3 Metaheurísticas Paralelas

Este capítulo aborda os principais tópicos a serem considerados na paralelização de metaheurísticas. O capítulo inicia apresentando a importância de paralelizar metaheurísticas. Em seguida, discute as mais comuns ferramentas usadas na implementação de algoritmos paralelos, dando ênfase ao ambiente de troca de mensagens MPI, adotado no desenvolvimento de todos os algoritmos implementados durante esta tese. A Seção 3.3 trata as estratégias de paralelização. A Seção 3.4 aborda os ambientes computacionais paralelos. Os modelos de algoritmos paralelos são vistos a seguir, na Seção 3.5. Por último, são mostradas as diversas arquiteturas de máquinas paralelas usadas na execução dos algoritmos.

3.1 Considerações Iniciais

A solução de problemas complexos requer a implementação de algoritmos eficientes. No contexto de problemas complexos, tais como aqueles pertencentes à classe dos NP-difíceis, a construção de bons algoritmos aproximados pode ser alcançada através do uso de metaheurísticas. Na prática, metaheurísticas são procedimentos de alto nível que coordenam heurísticas simples, objetivando explorar de forma eficiente o espaço de busca para encontrar boas soluções aproximadas (freqüentemente ótimas) para problemas de otimização combinatoria difíceis [105].

De fato, o uso de metaheurísticas freqüentemente oferece um prático padrão para solucionar problemas complexos de escala realista. Entretanto, por mais que as implementações das metaheurísticas sejam otimizadas e eficientes, há aplicações que requerem tempos computacionais grandes, excedendo o limite do que é considerado aceitável para a maioria das aplicações. Nesse caso, a computação paralela é fortemente recomendada pois pode acelerar a execução de algoritmos pesados através da distribuição de tarefas entre vários processadores.

O uso da computação paralela na implementação de metaheurísticas tem várias vantagens além da aceleração da busca por soluções de boa

qualidade. Cung et al. [46] revelam que, além da aceleração, a paralelização de metaheurísticas também torna as aplicações mais robustas, possibilitando a resolução de grandes problemas. Além disso, a disponibilidade de múltiplos processadores fornece mais oportunidade para a diversificação das estratégias de busca e sintonização dos parâmetros, ampliando assim a cobertura do espaço de busca e gerando mais soluções viáveis.

Todavia, o desempenho de um algoritmo paralelo depende de vários fatores chave, tais como a ferramenta para programação paralela empregada, os modelos de paralelismo implementados e a arquitetura das máquinas paralelas. Este capítulo apresentará nas próximas seções detalhes dos fatores que podem influenciar no desempenho de uma metaheurística paralela.

3.2

Ferramentas para Programação Paralela

Há muitas ferramentas de programação disponíveis para implementar programas paralelos, cada uma sendo mais apropriada para um tipo de problema específico ou arquitetura de máquina. Dessa forma, há basicamente três grupos de ferramentas usadas para implementar aplicações paralelas.

A maneira mais prática é usar um compilador paralelizável que automaticamente converte um programa seqüencial para um programa paralelo. Todavia, essa ferramenta trabalha só com granularidade fina, implicando que o programa paralelo seja dividido em várias tarefas leves, que seriam instruções ou pequenos grupos de simples operações. Nesse caso, o usuário não tem controle algum na definição das tarefas paralelas. Além disso, muitos desses compiladores são orientados para programas escritos em Fortran, embora existam algumas ferramentas para C e C++, tais como SUIF [140] e Portland [120].

Outra possibilidade para desenvolver programas paralelos é usar linguagens paralelas, que são essencialmente linguagens seqüenciais acrescidas de chamadas de sistemas especiais. Essas chamadas fornecem primitivas de baixo nível para troca de mensagem, criação e sincronização de processos. Entre essas, destacam-se: Ada [108], CilK [70], HPF [111], NESL [137], Linda [107] e Orca [23]. Tais linguagens são mais apropriadas para aplicações com granularidade fina ou média. Com essas linguagens, o usuário tem maior poder sobre a definição das tarefas paralelas. Todavia, cabe ao programador toda a responsabilidade de criação, sincronização e comunicação entre os processos paralelos.

