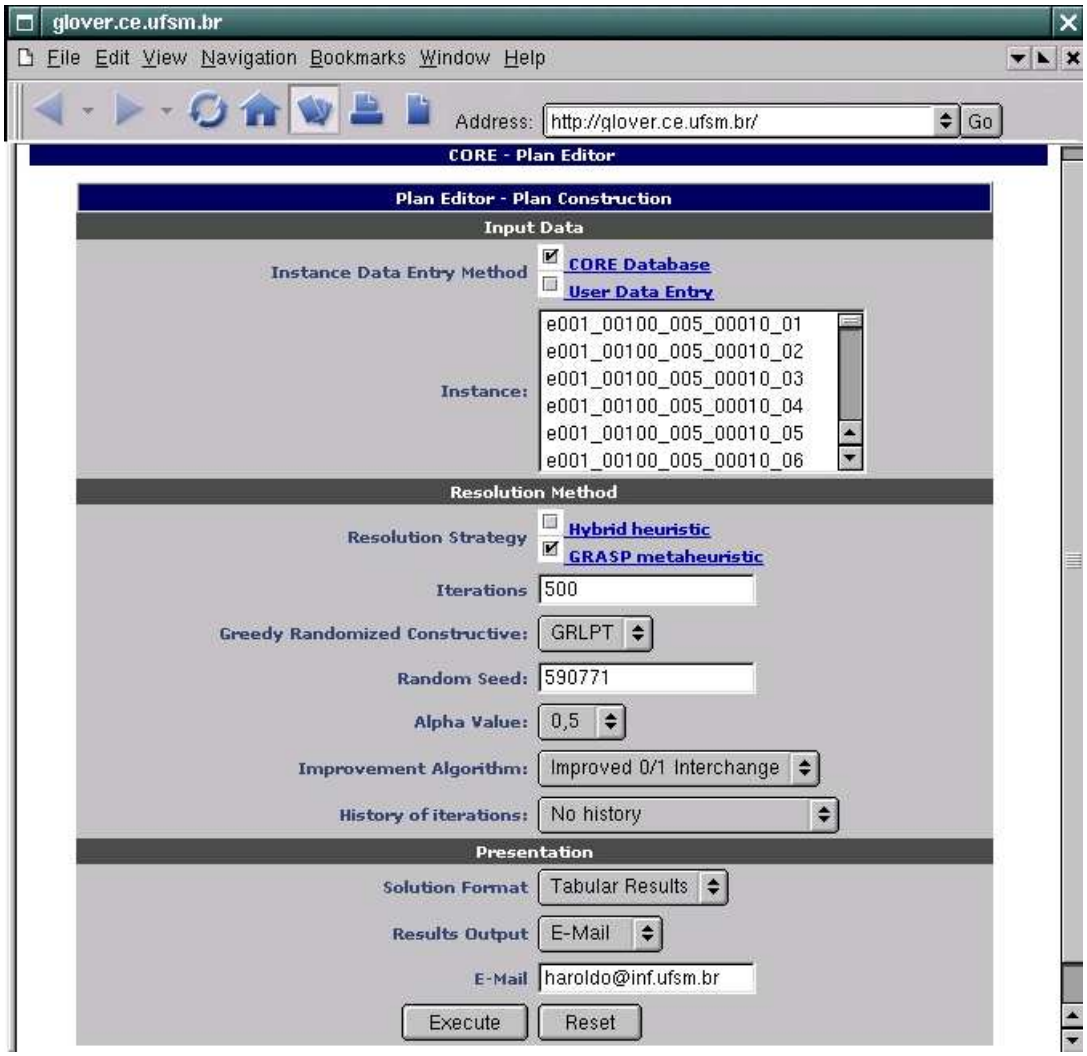


Figura 5.4: Construção de um plano de otimização GRASP.

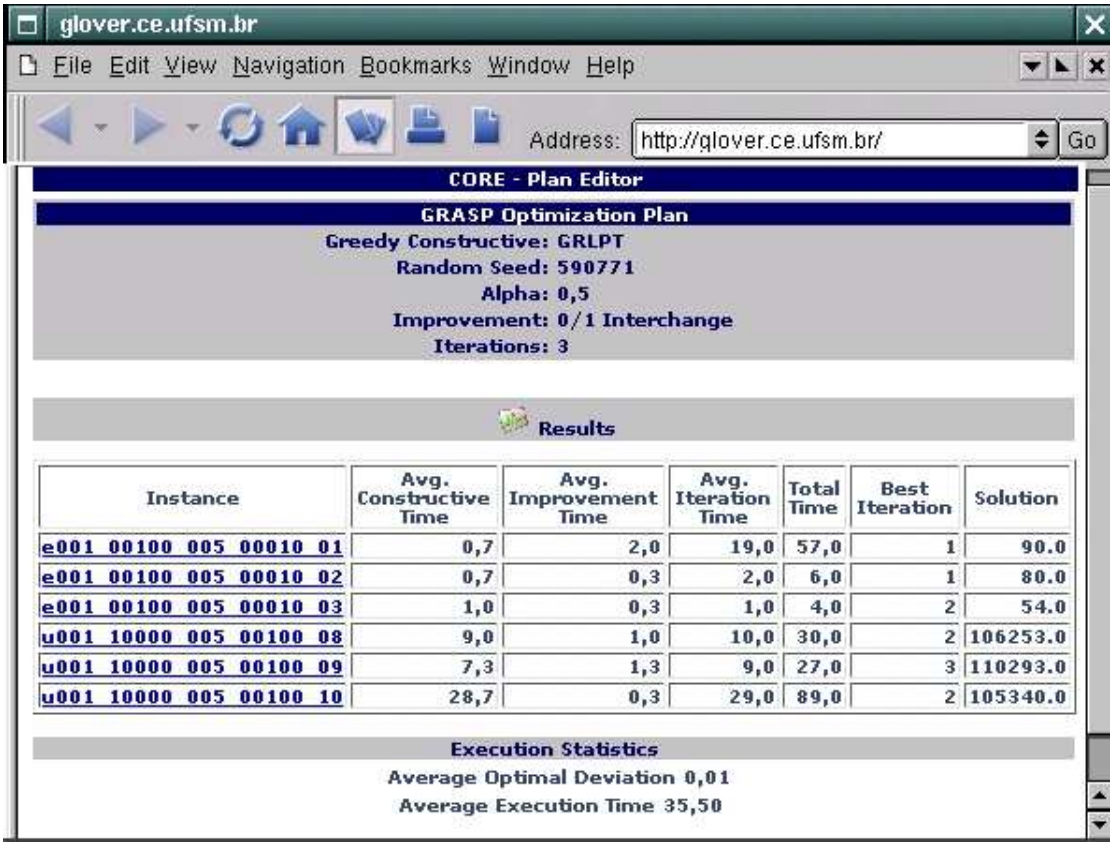


Figura 5.5: Visualização dos resultados de execução.

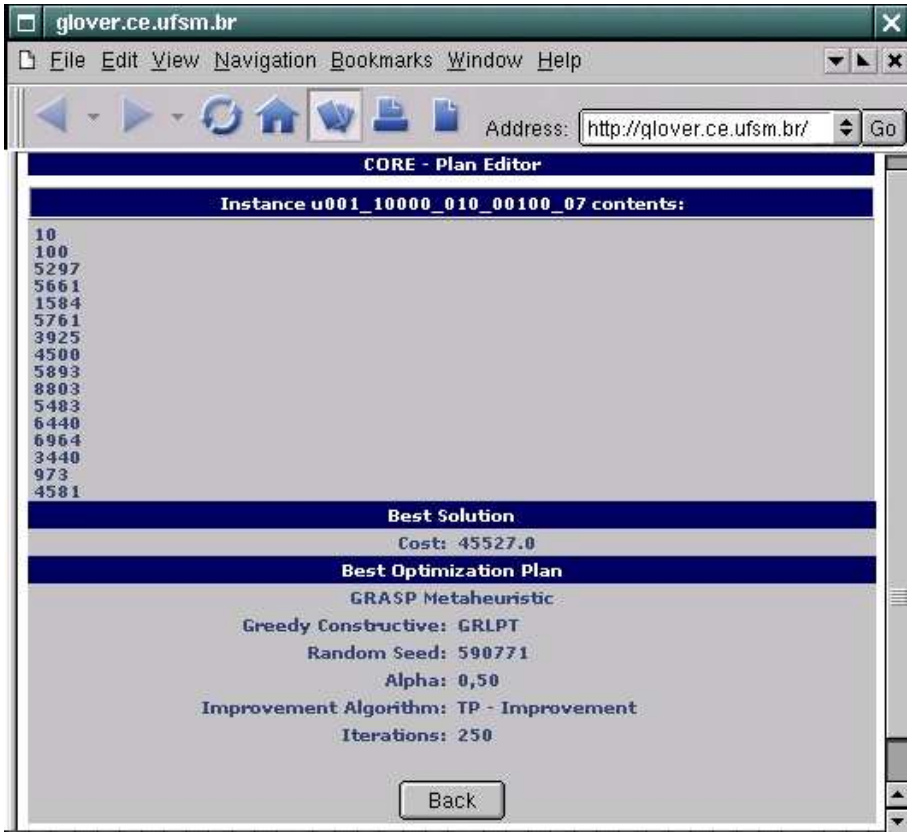


Figura 5.6: Visualização de informações relativas a uma instância.

idéia das metaheurísticas é explorar de maneira eficiente o espaço de soluções, uma única requisição pode demandar grande quantidade de processamento.

Vários trabalhos tratam dos aspectos de distribuição de metaheurísticas, sendo que a maioria ressalta o *speedup* praticamente linear que se obtém com a utilização da abordagem paralela, uma vez que grande parte dos métodos é trivialmente paralelizável [RIB 2001].

Em se tratando da paralelização de *GRASP* [RES 99] , duas estratégias gerais têm sido propostas: Na primeira se faz uma decomposição do espaço de busca, sendo que cada nodo recebe uma parte pré-construída da solução e se concentra em buscas que incluam aquele fragmento. A segunda opção é a distribuição de iterações. Nesse caso, é preciso certificar-se que a semente randômica de cada processador seja diferente para garantir a diversidade na busca.

Enquanto a primeira opção é fortemente dependente do problema a que se aplica, a segunda é genérica e pode ser diretamente implementada no ambiente *CORE*. Em qualquer um dos casos, um estudo sobre estratégias de divisão de carga que considerem recursos computacionais heterogêneos devem ser consideradas.