O terceiro grupo, e também o mais popular, é formado por bibliotecas de troca de mensagem, que são acopladas às linguagens seqüenciais. Nesse caso, o usuário tem total domínio do grau de paralelismo implementado e maior

flexibilidade na construção das aplicações paralelas, sem precisar se preocupar com detalhes de baixo nível na troca de mensagem entre as tarefas paralelas. Como exemplos dessas bibliotecas há OpenMP [118], MPI [114] e PVM [54]. Atualmente, MPI e PVM são as bibliotecas de troca de mensagem mais usadas para desenvolver programas paralelos.

PVM foi pioneiro entre as bibliotecas de troca de mensagem, tornando-se por muitos anos o padrão de fato para esse tipo de ferramenta. O PVM é um pacote de *software* que permite que um grupo de computadores interconectados, possivelmente com arquiteturas heterogêneas, possa trabalhar cooperativamente formando uma máquina paralela virtual. Atualmente, a biblioteca mais comumente usada é o MPI, que define uma biblioteca de funções para implementar a troca de mensagens de maneira padronizada e portátil. A Seção 3.2.1 descreve as principais características da plataforma MPI.

3.2.1

MPI - Message Passing Interface

O MPI surgiu a partir de um fórum internacional [114] com o objetivo de tornar-se um padrão de direito para os ambientes de troca de mensagem. Além da padronização, o MPI almejava otimizar a comunicação e aumentar o desempenho de aplicações paralelas ou distribuídas. A eficiência e a generalidade do MPI são garantidas através da disponibilidade de um extenso padrão que oferece diversas implementações para uma mesma funcionalidade. Por exemplo, para o envio de mensagens há funções que implementam comunicação ponto-a-ponto, comunicação coletiva e reduções que implementam comunicação envolvendo múltiplos processos. Outra característica importante dessa biblioteca, no que se refere à eficiência na comunicação, é a possibilidade de haver sobreposição temporal de computação e comunicação.

Como dito anteriormente, atualmente o MPI é a biblioteca de passagem de mensagem mais popular. Essa popularidade ocorreu não apenas por ele ter sido desenvolvido para executar em uma variedade de sistemas, mas também por dar suporte ao gerenciamento de heterogeneidade (criação de novos tipos de dados), à construção de programas modulares (construção de comunicadores), ao gerenciamento de latência (operações assíncronas) e à representação de operações globais (operações coletivas). A primeira especificação do padrão MPI foi o MPI-1, formado por 128 funções para comunicação entre processos, comunicação em grupo, gerenciamento e interação com o ambiente. A segunda especificação, MPI-2, foi concluída com 200 funções a mais.

As duas especificações do MPI trabalham com os conceitos de grupos e comunicadores. Grupos em MPI definem uma coleção ordenada de processos,

cada um com um identificador (*rank*) distinto. Os grupos são responsáveis por definir os nomes de baixo nível para comunicação interprocesso. Eles podem ser manipulados de forma separada dos comunicadores, mas somente comunicadores podem ser usados em operações de comunicação. Os comunicadores são divididos em dois tipos: intracomunicadores e intercomunicadores. Intracomunicadores são usados para operações com um único grupo de processos, e intercomunicadores para comunicação entre dois grupos de processos.

Dessa forma, após a inicialização de uma aplicação MPI, um comunicador denominado *MPI_COMM_WORLD* é criado automaticamente, contendo todos os processos pertencentes à aplicação. Os processos recebem dentro desse comunicador *ranks* com valores distintos, inteiros e crescentes a partir de zero. Assim, através desse comunicador os processos de uma aplicação podem se comunicar diretamente entre si.

Na especificação do MPI-1, os grupos são estáticos, significando que nenhum novo processo pode ser acrescentado a um grupo após a sua criação. A biblioteca fornece operações capazes de unir e separar grupos, resultando sempre na criação de um novo grupo. Essa estabilidade é incompatível com a necessidade de gerenciamento dinâmico dos processos presentes nos ambientes dinâmicos e heterogêneos como os *grids* computacionais, onde se faz necessária a utilização de mecanismos de tolerância a falhas e escalonamentos dinâmicos de processos de uma aplicação.

Para resolver esse problema, o padrão MPI-2 foi projetado para permitir a criação e a finalização cooperativa de processos após uma aplicação MPI ter iniciado. Ele fornece um mecanismo para estabelecer comunicação entre os novos processos criados e a aplicação MPI existente. A função *MPI_Comm_spawn()* cria dinamicamente processos MPI e estabelece comunicação com eles, retornando um intercomunicador contendo o processo pai no grupo local e o processo filho no grupo remoto. Quando um dos processos disparados acaba, os demais processos são capazes de continuar a execução. Além de facilitar a identificação e o tratamento de falhas, a função *MPI_Comm_spawn()* facilita a construção de mecanismos de balanceamento dinâmico de carga, permitindo a escolha do processador onde a tarefa deverá ser disparada.

O padrão MPI tem várias implementações, das quais as mais comuns são MPICH [82, 83] e LAM/MPI [142]. O projeto MPICH teve início em 1993 como uma tentativa de fornecer uma implementação imediata do padrão MPI-1. Atualmente, o MPICH é uma implementação completa do padrão MPI-1, com extensão para suportar as funcionalidades de E/S paralelo definidas no padrão MPI-2.

LAM/MPI é uma implementação do padrão MPI, que suporta todo

o padrão MPI-1 e grande parte do padrão MPI-2. Como consequência das funcionalidades implementadas pelo LAM/MPI, essa foi a distribuição do MPI escolhida como ambiente de desenvolvimento usado nesta tese. A Seção 3.2.2 descreve detalhadamente as principais características dessa versão.

3.2.2

LAM/MPI - Uma Versão do Padrão MPI

A implementação LAM do padrão MPI é baseada em duas partes: uma biblioteca e um ambiente de programação paralela (PPE, do inglês *Parallel Programming Environment*). A biblioteca do LAM contém todas as funções do padrão MPI-1 e grande parte das funções definidas no padrão MPI-2. O ambiente de programação paralela do LAM é baseado em *daemons* (processos que executam em *background* no sistema), que fornecem muitos dos serviços requeridos por programas MPI. LAM utiliza um *daemon* no nível do usuário (*lamd*) sob cada máquina, para criar um ambiente de execução persistente. Um conjunto de *lamds* é iniciado pelo comando *lamboot*.

Esse comando envia uma solicitação para cada máquina, tipicamente via *rsh* ou *ssh*, para que cada uma delas inicie um *lamd* local. Quando um *lamd* é criado, ele é preenchido com uma tabela de rotas contendo o endereço de todos os outros processos *lamds*. Os processos *daemons* são sempre usados na criação de processos MPI, mas podem ou não atuar como intermediários na comunicação.

Após a chamada à função *MPI_Init()*, todos os processos MPI ficam cientes da existência de todos os outros processos MPI, formando-se um comunicador consistente (*MPI_COMM_WORLD*). Todos os processos nesse comunicador precisam executar o *MPI_Finalize()* antes de terminar a execução. As chamadas às funções *MPI_Init()* são consideradas ponto de sincronização entre os processos MPI.

A distribuição LAM permite a utilização de arquivos de esquemas, chamado de *appschemas*, nas funções *MPI_Comm_spawn()*. O *appschema* é um arquivo ASCII que permite ao programador especificar um conjunto arbitrário de CPUs ou nós onde serão disparados processos MPI. Caso o programador da aplicação não especifique nenhum arquivo de esquema para a criação dinâmica de processos, o LAM escalona os processos segundo o algoritmo *round robin*, ou seja, atribui um processo paralelo para cada nó LAM, a partir da CPU 0 (ou com menor identificador) e continua até a CPU com maior identificador. Quando o número de processos da aplicação é maior que o número de CPUs, o procedimento de criação de processos se repete a partir da CPU 0.