5.2.7. Conclusão

Os métodos de otimização combinatória são ferramenta imprescindível para o bom aproveitamento de recursos, tanto no ambiente científico quanto industrial. A crescente demanda por métodos de resolução mais abrangentes, que tratem de dos inúmeros aspec-

tos dos problemas encontrados no mundo real implica um desenvolvimento mais rápido de novos métodos, que apresentem maior robustez na busca de soluções.

O ambiente apresentado nesse trabalho facilita o desenvolvimento de métodos heurísticos/metaheurísticos, possibilitando também a realização de experimentos com a montagem de métodos híbridos e metaheurísticas. Uma vez que mesmo heurísticas simples de busca local podem apresentar complexidade exponencial, a utilização eficiente de recursos computacionais para sua execução é questão grande importância.

O projeto genérico do ambiente, independente de problema ou método de resolução permite, primeiramente, o desenvolvimento rápido de novos métodos, uma vez que calcula-se que 84% do código encontra-se em classes genéricas reaproveitáveis, ao mesmo tempo que permite que diferentes estratégias de paralelização sejam experimentadas sem mudanças no código do método de resolução. Estudos sobre quais estratégias são mais promissoras estão em andamento.

5.3. Problemas de Equações Diferenciais Parciais

A simulação numérica é atualmente uma ferramenta essencial para o desenvolvimento tecnológico de uma grande gama de disciplinas. A partir de modelos que descrevem objetos e fenômenos variados em física, biologia, economia, etc., as aplicações de simulação numérica permitem realizar experiências virtuais (auxiliadas por computador) em situações onde as experiências reais são impossíveis ou inviáveis.

Muitos fenômenos complexos são modelados matematicamente através de sistemas de Equações Diferenciais Parciais (EDP). Tratam-se de problemas de condições iniciais e de condições de contorno, definidos em domínios contínuos. A solução analítica de problemas de EDP é muitas vezes impossível, por isso apela-se para os métodos numéricos. Dado um problema de EDP, estes métodos permitem calcular uma solução aproximada das equações em um domínio discreto, isto é, representado por um número finito de pontos.

A discretização das equações resulta geralmente em um sistema matricial linear que precisa ser resolvido. Numericamente, a precisão de uma solução depende do número de pontos da discretização utilizada, o que influencia fortemente a ordem (tamanho) do sistema linear. No caso de simulações de grande escala, o custo de resolução deste sistema pode ser bastante elevado, ultrapassando a casa dos milhares de horas em computadores convencionais. As aplicações de simulação numérica são portanto grandes consumidoras de recursos computacionais, tanto em termos de tempo de processamento quanto em memória necessária para armazenamento dos dados envolvidos.

Neste contexto, sistemas paralelos e distribuídos são cada vez mais utilizados para obter uma diminuição substancial dos tempos de execução e/ou efetuar estudos numéricos de maior precisão mantendo tempos de processamento razoáveis. O desenvolvimento de aplicações de simulação eficientes para este tipo de plataforma não é uma tarefa simples, exigindo conhecimentos tanto em programação paralela quanto em cálculo numérico. Em particular, todo trabalho de paralelização precisa levar em conta as propriedades numéricas dos métodos empregados, que muitas vezes limitam o grau de paralelismo possível.

O restante desta parte do curso está organizado como segue. A seção 5.3.1. introduz alguns conceitos básicos envolvidos na resolução de problemas de EDP. A seção

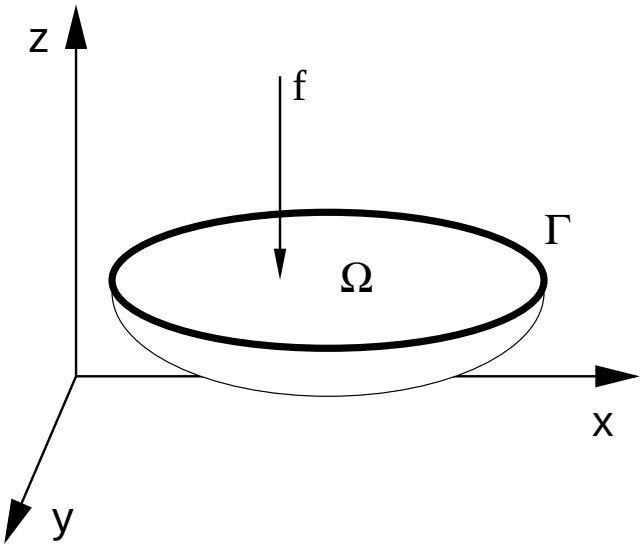


Figura 5.7: Membrana elástica submetida a uma força vertical.

5.3.2. aborda a resolução de EDP em arquiteturas paralelas com memória distribuída. São apresentadas diferentes combinações de soluções paralelas e métodos numéricos, bem como algumas ferramentas (bibliotecas, ambientes para resolução de problemas, etc.) que implementam tais soluções.

5.3.1. Resolução numérica de EDP

Equações diferenciais parciais são equações cujas incógnitas são funções de uma ou mais variáveis. Um exemplo clássico de EDP é a equação de Poisson, que envolve o operador diferencial de Laplace Δ. Em dimensão 2, esta equação se escreve:

$$-\Delta u(x,y) := -\frac{\partial^2 u}{\partial x^2}(x,y) - \frac{\partial^2 u}{\partial y^2}(x,y) = f(x,y)$$

Vários fenômenos físicos podem ser modelados através desta equação. Um problema comumente citado[LUC 96] consiste em determinar os deslocamentos verticais $u(x,y)$ de uma membrana elástica fixa pela borda e submetida a uma força vertical dada por $f(x,y)$ (veja figura 5.7). O domínio de definição da EDP (membrana) e a fronteira (borda) deste domínio são respectivamente denotados por Ω e Γ. A condição de contorno para este problema é $u = 0$ em Γ, significando que a borda de Ω não pode se deslocar.

Qualquer que seja o problema de EDP, seu tratamento computacional passa pela **discretização** das equações no domínio em que estão definidas. Entre os métodos discretos mais populares estão os métodos de diferenças finitas (MDF) e elementos finitos (MEF). Ambos os métodos requerem a definição de uma malha (*mesh* ou *grid*) de pontos (nós) que recobrem o domínio de definição do problema. A solução da EDP é aproximada em cada um dos N nós da malha, resultando geralmente em um sistema de N equações e N incógnitas que pode ser escrito sob forma matricial. De maneira geral, solução em cada nó depende da solução em nós "vizinhos" na malha, portanto o sistema é caracterizado por uma **matriz esparsa**. Esta descrição sucinta coloca em evidência as principais características comuns entre os métodos MDF e MEF.

Uma das principais diferenças entre os dois métodos reside em torno das técnicas de aproximação utilizadas para obter as equações discretas (tais detalhes matemáticos são abordados por exemplo em [DOU 2003]). As vantagens e desvantagens de cada método estão relacionadas a estas técnicas. O método MDF usa aproximações mais ”simples”, sendo portanto de mais fácil compreensão e implementação. Trata-se de um método amplamente utilizado para vários problemas de EDP, particularmente eficiente para domínios com geometria regular. Seu emprego é mais difícil no caso de problemas em domínios irregulares, quando geralmente prefere-se o método de elementos finitos. De fato, as principais aplicações do MEF são para problemas de cálculo estrutural e mecânica computacional, onde é comum o uso de geometrias complexas. As malhas empregadas têm geralmente estrutura irregular, o que dificulta sua geração e manipulação no contexto de uma simulação. Esta questão é aprofundada na seção 5.3.1.1. mais adiante.

A implementação da etapa de discretização exige um bom conhecimento do problema e dos métodos MDF ou MEF. Trata-se de uma etapa realizada tipicamente durante a inicialização das simulações. A fase de discretização é pouco onerosa computacionalmente, representando geralmente uma pequena parcela do tempo de processamento de uma simulação de larga escala. Para aplicações ”realísticas”, o cálculo de maior custo computacional é a resolução do sistema matricial obtido na fase de discretização. Existem vários métodos numéricos para a resolução destes sistemas, todos com complexidade expressa em função do número N de incógnitas do sistema. Este número depende do domínio de definição do problema, da discretização utilizada e da precisão desejada, chegando facilmente à casa dos milhões de nós para simulações de média a grande escala.

5.3.1.1. Discretização do domínio físico: geração de malhas

Uma malha é fundamentalmente representada por uma lista de nós e uma lista que expressa a interconexão entre estes nós. Segundo a topologia de interconexão, as malhas podem ser classificadas como estruturadas ou não-estruturadas[GEO 91]. Nas malhas **estruturadas**, o esquema de interconexão é tal que basta conhecer o índice do ponto para se saber sua posição na malha. Este é o caso, por exemplo, das discretizações quadrangulares quando se trabalha em 2 dimensões. Em uma tal malha, um ponto (i, j) tem por vizinho à “esquerda” o ponto $((i-1), j)$ e por vizinho à “direita” o ponto $((i+1), j)$. Nas malhas **não-estruturadas**, por outro lado, o esquema de interconexão dos pontos pode ser de qualquer tipo. As malhas não-estruturadas são portanto mais genéricas, permitindo o recobrimento de domínios com geometria complexa.

O método de elementos finitos se presta bem à utilização de malhas não-estruturadas, enquanto as malhas estruturadas são mais frequentemente associadas aos métodos de discretização de tipo diferenças finitas. A figura 5.8 fornece um exemplo simples que ilustra as diferenças entre malhas estruturadas e não-estruturadas.

A características das malhas são também ligadas ao problema físico a ser resolvido e às suas eventuais singularidades. Em consequência, a natureza do problema condiciona a escolha de malhas mais finas (i.e. com maior densidade de nós) em certas zonas do domínio a fim de obter soluções com precisão suficiente (veja exemplo na figura 5.3.1.1.). A geração de malhas adaptadas a cada problema não é uma operação simples, principalmente no caso de domínios de geometria irregular. Muitos trabalhos de pesquisa envolvem a concepção ou aperfeiçoamento de técnicas de geração automática de malhas. Para maiores detalhes sobre este assunto, sugere-se por exemplo [GEO 91].