Atualmente, com o surgimento dos ambientes *grid*, novas versões do padrão MPI têm surgido para atender aos requisitos específicos desse novo ambiente. A implementação LAM/MPI a partir da versão 7.0 passou a fornecer suporte para execução de aplicações paralelas em *grids* computacionais. Outra versão também habilitada para *grid* é o MPICH-G2, que é uma extensão da implementação MPICH. A versão LAM/MPI apresenta várias vantagens em relação ao MPICH-G2, em particular, o suporte às funções de gerenciamento dinâmico de processos, essenciais para ambientes heterogêneos e dinâmicos como os *grids*.

Embora os ambientes de passagem de mensagem apresentados sejam usados na implementação de metaheurísticas paralelas, muitos pesquisadores têm trabalhado com o objetivo de facilitar a paralelização de metaheurísticas através do reuso de códigos existentes, omitindo detalhes específicos dos algoritmos paralelos através de *frameworks*. A próxima seção descreve alguns dos *frameworks* usados na literatura.

3.2.3

Frameworks para Metaheurísticas

Frameworks de algoritmos constituem uma maneira de reduzir o esforço de desenvolvimento para aplicações seqüenciais e paralelas. Os usuários apenas descrevem os elementos que pertencem ao problema e então instanciam as funções que parametrizam a técnica selecionada. No caso paralelo, os *frameworks* escondem os detalhes da implementação e permitem executar algoritmos paralelos escrevendo apenas códigos seqüenciais. Assim, os *frameworks* contribuem para reduzir a distância entre o usuário e as implementações paralelas.

No campo das metaheurísticas, os *frameworks* são padrões voltados para o reuso integral ou parcial do código. Na implementação de um *framework* é aconselhável que ele seja orientado a objeto (OO) e definido como um conjunto de classes abstratas para solucionar uma família de problemas relacionados. Os *frameworks* são principalmente baseados na separação conceitual entre os métodos de solução e os problemas de otimização que eles tratam.

Essa separação requer uma compreensão profunda do domínio da aplicação. A modelagem do domínio da aplicação resulta em um conjunto de classes reusáveis com alguns aspectos variáveis e outros constantes. A implementação da parte constante é fornecida pelo *framework* e encapsulada dentro de classes abstratas genéricas. A parte variável é específica do problema e fixada no *framework*, mas ela é implementada pelo usuário. Essa parte é usada para implementar os detalhes específicos de cada aplicação. Dessa forma, quando o usuário trabalha com *framework* para implementar aplicações parale-

las, ele não precisa ter conhecimento algum sobre paralelismo porque todas as construções relacionadas são incluídas dentro da biblioteca de maneira transparente ao usuário.

Há na literatura vários trabalhos voltados para o projeto reusável de metaheurísticas paralelas e seqüenciais. Alguns desses *frameworks* são restritos para uma classe de metaheurística, como por exemplo os específicos para algoritmos evolucionários: ECJ [103], Distributed BEAGLE [72], J-DEAL [42] e DREAM [18]; ou os específicos para busca local como EasyLocal++ [50] e Localizer++ [112], sendo esses últimos apenas para metaheurísticas seqüenciais. Todavia, há *frameworks* mais gerais que trabalham simultaneamente tanto com metaheurísticas seqüenciais quanto paralelas, destacando-se MALLBA [6, 7] e ParadisEO [36].

Framework MALLBA

MALLBA é um *framework* escrito em C++/MPI que oferece um conjunto de técnicas de resolução para solucionar problemas de otimização. Cada técnica de resolução é encapsulada dentro de um bloco separado. Atualmente, os seguintes blocos para métodos exatos estão disponíveis: divisão e conquista, *branch-and-bound* e programação dinâmica. Para métodos heurísticos existem *hill climbing*, *metropolis*, *simulated annealing*, busca tabu e algoritmo genético.

MALLBA é um dos poucos *frameworks* que propõem a possibilidade de desenvolver métodos híbridos paralelos, compostos por métodos exatos e heurísticos, como por exemplo algoritmos genéticos e busca tabu; algoritmos genéticos e *simulated annealing*; ou *branch-and-bound* e *simulated annealing*.

Além de oferecer código para problemas reais, MALLBA fornece código genérico em que os usuários podem especializar para seus próprios problemas. O princípio básico do *framework* MALLBA consiste em permitir que os usuários possam instanciar qualquer problema de otimização de seu interesse, definindo apenas as características mais importantes dependentes do problema.

Os elementos relacionados à funcionalidade do algoritmo são omitidos do usuário. A biblioteca MALLBA fornece implementações para três ambientes: seqüencial, LAN e WAN. Todavia, MALLBA não tem suporte para implementações paralelas em ambiente *grid* [7].

Framework ParadisEO

Outro *framework* que dá suporte à implementação paralela de diferentes metaheurísticas e métodos exatos é o ParadisEO. O *framework* ParadisEO é uma extensão do *framework* EO (*Evolving Objects*), desenvolvido inicialmente para algoritmos evolucionários seqüenciais. Essa extensão inclui vários métodos

de busca local, modelos distribuídos, modelos paralelos e diferentes mecanismos de hibridização.

O ParadisEO é código aberto e sua implementação divide as classes de implementação em duas categorias: classes fornecidas e classes requeridas. As classes fornecidas implementam a parte invariante do código. As demais codificam a parte específica do problema, as quais são classes abstratas que têm que ser especificadas e implementadas pelo usuário. Neste *framework* o usuário tem a possibilidade de escolher entre os ambientes MPI ou PVM para a camada de comunicação. Além disso, a granularidade fina dos objetos do ParadisEO possibilita uma maior flexibilidade, adaptabilidade e reuso do código. Contudo, atualmente, a execução sob um *grid* computacional só é possível através do *framework* ParadisEO-CMW [109], que é uma extensão do ParadisEO habilitada para *grid*.

Essa extensão é combinada com a biblioteca Condor-MW [81] que é a responsável por garantir a execução das tarefas sob o ambiente *grid* de maneira transparente para o usuário. A arquitetura do ParadisEO-CMW é baseada em um *middleware grid* e duas interfaces, sendo uma interface de infra-estrutura e uma interface de programação da aplicação *grid*. A primeira interface fornece as ferramentas de comunicação e gerenciamento de recursos. A camada do *middleware* é implementada pelo Condor [143, 144] e as interfaces são disponibilizadas pelo *framework* MW.

Condor é um sistema HTC (*High-Throughput Computing*) que trata recursos computacionais heterogêneos e múltiplos usuários. Ele distribui as tarefas submetidas pelo usuário às máquinas ociosas pertencentes à sua lista de recursos. Para verificar a disponibilidade dos recursos, Condor usa a carga média de CPU e informações sobre o uso recente de alguns periféricos, como teclado e *mouse*. O MW é um *framework* desenvolvido para permitir a implementação rápida de aplicações paralelas, usando o modelo mestre-trabalhador para o ambiente *grid*.

Contudo, apesar de ambos os *frameworks* MALLBA e ParadisEO-MW fornecerem um alto nível de abstração dos detalhes da implementação paralela, o programador é obrigado a implementar sua aplicação paralela seguindo algumas das estratégias de paralelização previamente definidas nesses *frameworks*, impossibilitando maior adaptabilidade das aplicações ao ambiente.

Uma diferença básica entre os *frameworks* MALLBA e ParadisEO-MW, é que o segundo é capaz de executar em ambiente *grid*, mas o primeiro não. Embora ParadisEO-MW seja capaz de executar em *grids*, ele só dispara novas tarefas em máquinas ociosas. Em um ambiente *grid* real isso significa que a aplicação pode levar muito tempo para iniciar sua execução, esperando que

alguma máquina entre no estado de ociosidade.

O *framework* SGA EasyGrid executa aplicações paralelas em ambientes de *grid* sem exigir ociosidade das máquinas, sendo capaz de adaptar a execução da aplicação às mudanças dinâmicas do ambiente. Todos os detalhes específicos desse *middleware* serão vistos na Seção 4.3.

Como visto nesta seção, é importante que o programador escolha a ferramenta de paralelização que mais se adapte às suas necessidades e às características da aplicação. Contudo, outro fator que tem papel fundamental no desempenho final da aplicação paralela é a estratégia de paralelização usada para definir a gerência e o sincronismo das tarefas. As estratégias mais comuns de paralelização são vistas na próxima seção.