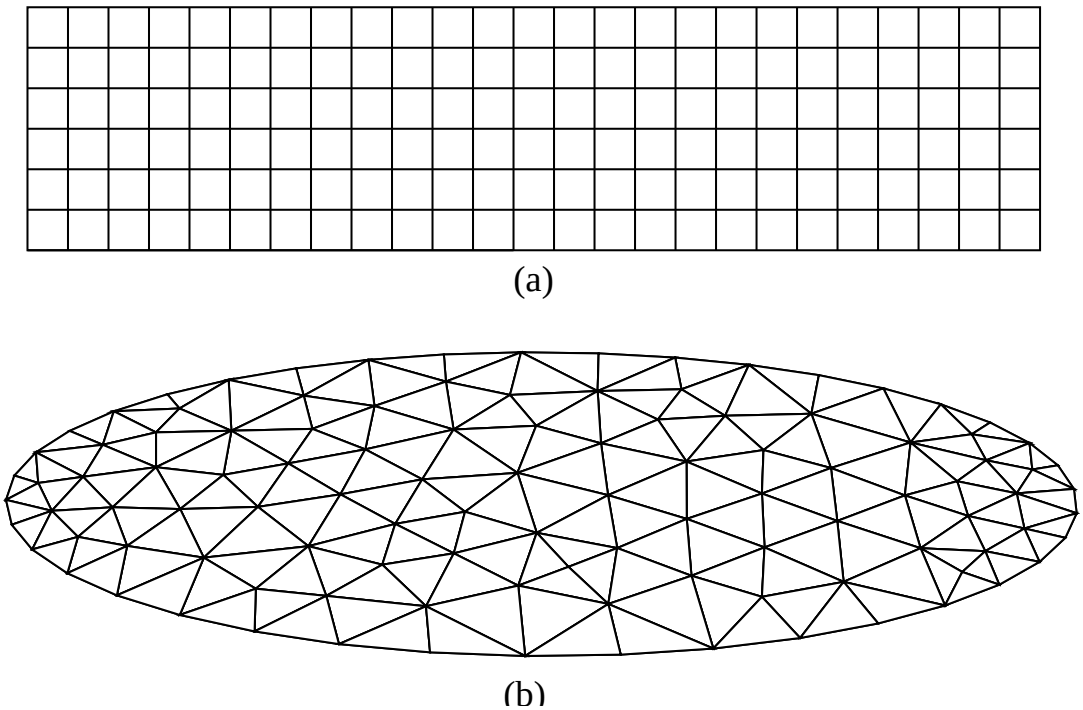


Figura 5.8: (a) Exemplo de malha estruturada. (b) Exemplo de malha não estruturada.

5.3.1.2. Métodos para resolução de sistemas de equações lineares

A resolução de um sistema linear $Ax = b$ pode ser efetuada através de vários métodos[GOL 96]. Geralmente, estes métodos são agrupados em duas categorias: métodos diretos e métodos iterativos.

Os métodos **diretos** são baseados em uma fatorização da matriz A permitindo chegar à solução exata do sistema em um número finito de operações. O método de eliminação de Gauss, um dos métodos diretos mais conhecidos, consiste em fatorizar a matriz A em um produto de duas matrizes L e U, onde L é uma matriz triangular inferior e U uma matriz triangular superior. Dada esta fatorização LU, a solução do sistema original $Ax = b$ requer a resolução em sequência de dois sistemas triangulares: $Ly = b$ e $Ux = y$. O método de Gauss é utilizado quando A é uma matriz quadrada sem estrutura especial. Quando A é simétrica, positiva e definida, a fatorização LU pode ser colocada na forma LL^T, onde L^T é a transposta da matriz L. Esta operação é conhecida como fatorização de Cholesky.

A principal vantagem dos métodos diretos está na sua precisão e estabilidade numérica. No entanto, a complexidade destes métodos, dada por $O(N^3)$ operações e $O(N^2)$ variáveis em memória, restringe seu emprego a sistemas de pequena ordem (em torno de $N \leq 5000$). Além disso, as fatorizações tendem a destruir a estrutura esparsa das matrizes, fazendo com que elementos não-nulos apareçam nas matrizes triangulares. Estas matrizes se tornam assim mais densas que a matriz original (processo conhecido como *fill-in*).

Os métodos **iterativos**, por sua vez, são baseados no cálculo de aproximações sucessivas da solução do sistema linear. A cada passo ou iteração calculam-se soluções mais precisas, que devem convergir para a solução exata partindo de uma solução inicial arbitrária. Os métodos iterativos do subespaço de Krylov[SAA 96] estão entre os mais

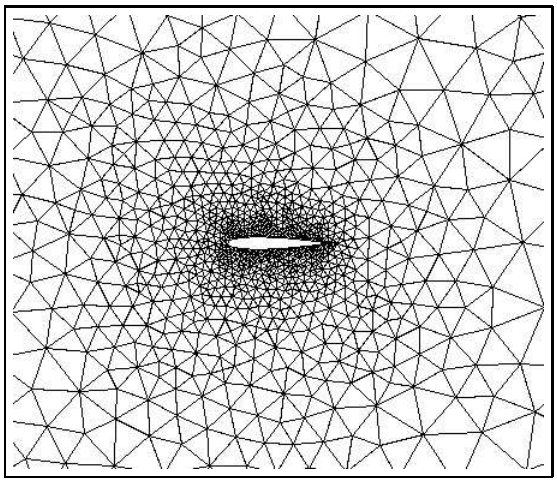


Figura 5.9: Malha em torno de um perfil de asa de avião.

utilizados atualmente. Fazem parte desta família, por exemplo, o método do Gradiente Conjugado (CG) e o método do Resíduo Mínimo Generalizado (GMRES). Uma vantagem dos métodos iterativos é que estes envolvem operações simples como produtos matriz-vetor, produtos vetoriais e produtos escalares. Teoricamente, estes métodos convergem em $O(N)$ iterações, mas na prática utilizam-se pré-condicionadores[4] para a matriz A a fim de acelerar a convergência. Estes métodos são em geral superiores aos métodos diretos para sistemas de grande porte ($N \geq 10000$). No entanto, a convergência dos métodos iterativos é altamente dependente da disponibilidade de um bom pré-condicionador.

A fim de reduzir o espaço em memória utilizado por estes métodos, muitas implementações exploram a estrutura esparsa da matriz para reduzir ao mínimo o número de coeficientes nulos armazenados. Existem inúmeros formatos de armazenamento de matrizes esparsas: em banda, em perfil, comprimido por linhas, por colunas, por diagonais, etc.

Embora os algoritmos existentes para resolução de sistemas lineares tenham evoluído e alcancem atualmente bons níveis de desempenho, o volume de dados armazenados é significativo e os tempos de resolução são longos quando o sistema é de grande porte. Nestes casos, faz-se apelo ao processamento paralelo seja agindo a nível matricial, seja agindo a nível da discretização matemática do problema.

5.3.2. Algoritmos paralelos para problemas de EDP

Existem basicamente duas abordagens principais para a resolução de EDP em paralelo. Na primeira, o enfoque é dado à paralelização da resolução dos grandes sistemas lineares oriundos da discretização por MDF, MEF, etc. Nesta abordagem, operações correntes em álgebra linear (produtos escalares, produtos matriz-vetor, fatorizações de matrizes, etc.) precisam ser modificadas para que o algoritmo de resolução funcione em arquitetura paralela.

A segunda abordagem consiste na utilização de métodos de decomposição de domínios. Estes métodos são baseados em um particionamento do domínio de cálculo em sub-domínios, permitindo substituir a resolução do sistema original por uma coleção

[4]Um pr´e-condicionador para uma matriz A ´e uma matriz "semelhante" a A e que pode substitu´ı-la no sistema $Ax = b$, tornando este sistema mais f´acil de ser resolvido.

de resoluções independentes em cada sub-domínio. Estas resoluções são entrelaçadas em um cálculo iterativo que determina as condições de contorno nas fronteiras entre sub-domínios.

É interessante notar que, na prática, estas duas abordagens são muitas vezes utilizadas de maneira complementar. Em particular, os métodos de decomposição de domínios são comumente usados como pré-condicionadores para acelerar a convergência de métodos iterativos de resolução de sistemas lineares. Uma apresentação mais detalhada destas duas abordagens é fornecida a seguir.

5.3.2.1. Paralelização da resolução de sistemas lineares

A paralelização da resolução de sistemas lineares é um campo bastante explorado, tendo resultado em algumas ferramentas e bibliotecas que facilitam grandemente a construção de simulações paralelas e distribuídas. Como o conjunto de técnicas numéricas é bastante vasto, a maioria destas bibliotecas são especializadas em métodos iterativos ou em métodos diretos.

No que concerne os métodos diretos, os esforços de paralelização se concentram na fatorização da matriz e do segundo membro, que representa a operação de maior custo computacional. Para maiores detalhes sobre a resolução de sistemas lineares em paralelo o leitor pode consultar o artigo [HEA 91] e ao livro [DON 98]. As ferramentas que oferecem métodos diretos paralelos são em geral mais recentes que as ferramentas centradas nos métodos iterativos. Como exemplo deste tipo de ferramenta podemos citar CAPSS[HEA 97], PaStiX[HEN 99] e PSPASES[GUP 97], para matrizes esparsas simétricas, definidas e positivas, e SuperLU[LI 99] e S+[FU 98] para sistemas não-simétricos. Uma comparação entre estas ferramentas é encontrada em [AME 2000]. A paralelização destes métodos não é abordada neste curso.

Quanto aos métodos iterativos, pode-se dizer que estes são relativamente fáceis de paralelizar pois as operações envolvidas se resumem basicamente a produtos matriz-vetor e produtos escalares. Dado um particionamento adequado dos dados envolvidos, estas operações são facilmente paralelizáveis. No entanto, a convergência destes algoritmos depende em grande parte de bons pré-condicionadores paralelos. A escalabilidade é um aspecto crítico para estes métodos uma vez que a velocidade de convergência decresce com o aumento do número de processadores. As ferramentas que oferecem este tipo de método são numerosas. Um panorama das principais bibliotecas para resolução iterativa de sistemas lineares pode ser encontrado em [EIJ 98]. Entre as principais ferramentas atualmente disponíveis podemos citar PETSc[BAL 97], PARMS[CAI 2002], Aztec[HUT 98], BlockSolve[JON 95] e ParPre[CHA 97].