3.3

Estratégias de Paralelização

A escolha adequada de uma estratégia de paralelização é uma das características-chaves a serem consideradas na implementação de uma metaheurística paralela eficiente. Estratégias de paralelização diferenciam-se consideravelmente entre si, pois para cada tipo de metaheurística há estratégias mais adequadas. Dessa forma, embora haja na literatura várias classificações de estratégias de paralelismo [43, 110], neste trabalho será descrita a classificação para metaheurísticas de busca local proposta por Cung et al. [46].

Essa classificação é independente da arquitetura e baseou-se em [149]. Nesse trabalho, Verhoeven e Aarts [149] destacam que metaheurísticas baseadas em busca local devem ser vistas como a exploração do grafo de vizinhança (ou espaço de soluções) associado com uma instância, no qual os nós correspondem às soluções e os arcos conectam soluções vizinhas. Cada iteração consiste basicamente na avaliação da solução na vizinhança da solução corrente, seguido por um movimento para um deles, evitando tanto quanto possível parar prematuramente em um ótimo local, até que nenhuma melhoria da melhor solução conhecida possa ser alcançada. A seqüência de soluções avaliadas ao longo da busca define um caminho (ou uma trajetória) no grafo de vizinhança.

Assim, Cung et al. [46] distinguiram dois padrões para a paralelização da busca local, de acordo com o número de trajetórias investigadas no espaço de soluções: caminho único (*single walk*) e caminhos múltiplos (*multiple walk*). No caso de uma paralelização com caminho único, uma única trajetória de busca é investigada no espaço de soluções. A busca pela melhor solução, em cada iteração, é realizada em paralelo, ou pela paralelização da avaliação da função de custo ou pela decomposição do domínio. Nesse caso, a avaliação

da vizinhança ou a instância do problema são decompostas e distribuídas em diferentes processadores.

Nesse caso, a busca na vizinhança é mais rápida, mas somente uma trajetória de busca é investigada, como na implementação sequencial. A paralelização com caminho único geralmente é implementada com granularidade fina ou média e seus processos são extremamente dependentes da aplicação, em termos de definição da estrutura de vizinhança e da função de custo.

Por outro lado, a paralelização por caminhos múltiplos caracteriza-se pela investigação em paralelo de múltiplas trajetórias, cada uma das quais por um processador diferente. Essa estratégia trabalha com uma granularidade grossa. Nesse caso, além da busca por melhores acelerações, é desejado também que soluções melhores possam ser encontradas, pois vários espaços de soluções são investigados ao mesmo tempo.

Os processos envolvidos na paralelização por caminhos múltiplos podem ser independentes ou cooperativos. No primeiro caso, nenhuma troca de informação é realizada entre os processos, ou seja, cada processo executa sua tarefa e ao término envia sua solução para um processo central, responsável por reunir todas as soluções geradas. No segundo caso, a informação coletada durante as trajetórias é disseminada e usada por outros processos para melhorar ou para acelerar a busca. Os processos cooperam dinamicamente durante a busca, enviando e recebendo informações que podem alterar o seu percurso no espaço de soluções, podendo direcionar sua busca para caminhos mais promissores.

No caso de estratégias independentes, é possível distinguir dois padrões básicos: exploração em paralelo de múltiplas trajetórias originadas a partir de diferentes nós do grafo, ou exploração em paralelo de subgrafos do grafo de vizinhança. No primeiro caso, exploração em paralelo de múltiplas trajetórias originadas a partir de diferentes nós do grafo de vizinhança, cada trajetória começa a partir de uma diferente solução. Os processos podem usar o mesmo algoritmo de busca local ou não, assim como parâmetros iguais ou diferentes. Assim, se p processadores são usados, essa estratégia corresponde à execução sucessiva de p buscas independentes sequenciais.