Estas bibliotecas trabalham com uma representação distribuída do sistema linear a ser resolvido. Em geral, esta distribuição consiste em alocar um conjunto de linhas da matriz esparsa a cada processador (veja seção 5.3.2.2. a seguir). No que diz respeito à implementação paralela, a solução mais comum é baseada na utilização de MPI para a troca de mensagens nas arquiteturas com memória distribuída. A biblioteca SuperLU utiliza processos leves (*threads*) para as arquiteturas com memória compartilhada. A maioria destas ferramentas utiliza o padrão MPI para implementar as comunicações em arquiteturas com memória distribuída. As linguagens de programação que predominam são Fortran, C e C++.

5.3.2.2. Paralelização de métodos iterativos

As estratégias de paralelização de métodos iterativos se baseiam geralmente em uma distribuição do sistema de equações lineares sobre um conjunto de processadores. Uma abordagem para a solução deste problema é apresentada em [SAA 95] e utilizada em uma biblioteca precursora da ferramenta PARMS. A descrição a seguir se baseia nestas referências.

Representação distribuída de matrizes e vetores A representação distribuída de um sistema $Ax = b$ tem geralmente origem no particionamento do grafo de adjacência associado à matriz do sistema. Dada uma matriz esparsa A de dimensão $N \times N$, o grafo $G = (V, E)$ associado a esta matriz é definido por

- um conjunto de nós $V = \{1, 2, ..., N\}$, cada nó correspondendo a uma incógnita do sistema;
- um conjunto de arestas $E = \{(i, j) \mid a_{ij} \neq 0\}$ representando as dependências entre as equações do sistema.

Este grafo é não-orientado se a matriz A é simétrica. Neste caso as arestas (i, j) coincide com as arestas (j, i). A figura 5.10 ilustra esta dualidade entre a representação matricial de um sistema com 12 incógnitas e o grafo de adjacência associado.

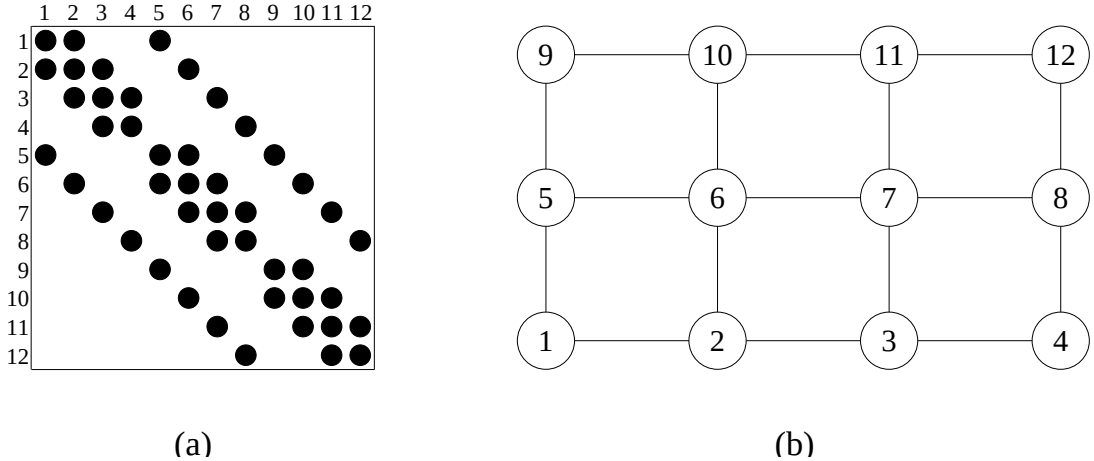


Figura 5.10: (a) Perfil de uma matriz esparsa de dimensão 12x12. (b) Grafo de adjacência associado.

Dado um particionamento do conjunto V em p sub-conjuntos $V_1, ..., V_p$, o grafo G se decompõe em p sub-grafos $G_i = (V_i, E_i)$. Os sub-conjuntos E_i expressam as dependências dos nós pertencentes a V_i em relação a outros nós inclusos ou não em V_i.

Geralmente, cada sub-grafo é alocado a um processador distinto, sendo que p designa ao mesmo tempo o número de sub-grafos e o número de processadores. Esta distribuição do grafo é equivalente à alocação de um conjunto de equações, i.e., de linhas do sistema matricial, a cada processador.

Em uma partição (sub-grafo) G_i pode-se identificar os seguintes conjuntos de nós:

- nós internos associados ao sub-grafo G_i (ou ao processador i): nós exclusivamente conectados a outros nós pertencentes a G_i;

- nós de interface locais: nós conectados a nós pertencentes a G_i e também a outros sub-grafos;
- nós de interface externos: nós pertencentes a outros sub-grafos mas que são conectados a nós pertencentes a G_i.

A identificação destes conjuntos de nós permite preparar as estruturas de dados locais para as comunicações necessárias ao longo do cálculo iterativo. De fato, as matrizes e vetores locais são construídos através da permutação das equações afetadas a cada processador, de maneira a separar os nós internos dos nós de interface locais. Os nós de interface externos são igualmente levados em conta nestas estruturas.

Globalmente, a matriz esparsa distribuída tem a forma apresentada na figura 5.11. Distingue-se nesta figura a matriz retangular A^1_{loc} de dimensão $N_1 \times N$ representando as N_1 equações (ou nós) afetadas ao processador P_1. Esta matriz é composta pelos seguintes blocos:

- B^1_{loc}: uma matriz quadrada de dimensão $N_1 \times N_1$. Os elementos a_{ij} deste bloco são tais que j corresponde a uma variável local, isto é, a um nó afetado ao processador i. Em B^1_{loc} as linhas pontilhadas separam os nós internos a G_1 dos nós de interface locais de G_1 ;
- B^1_{ext}: um bloco de elementos a_{ij} tais que j corresponde a uma variável externa.

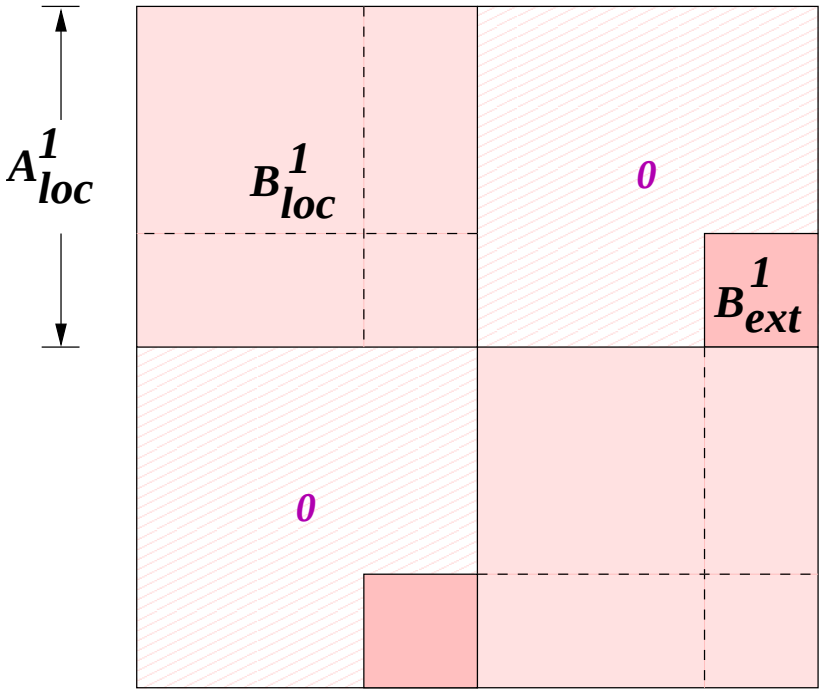


Figura 5.11: Visão global de uma matriz esparsa distribuída.

Os vetores associados ao processo de resolução (segundo membro b, vetor de solução x, etc.) são particionados e organizados em conformidade com a matriz. Estruturas de dados adicionais, em particular uma lista de processadores vizinhos e uma lista de nós de interface locais, também são geralmente empregadas para gerenciar as comunicações.

Algoritmos paralelos Em métodos iterativos do subespaço de Krylov (CG ou GMRES, por exemplo), os cálculos se resumem a combinações lineares de vetores, produtos escalares e produtos matriz-vetor.

As combinações lineares de vetores distribuídos são operações totalmente paralelas, que não necessitam de comunicações. Por exemplo, uma soma de dois vetores distribuídos só necessita das representações locais destes vetores. No que concerne os produtos escalares distribuídos, o algoritmo consiste em efetuar o produto escalar dos vetores locais, seguido de uma soma global dos resultados locais. Quando se utiliza o padrão MPI para implementação das comunicações, esta operação geralmente é realizada através de uma primitiva de comunicação global tipo MPI_Reduce.

Os produtos matriz-vetor estão entre as operações mais onerosas que compõem os métodos iterativos. Estas operações só são ultrapassadas em quantidade de cálculo pela fase de pré-condicionamento da matriz. Dados uma matriz e um vetor distribuídos segundo o formato descrito no início desta seção, o algoritmo paralelo para o cálculo do produto matriz-vetor se implementa como segue:

1. multiplicação do bloco B_{loc} da matriz pelo vetor local;
2. envio dos componentes do vetor local correspondentes aos nós de interface locais;
3. recepção dos valores correspondentes aos nós de interface externos;
4. multiplicação do bloco B_{ext} da matriz pelo vetor de valores externos. O resultado obtido é adicionado ao valor obtido na primeira multiplicação.

As etapas 1, 2 e 3 deste algoritmo podem ser efetuadas simultaneamente com o auxílio de primitivas de comunicação assíncronas disponíveis em MPI. As comunicações são de tipo ponto-a-ponto, se estabelecendo somente entre processadores vizinhos. O cálculo de um pré-condicionador para a matriz distribuída é necessário para acelerar a convergência dos métodos iterativos. Os pré-condicionadores paralelos mais eficientes são geralmente baseados em métodos de decomposição de domínios.

5.3.2.3. Métodos de decomposição de domínios

Métodos de decomposição de domínios são geralmente utilizados quando o problema físico é resolvido em um espaço discreto de grande dimensão. Quando os problemas apresentam singularidades que precisam de um tratamento particular (cantos, fissuras, materiais diferentes, etc., conforme figura 5.12), utilizam-se malhas com grande número de pontos a fim de se obter soluções mais precisas.

Outra aplicação interessante dos métodos de decomposição de domínios são os problemas numéricos que envolvem acoplamento de modelos matemáticos sobre domínios naturalmente disjuntos. Neste caso, as equações a serem resolvidas diferem de um domínio a outro (oceano/atmosfera, fluido/estrutura, conforme sugere a figura 5.13).

O algoritmo geral dos métodos de decomposição de domínios segue o princípio clássico de "divisão e conquista". Isto conduz a uma paralelização natural, particularmente adaptada às arquiteturas paralelas com memória distribuída. É interessante notar que estes métodos podem ser mais rápidos que os métodos iterativos clássicos mesmo em arquiteturas não paralelas[ROU 95].

Todos os métodos de decomposição de domínio requerem um particionamento do espaço discreto (malha) em sub-domínios. A seção 5.3.2.4. fornece uma breve introdução

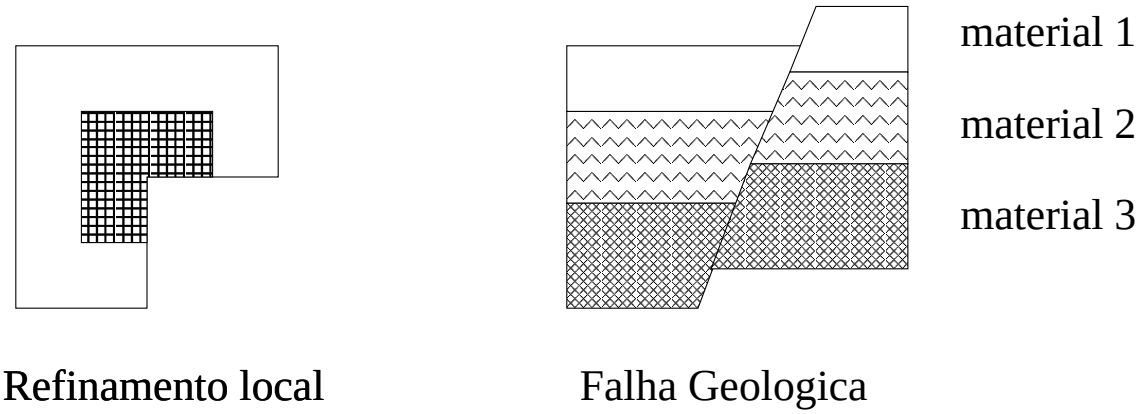


Figura 5.12: Singularidades nos domínios de definição de problemas..

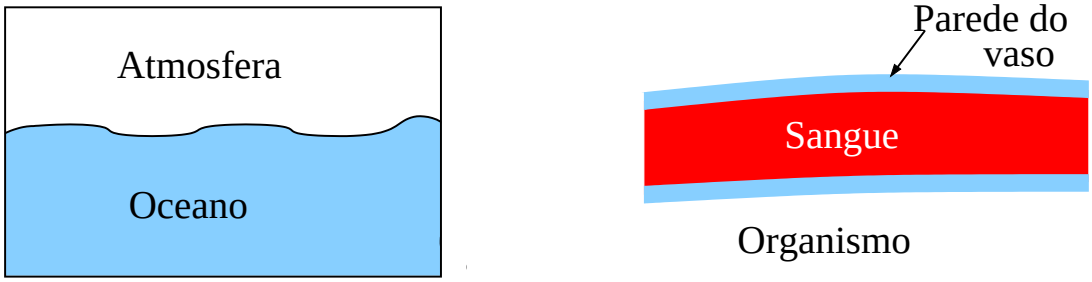


Figura 5.13: Acoplamento de modelos matemáticos..

a este assunto. Existem vários métodos numéricos de decomposição de domínios. A maioria dos autores classifica esta família de métodos em métodos com sobreposição (seção 5.3.2.5.) e métodos sem sobreposição (seção 5.3.2.6.).

5.3.2.4. Particionamento de malhas

O emprego de um método de decomposição de domínios requer o particionamento do domínio de cálculo. Este particionamento é guiado por critérios matemáticos, que exercem uma influência na convergência dos métodos, e por critérios computacionais, que influenciam no desempenho paralelo.

De acordo com a notação introduzida na seção 5.3.2.2., um sub-domínio se assemelha a um sub-grafo G_i. As ferramentas existentes são essencialmente baseadas em algoritmos que trabalham com as coordenadas geométricas da malha, ou em algoritmos que utilizam o grafo associado à matriz de adjacência da malha. Entre as ferramentas mais conhecidas podemos citar Metis[KAR 97], Scotch[PEL 96] e Chaco[HEN 95].

De maneira geral, é desejável que os algoritmos de particionamento respeitem as seguintes restrições geométricas:

- repartição igualitária do número de arestas, o que equivale a uma distribução de carga equilibrada;
- minimização do número de interfaces, o que equivale a minimizar o número de vizinhos de cada sub-domínio, e consequentemente o número comunicações durante

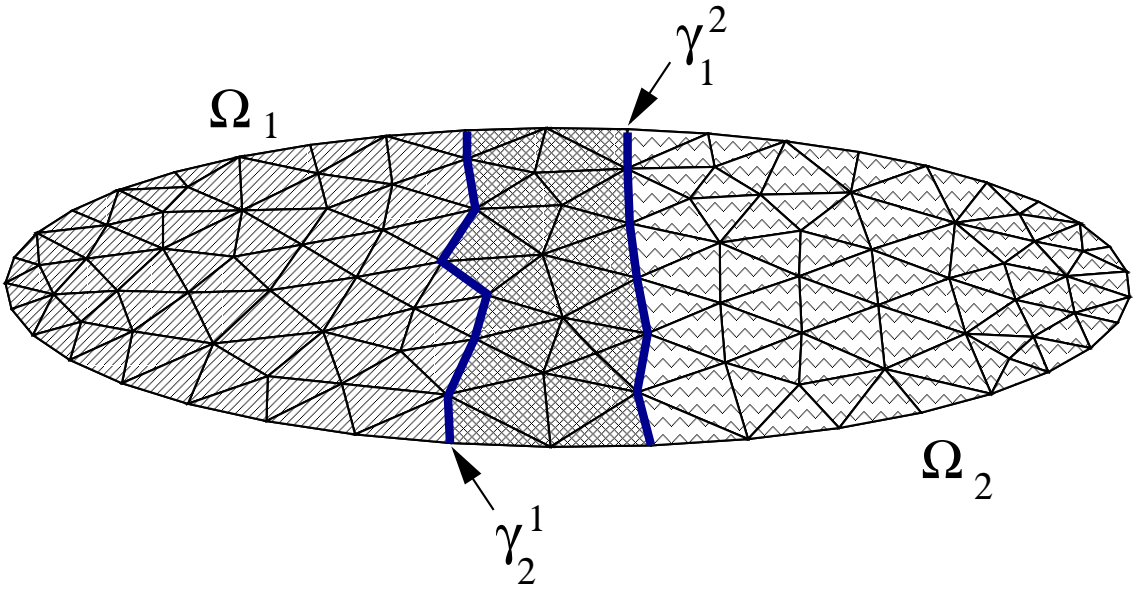


Figura 5.14: Decomposição com sobreposição.

a resolução paralela;

- minimização do número de arestas nas interfaces entre sub-domínios em relação ao número de arestas situadas no interior de cada sub-domínio.

Tais restrições conduzem geralmente a sub-domínios relativamente compactos, o que numericamente favorável à eficiência dos métodos de decomposição de domínios.

5.3.2.5. Métodos com sobreposição

Estes métodos se caracterizam pelo particionamento do espaço físico em um conjunto de domínios que se sobrepõem. Este curso se limita à apresentação do método de Schwarz e suas duas variantes: o método multiplicativo e o método aditivo. Para uma discussão mais aprofundada sobre estes métodos, recomenda-se a consulta a [SMI 96] et [CHA 94]. O funcionamento geral do método de Schwarz é apresentado a seguir considerando a decomposição de um domínio Ω em dois sub-domínios sobrepostos Ω_1 e Ω_2 (veja figura 5.14).

Seja Γ a fronteira do domínio Ω e γ_k^l as partes de fronteira de Ω_k ($k = 1, .., K$) incluídas nos sub-domínios Ω_l ($l = 1, .., K, l \neq k$). O método de Schwarz é um método iterativo que consiste em resolver os problemas de EDP de maneira alternada em cada sub-domínio, sendo que os valores calculados em Ω_l são utilizados como condições de contorno impostas sobre a fronteira γ_k^l de Ω_k na iteração seguinte. A iteração escolhida para atualizar as condições de contorno determina as variantes deste método.

Método de Schwarz multiplicativo Sejam u_1^0 e u_2^0 os valores iniciais da solução u_0, respectivamente nos sub-domínios Ω_1 e Ω_2, e f_1 e f_2 as decomposições correspondentes do segundo membro f. Considerando estas notações, e tomando como exemplo o problema de Poisson, o método multiplicativo consiste em calcular u_1^n e u_2^n de maneira que,

a cada iteração n, resolve-se o problema

$$\left\{\begin{array}{rcll} -\Delta u_1^n & = & f & \text{em } \Omega_1, \\ u_1^n & = & 0 & \text{sobre } \Gamma_1, \text{ (condição de contorno inicial)} \\ u_1^n & = & u_2^{n-1} & \text{sobre } \gamma_1^2 \end{array}\right.$$

para u_1^n, e em seguida resolve-se

$$\left\{\begin{array}{rcll} -\Delta u_2^n & = & f & \text{em } \Omega_2, \\ u_2^n & = & 0 & \text{sobre } \Gamma_2, \text{ (condição de contorno inicial)} \\ u_2^n & = & u_1^n & \text{sobre } \gamma_2^1. \end{array}\right.$$

para u_2^n.

Nota-se que este método utiliza as soluções u_l^n recém calculadas para determinar u_k^n ($l = k$), ou seja, existe uma sequência que deve ser respeitada neste cálculo: primeiro um sub-domínio, depois outro. A figura 5.15 ilustra a execução deste algoritmo para um problema de Poisson em dimensão 1. Deduz-se que esta variante não é naturalmente adaptada ao processamento paralelo. Este defeito pode ser compensado utilizando-se, por exemplo, algoritmos de ordenação preto-vermelho (veja [WIL 99], página 322).

Método de Schwarz aditivo A inicialização da resolução para o método aditivo se dá conforme a seção precedente. Após esta inicialização, calcula-se as iterações sucessivas como segue:

$$\left\{\begin{array}{rrcll} & -\Delta u_k^n & = & f_k & \text{em } \Omega_k, \\ \forall k = 1, 2 & u_k^n & = & 0 & \text{sobre } \Gamma_k, \\ & u_k^n & = & u_l^{n-1} & \text{sobre } \gamma_k^l, \quad \forall l = 1, 2,\ l \neq k. \end{array}\right.$$

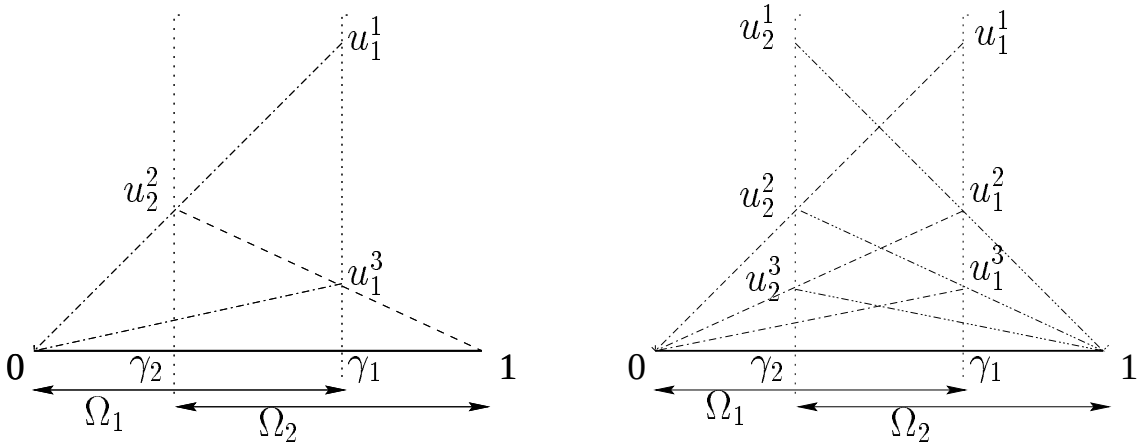
Este método é adaptado ao processamento paralelo pois as resoluções nos sub-domínios podem ser efetuadas simultaneamente. De fato, o cálculo da solução na iteração n só depende das soluções obtidas na iteração $n-1$. A figura 5.15 ilustra a execução deste algoritmo para o problema de Poisson em dimensão 1.

Convergência do método de Schwarz Os métodos de Schwarz multiplicativo e aditivo diferem em termos de velocidade de convergência. A seguir examina-se o comportamento de convergência destes métodos considerando-se a resolução do problema de Poisson em um domínio unidimensional:

$$\left\{\begin{array}{rcll} -\Delta u & = & 0 & \text{no intervalo }]0, 1[, \\ u(0) & = & 0, & \\ u(1) & = & 0. & \end{array}\right.$$

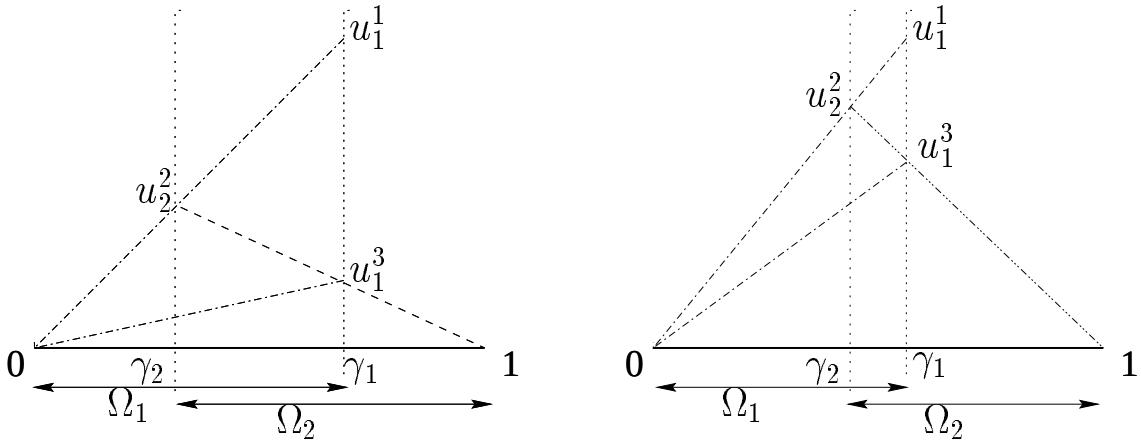
cuja solução trivial é $u = 0$. As soluções dos problemas locais (em cada sub-domínio) são igualmente segmentos de retas.

A figura 5.15 ilustra graficamente a convergência dos dois métodos para a solução exata do problema. Nota-se que o método aditivo converge mais lentamente que o método multiplicativo. De acordo com [SMI 96], na prática, para vários casos de decomposições em dois sub-domínios, o método de Schwarz aditivo exige o dobro do número de iterações para a convergência em relação ao método multiplicativo.



Schwarz multiplicativo Schwarz aditivo

Figura 5.15: Representação gráfica da convergência do método de Schwarz.



(a) (b)

Figura 5.16: Importância da sobreposição para o método de Schwarz. (a) Grande sobreposição. (b) Pequena sobreposição.

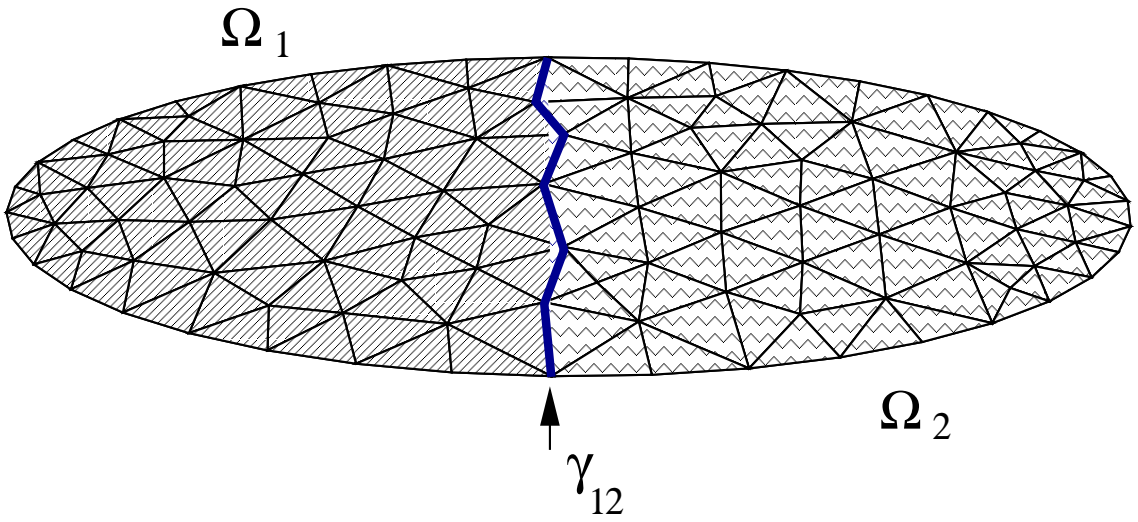


Figura 5.17: Decomposição sem sobreposição.

A velocidade de convergência destes métodos é também fortemente influenciada pela extensão da sobreposição. Através da representação gráfica da figura 5.16, verifica-se que os métodos com sobreposição convergem mais rápido quando a sobreposição é grande.

A sobreposição implica em uma redundância de cálculo pois alguns nós da malha são associados a vários sub-domínios. Deduz-se que o custo de cálculo dos problemas locais cresce com o tamanho da sobreposição. O uso destes métodos depende portanto de um bom compromisso entre a velocidade de convergência e a quantidade de cálculos redundantes. Também é importante notar que a velocidade de convergência diminui com o aumento do número de sub-domínios.

5.3.2.6. Métodos sem sobreposição

Estes métodos foram desenvolvidos durante os anos 70 para permitir a realização de cálculos envolvendo grandes estruturas. Os recursos computacionais disponíveis eram geralmente insuficientes, e os cientistas tiveram a idéia de efetuar os cálculos por sub-estruturas a fim de conseguir realizá-los com os computadores da época.

Em um método sem sobreposição, decompõe-se um domínio Ω em K sub-domínios disjuntos $\Omega_k (k = 1, .., K)$. Dois sub-domínios distintos compartilham apenas os nós de sua interface (se forem adjacentes), não compartilhando nenhum outro nó interior aos sub-domínios. Um exemplo de decomposição em dois sub-domínios é mostrado na figura 5.17. Esse exemplo será usado nas explicações que seguem.

Todos os métodos sem sobreposição levam à construção de um **sistema de interface**. Uma vez construído este sistema, pode-se resolvê-lo usando os métodos diretos ou iterativos clássicos. Esta seção apresenta um método sem sobreposição conhecido como método do complemento de Schur primal. Para mais detalhes a respeito deste e de outros métodos sem sobreposição, sugere-se consultar [FAR 94] e [SMI 96].

Método do complemento de Schur ou Schur primal Após o particionamento da malha, renumera-se as incógnitas do sistema de forma a isolar as variáveis associadas exclusivamente a cada um dos dois sub-domínios, bem como os nós na interface entre estes

sub-domínios. O sistema algébrico $Ax = b$ pode então ser escrito:

$$\begin{pmatrix} A_{11} & 0 & A_{13} \\ 0 & A_{22} & A_{23} \\ A_{31} & A_{32} & A_{33} \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} = \begin{pmatrix} b_1 \\ b_2 \\ b_3 \end{pmatrix}$$

Nesta decomposição em blocos, pode-se distinguir:

- os blocos diagonais A_{11} e A_{22}, que correspondem à matriz associada aos nós situados no interior de cada um dos dois sub-domínios.
- o bloco A_{33}, que corresponde à matriz associada aos nós situados na interface entre os dois sub-domínios.
- os demais blocos, que são transpostos entre si e representam as interações entre cada sub-domínio e a interface.

Esta forma de estruturar o sistema ressalta a inexistência de acoplamento direto entre as variáveis situadas em sub-domínios diferentes. Não é necessário construir o sistema completo, sendo que a resolução por sub-domínios pode usar as matrizes locais seguintes:

$$A_1 = \begin{pmatrix} A_{11} & A_{13} \\ A_{31} & A_{33}^{(1)} \end{pmatrix} \text{ et } A_2 = \begin{pmatrix} A_{22} & A_{23} \\ A_{32} & A_{33}^{(2)} \end{pmatrix}.$$

O bloco $A_{33}^{(1)}$ (respectivamente $A_{33}^{(2)}$) representa as interações entre os nós de interface e os nós correspondentes ao sub-domínio Ω_1 (respectivamente Ω_2). A soma destes dois blocos locais forma o bloco A_{33} do sistema global.

A partir desta reestruturação do sistema $Ax = b$, pode-se extrair um sistema correspondente aos nós de interface, dado por

$$Sx_3 = c,$$

onde

$$S = A_{33} - A_{31}A_{11}^{-1}A_{13} - A_{32}A_{22}^{-1}A_{23}$$

e

$$c = b_{33} - A_{31}A_{11}^{-1}b_1 - A_{32}A_{22}^{-1}b_2.$$

A matriz S é chamada **complemento de Schur** de A. Esta matriz condensada na interface é densa, e para construí-la é necessário inverter as matrizes dos sistemas correspondentes a cada sub-domínio. O custo desta operação é proibitivo para sistemas de grande porte.

Os sistemas correspondentes aos sub-domínios Ω_1 e Ω_2 podem ser resolvidos de maneira totalmente independente. Quanto à resolução do sistema de interface, prefere-se geralmente resolvê-lo sem calcular explicitamente o complemento de Schur S. Isto pode ser feito, por exemplo, através de um método iterativo de ponto fixo com pré-condicionamento $\tilde{S}$ da matriz S:

$$\tilde{S}(x_3^{n+1} - x_3^n) = c - Sx_3^n.$$

O algoritmo 1 apresenta um tal processo iterativo para a resolução de um sistema $Ax = b$ pelo método do complemento de Schur. Neste algoritmo, considera-se uma decomposição generalizada a K sub-domínios acoplados por uma interface global $\mathcal{I}$.

Algoritmo 1 Método de ponto fixo para o complemento de Schur em paralelo.

1: **para** $n = 1$ **até** a convergência **faça**
2: **para** $k = 1$ **até** K (número de sub-domínios) **faça [em paralelo]**
3: U_k^n solução de $A_{kk}U_k^n = F_k - A_{k\mathcal{I}}U_{\mathcal{I}}^n$
4: $Y_k^n = A_{\mathcal{I}k}U_k^n$
5: **fim para**
6: $\tilde{S}U_{\mathcal{I}}^n = F_{\mathcal{I}} - A_{\mathcal{I}\mathcal{I}}U_{\mathcal{I}}^{n-1} - \sum_{k=1}^{K} Y_k^n + \tilde{S}U_{\mathcal{I}}^{n-1}$
7: **fim para**

O cálculo do vetor $U_{\mathcal{I}}^n$ (linha 6 do algoritmo 1) pode ser feito por processador "mestre" ou por todos os processadores ao mesmo tempo. No primeiro caso, este cálculo requer comunicações entre o processador coordenador e os processadores que calculam cada um dos sub-domínios. No segundo caso, as comunicações ocorrem entre processadores que calculam sub-domínios tendo ao menos um nó em comum. Estas comunicações são imprescindíveis para garantir a continuidade da solução nas interfaces.

5.3.2.7. Resolução global por um método de Krylov

Os métodos de decomposição de domínios podem intervir na resolução de grandes problemas matriciais de duas maneiras. Na primeira, estes métodos levam à construção de um problema de interface. Neste caso, resolve-se os problemas locais com o auxílio de um método direto, por exemplo, e o problema de interface através de um método iterativo de ponto fixo ou um método de Krylov.

Na segunda, estes métodos podem ser utilizados como pré-condicionadores para um método de Krylov. Os dois casos se assemelham do ponto de vista da execução paralela. A seguir estuda-se a primeira alternativa utilizando o formalismo introduzido no parágrafo 5.3.2.6. para os métodos sem sobreposição.

Genericamente, estes métodos procuram resolver, no caso de 2 sub-domínios, o sistema

$$\begin{pmatrix} C_{11} & 0 & C_{13} \\ 0 & C_{22} & C_{23} \\ C_{31} & C_{32} & C_{33} \end{pmatrix} \begin{pmatrix} X_1 \\ X_2 \\ X_3 \end{pmatrix} = \begin{pmatrix} D_1 \\ D_2 \\ D_3 \end{pmatrix}$$

com o sistema condensado na interface

$$SX_3 = G$$

onde

$$S = C_{33} - \sum_{k=1}^{2} C_{3k}C_{kk}^{-1}C_{k3}$$

e

$$G = C_{33} - \sum_{k=1}^{2} C_{3k}C_{kk}^{-1}D_k.$$

O algoritmo 2 apresenta a resolução deste sistema pelo método do gradiente conjugado. Todas as operações que compõem este algoritmo utilizam matrizes e vetores locais.

O paralelismo do algoritmo está escondido no interior das operações algébricas. As instruções (4, 5, 10, 11 e 14), marcadas com um retângulo, necessitam de comunicações.

Algoritmo 2 : Algoritmo CG genérico para o sub-domínio $\Omega_k, k = 1, 2$.

1: $X_3^0 = 0$ {inicialização}
2: X_k solução de $C_{kk}X_k = D_k - C_{k3}X_3^0$
3: $\xi_k = C_{3k}X_k$
4: $\boxed{p^0 = r^0 = D_3 - C_{33}X_3^0 - \xi_1 - \xi_2}$ {comunicações}
5: $\boxed{\epsilon^0 = \|r^0\|^2}$ {comunicações globais}

6: **para** $i = 0, 1, ...$ **até** a convergência **faça**
7: z_k solução de $C_{kk}z_k = C_{k3}p^i$
8: $\xi_k = C_{3k}z_k$
9: $\xi_3 = C_{33}p^i$
10: $\boxed{\psi = \xi_3 - \xi_1 - \xi_2}$ {comunicações}
11: $\boxed{\alpha^i = \|r^i\|^2/(\psi, p^i)}$ {comunicações globais}
12: $X_3^{i+1} = X_3^i + \alpha^i p^i$
13: $r^{i+1} = r^i - \alpha^i \psi$
14: $\boxed{\beta^{i+1} = \|r^{i+1}\|/\|r^i\|^2}$ {comunicações globais}
15: $p^{i+1} = r^{i+1} - \beta^{i+1}p^i$
16: **fim para**

As outras instruções se efetuam de maneira local. As instruções 2 e 7 representam os cálculos locais sobre os sub-domínios. Estes calculos requerem a utilização de um método de resolução do sistema local. Os cálculos de interface estão presentes nas instruções 3–4 et 8–10. Estes produtos matriz-vetor necessitam de comunicações.

A verificação da convergência (ponto de sincronização global) é efetuada a nível da instrução 14, que necessita das normas calculadas nas instruções 5 e 14. Estes cálculos implicam em pontos de sincronização adicionais. O cálculo contido na instrução 11 necessita do cálculo de um produto escalar global.

Uma vez obtida a solução X_3 condensada na interface, resolve-se novamente os sistemas locais

$$C_{kk}X_k = D_k - C_{k3}X_3$$

a fim de calcular as soluções locais X_k.

5.3.3. Considerações finais

Os métodos de decomposição de domínios aliados aos métodos iterativos do subespaço de Krylov constituem uma solução bastante utilizada para a paralelização de problemas de EDP (veja por exemplo os trabalhos apresentados na edição da ERAD de 2002[ERA 2002], páginas 267 e 271). Resultados recentes de simulações utilizando tais métodos podem ser encontrados, por exemplo, em [CAI 2001, ABD 2001, CHA 2001].

Dada uma implementação paralela dos métodos descritos neste curso, o alcance de bons níveis de desempenho depende, entre outros fatores, de uma boa repartição da carga de trabalho entre os processadores disponíveis. Como o cálculo das condições de contorno implica em uma sincronização entre dois ou mais processadores, a presença de desequilíbrios de carga faz com que os processadores com menor carga fiquem ociosos esperando pelos seus “vizinhos” mais carregados, o que se traduz em uma redução

da eficiência paralela. Os algoritmos que visam a repartição de carga para os métodos de decomposição de domínios são geralmente eficazes para problemas simples. Entretanto, quando é necessário trabalhar com discretizações complexas tanto do ponto de vista geométrico quanto numérico, a distribuição igualitária da carga de trabalho é difícil de realizar.

No cenário atual do desenvolvimento de aplicações de simulação numérica, uma tendência é o uso de métodos discretos baseados em malhas adaptativas[VER 96]), isto é, malhas que são modificadas ao longo do cálculo de acordo com uma análise do erro de aproximação cometido. Esta técnica permite utilizar uma malha mais fina nas zonas do domínio onde a solução possui uma singularidade (choques ou turbilhões, por exemplo), obtendo resultados precisos com um número mínimo de incógnitas a calcular. O emprego de malhas adaptativas introduz mais um grau de dificuldade à programação paralela desta classe de aplicações, à medida em que a distribuição da carga de trabalho sobre um conjunto de processadores deve ocorrer de forma dinâmica ao longo da simulação.

5.4. Bibliografia

[ABD 2001] ABDALLAH, A. B. et al. Ahpik: a parallel multithreaded framework using adaptivity a nd domain decomposition methods for solving pde problems. In: INTERNATIONAL CONFERENCE ON DOMAIN DECOMPO SITION METHODS, 13., 2001, Lyon, France. **Proceedings...** [S.l.: s.n.], 2001.

[AME 2000] AMESTOY, P. R. et al. **Analysis, tuning and comparison of two general sparse solvers for distributed memory computers**. [S.l.]: ENSEEIHT-IRIT, 2000. (RT/APO/00/3).

[ARN 2001] ARNOLD, D. et al. **Users' guide to netsolve v1.4**. Disponível em: http://icl.cs.utk.edu/netsolve/. Acesso em: setembro 2001.

[BAK 95] BAKER, K. **Elements of sequencing and scheduling**. Hanover - NH: Amos Tuck School of Business Administration, 1995.

[BAL 97] BALAY, S.; MCINNES, W. G. L. C.; SMITH, B. Efficient management of parallelism in object-oriented numerical software libraries. In: MODERN SOFTWARE TOOLS IN SCIENTIFIC COMPUTING, 1997. **Anais...** Birkhauser Press, 1997.

[BEC 98] BECKER, P.; MÜLLER, S. Visual support for combining algorithms via the internet. In: IN PROCEEDINGS OF THE DATABASE AND EXPERT SYSTEMS APPLICATIONS, 9TH INTERNATIONAL CONFERENCE, VIENNA – AUSTRIA, 1998. **Anais...** [S.l.: s.n.], 1998.

[CAI 2001] CAI, X.; ØDEGåR, . On the performance of PC clusters in solving partial differential equations. In: TENTH SIAM CONFERENCE ON PARALLEL PROCESSING FOR SCIENTIFIC COMPUTING, 2001. **Proceedings...** [S.l.: s.n.], 2001.

[CAI 2002] CAI, X.; SAAD, Y.; SOSONKINA, M. Using the parallel algebraic recursive multilevel solver in modern physical applications. In: P. M. A. SLOOT C. J. K. TAN, J. J. D.; HOEKSTR, A. G. (Eds.). **Computational sciences - ICCS 2002**. [S.l.]: Springer-Verlag, 2002. p.345–355. (Lecture Notes in Computer Science, v.2330).

[CHA 94] CHAN, T. F.; MATHEW, T. P. Domain decomposition algorithms. In: **Acta numerica 1994**. [S.l.]: Cambridge University Press, 1994. p.61–143.

[CHA 97] CHAN, T.; V.EIJKHOUT. **Design of a library of parallel preconditioners**. [S.l.]: UCLA CAM, 1997. (97-58).

[CHA 2001] CHARãO, A. S. **Multiprogrammation parallèle générique des méthodes de décomposition de domaine**. 2001. Tese (Doutorado em Ciência da Computação) — Institut National Polytechnique de Grenoble.

[COF 78] COFFMAN, E.; GAREY, M.; JOHNSON, D. An application of bin-packing to multiprocessor scheduling. **SIAM Journal on Computing**, v.7, p.1–17, 1978.

[CZY 97] CZYZYK, J.; MESNIER, M.; MORE, J. **The Network-Enabled optimization system (NEOS) server**. Argonne, Illinois: Argonne National Laboratory, 1997. (MCS-P615-1096).

[DON 98] DONGARRA, J. J. et al. **Numerical linear algebra for high-performance computers**. [S.l.]: SIAM, 1998.

[DOU 2003] DOUGLAS, C. C.; HAASE, G.; LANGER, U. **A tutorial on elliptic pde solvers and their parallelization**. [S.l.]: SIAM (a ser publicado), 2003.

[EIJ 98] EIJKHOUT, V. Overview of iterative linear system solver packages. **NHSE Review**, v.3, n.1, 1998. http://www.nhse.org/NHSEreview/98-1.html.

[ERA 2002] ERAD2002 - ANAIS DA II ESCOLA REGIONAL DE ALTO DESEMPENHO, 2002. **Anais...** [S.l.: s.n.], 2002.

[FAR 94] FARHAT, C.; ROUX, F.-X. Implicit parallel processing in structural mechanics. **Computational Mechanics Advances**, v.2, p.1–124, 1994.

[FEO 95] FEO, T. A.; RESENDE, M. G. C. Greedy randomized adaptive search procedures. **Journal of Global Optimization**, v.6, p.109–133, 1995.

[FIN 79] FINN, G.; HOROWITZ, E. A linear time approximation algorithm for multiprocessor scheduling. **BIT**, v.19, p.312–320, 1979.

[FOU 2000] FOURER, R.; GOUX, J. Optimization as an internet resource. **Interfaces**, v.30, 2000.

[FRA 94] FRANÇA, P. et al. A composite heuristic for the identical parallel machine scheduling with minimum makespan objective. **Computers & Operations Research**, v.21, p.205–210, 1994.

[FU 98] FU, C.; JIAO, X.; YANG, T. Efficient sparse LU factorization with partial pivoting on distributed memory architectures. **IEEE Transactions on Parallel and Distributed Systems**, v.9, n.2, p.109–??, Feb. 1998.

[GAM 95] GAMMA, E. et al. **Design patterns**: elements od reusable object-oriented software. New York, NY: Addison-Wesley Publishing Company, 1995. (Addison-Wesley Professional Computing Series).

[GAR 79] GAREY, M.; JOHNSON, D. **Computers and intractability**. [S.l.]: W.H. Freeman and company, 1979.

[GEO 91] GEORGE, P. L. **Génération automatique de maillages. applications aux méthodes d'élements finis**. Paris: Masson, 1991. (Collection Recherches en Mathématiques Appliquées).

[GLO 97] GLOVER, F.; LAGUNA, M. **Tabu search**. Boston: Kluwer, 1997.

[GOL 89] GOLDBERG, D. **Genetic algorithms in search optimization & machine learning**. Menlo Park: Addison-Wesley, 1989.

[GOL 96] GOLUB, G. H.; LOAN, C. F. V. **Matrix computations**. 3.ed. Baltimore, MD, USA: The Johns Hopkins University Press, 1996. xxx + 698p. (Johns Hopkins Studies in the Mathematical Sciences).

[GRA 79] GRAHAM, R. L. et al. Optimization and approximation in deterministic sequencing and scheduling: a survey. **Discrete Mathematics**, v.5, p.287–326, 1979.

[GUP 97] GUPTA, A.; KARYPIS, G.; KUMAR, V. Highly scalable parallel algorithms for sparse matrix factorization. **IEEE Transactions on Parallel and Distributed Systems**, v.8, n.5, p.502–520, May 1997.

[HEA 91] HEATH, M. T.; NG, E.; PEYTON, B. W. Parallel algorithms for sparse linear systems. **SIAM Review**, v.33, n.3, p.420–460, 1991.

[HEA 97] HEATH, M. T.; RAGHAVAN, P. Performance of a fully parallel sparse solver. **The International Journal of Supercomputer Applications and High Performance Computing**, v.11, n.1, p.49–64, Spring 1997.

[HEN 95] HENDRICKSON, B.; LELAND, R. A multilevel algorithm for partitioning graphs. In: SUPERCOMPUTING 95, 1995. **Proceedings...** [S.l.: s.n.], 1995.

[HEN 99] HENON, P.; RAMET, P.; ROMAN, J. A mapping and scheduling algorithm for parallel sparse fan-in numerical factorization. In: EURO-PAR'99 PARALLEL PROCESSING, 1999. **Anais...** Springer-Verlag, 1999. n.1685, p.1059–1067. (Lecture Notes in Computer Science).

[HUT 98] HUTCHINSON, S. A. et al. **Aztec user's guide**: version 2.0. [S.l.]: Sandia National Laboratories, 1998.

[JON 95] JONES, M. T.; PLASSMANN, P. E. **BlockSolve95 users manual**: Scalabel library software for the solution of sparse linear systems. [S.l.]: Argonne National Lab., 1995. (ANL-95/48).

[KAR 97] KARYPIS, G.; KUMAR, V. A fast and high quality multilevel scheme for partitionning irregular graphs. **SIAM Journal of Scientific Computing**, 1997.

[KIR 83] KIRKPATRICK, S.; JR, C. G.; VECCHI, M. Optimization by simulated annealing. **Science**, v.2, p.63–74, 1983.

[LAW 89] LAWLER, E. et al. **Sequencing and scheduling**: algorithm and complexity. Amsterdan: Center for Mathematics and Computer Science, 1989. (BS-R8909).

[LI 99] LI, X. S.; DEMMEL, J. W. A scalable sparse direct solver using static pivoting. In: NINTH SIAM CONFERENCE ON PARALLEL PROCESSING FOR SCIENTIFIC COMPUTING, 1999. **Proceedings...** [S.l.: s.n.], 1999.

[LUC 96] LUCQUIN, B.; PIRONNEAU, O. **Introduction au calcul scientifique**. Paris: Masson, 1996.

[MAT 2001] MATHEW, A.; ROULO, M. **Accelerate your RMI programming**: speedup performance bottlenecks created by RMI. Disponível em: http://www.javaworld.com. Acesso em: março 2001.

[MÜL 2002] MÜLLER, F. et al. Avaliação de heurísticas para o $P||C_{max}$ através do ambiente de metacomputação CORE. In: XXII ENCONTRO NACIONAL DE ENGENHARIA DE PRODUÇÃO / ANAIS DE RESUMOS, 2002, Curitiba. **Anais...** [S.l.: s.n.], 2002.

[PEL 96] PELLEGRINI, F.; ROMAN, J. SCOTCH: A software package for static mapping by dual recursive bipartitioning of process and architecture graphs. In: LIDDELL, H. et al. (Eds.). **High-performance computing and networking (hpcn'96 europe)**. [S.l.]: Springer-Verlag, 1996. p.493–498. (Lecture Notes in Computer Science, v.1067).

[REI 91] REINELT, G. A traveling salesman problem library. **ORSA Journal on Computing**, v.3, p.376–384, 1991.

[RES 99] RESENDE, M. G.; PARDALOS, P. M.; EKSIOGLU, S. D. Parallel metaheuristics for combinatorial optimization. In: INTERNATIONAL SCHOOL ON ADVANCED ALGORITHMIC TECHNIQUES FOR PARALLEL COMPUTATIONS WITH APPLICATIONS, 1999, Natal (RN) Brazil. **Anais...** [S.l.: s.n.], 1999.

[RIB 2001] RIBEIRO, C. C.; HANSEN, P. Strategies for the parallel implementation of metaheuristics. **Essays and surveys in metaheuristics**, p.263–308, 2001.

[ROU 95] ROUX, F.-X. Méthodes de décomposition de domaine pour des problèmes elliptiques. **Calculateurs Parallèles**, v.7, n.3, p.237–253, 1995.

[SAA 95] SAAD, Y.; MALEVSKY, A. PSPARSLIB: A portable library of distributed memory sparse iterative solvers. In: PARALLEL COMPUTING TECHNOLOGIES (PaCT-95), 1995, St. Petersburg, Russia. **Proceedings...** [S.l.: s.n.], 1995. n.3rd International Conference.

[SAA 96] SAAD, Y. **Iterative methods for sparse linear systems**. Boston: PWS Publishing, 1996.

[SMI 96] SMITH, B.; BJORSTAD, P.; GROPP, W. **Domain decomposition**: parallel multilevel methods for elliptic partial differential equations. [S.l.]: Cambridge University Press, 1996.

[VER 96] VERFÜRTH, R. **A review of a posteriori error estimation and adaptive mesh-refinement techniques**. [S.l.]: Wiley and Teubner, 1996.

[WIL 99] WILKINSON, B.; ALLEN, M. **Parallel programming**: techniques and applications using networked workstations and parallel computers. [S.l.]: Prentice-Hall, 1999.

Sumário