No segundo, o grafo é decomposto em vários subgrafos, e esses são explorados em paralelo, sem intervenção das trajetórias correspondentes. Nessa estratégia, cada processo recebe um subgrafo e faz a busca no seu espaço de soluções. Essa estratégia é adequada quando o espaço de soluções a ser investigado é grande, possibilitando assim que toda a vizinhança seja pesquisada.

Estratégias de paralelização por caminhos múltiplos com busca independente podem ser facilmente implementadas. Muitas implementações robustas podem ser obtidas com diferentes ajustes dos parâmetros em cada processador. Todavia, esse modelo pode facilmente ser simulado através de algoritmos sequenciais multi-partidas, por várias execuções sucessivas com diferentes inicializações. A falta de cooperação entre os processos de busca não permite o uso de informações coletadas por diferentes processadores ao longo de suas trajetórias. Além disso, geralmente as trajetórias de busca são muito longas e problemas de desbalanceamento de carga são comuns. Trabalhos redundantes podem ser feitos se o espaço de busca não for particionado apropriadamente.

A estratégia de paralelização por caminhos múltiplos com busca cooperativa é a mais geral e promissora estratégia de paralelização para metaheurísticas, demandando mais esforço de programação e implementação [46]. Os processos de busca trocam e compartilham informações coletadas ao longo das trajetórias que investigaram.

Essa informação compartilhada é implementada ou como variável global armazenada na memória compartilhada ou como um vetor na memória local de um processador central, que pode ser acessado por todos os outros processadores. Nesse modelo, há três aspectos importantes a serem considerados na implementação, que refletem diretamente no desempenho da aplicação. Os aspectos abordam qual informação compartilhar, como acessar essa informação e quando realizar a troca da informação entre as tarefas paralelas.

A escolha da informação a ser compartilhada ou trocada é um aspecto difícil a ser considerado porque é necessário escolher informações relevantes, sem que muito espaço em memória seja usado. O outro aspecto, relacionado à coleta da informação, deve ser cuidadosamente planejado para evitar que as informações sejam coletadas em espaços curtos de tempo, correndo o risco de capturar informações irrelevantes, ou que seja gasto muito tempo na coleta da informação. Por último, é necessário estabelecer regras que definam quando a troca de informação deve ocorrer, se a troca de informação será por demanda ou periódica, se ela será iniciada por um processo central ou se cada processo é responsável por buscar sua solução.

É importante que todos esses aspectos sejam considerados de acordo com as características da aplicação e da plataforma usada na execução da metaheurística paralela. A próxima seção descreve as principais características dos ambientes computacionais usados para executar aplicações paralelas.

3.4 Ambientes Computacionais Paralelos

Em contraste com os algoritmos seqüenciais, algoritmos paralelos são fortemente dependentes da arquitetura paralela para o qual são projetados. As arquiteturas usadas em computação paralela diferenciam desde máquinas de memória compartilhada (ou multiprocessadas) até máquinas de memória distribuída (ou multicomputadores) [47].

As máquinas com memória compartilhada caracterizam-se por adotarem vários processadores vinculados a uma única memória, nesse caso a troca de informação é realizada através de escrita e leitura em memória. Embora essas máquinas sejam amplamente usadas, elas apresentam uma desvantagem causada pela limitação do número máximo de processadores que podem fazer parte de cada máquina.

Por outro lado, com as melhorias apresentadas pelas redes locais e a produção em massa de computadores pessoais, as máquinas com memória distribuída têm sido constantemente usadas como uma alternativa para a construção de máquinas paralelas. Essas máquinas são compostas de coleções de computadores interconectados, cada uma tendo seu próprio processador, memória e adaptador de rede. Comparada às máquinas de memória compartilhada, as máquinas de memória distribuída apresentam vantagens significativas, principalmente uma melhor relação custo/benefício, maior escalabilidade e maior flexibilidade, representando uma ótima escolha para executar aplicações intrinsecamente distribuídas [8].

Atualmente, entre as máquinas de memória distribuída, as mais populares são os *clusters* e *grids* de computadores. Um *cluster* é essencialmente um grupo de computadores pessoais ou estações de trabalho conectadas através de uma rede. A principal vantagem dos sistemas de *clusters* é a sua relação custo/benefício, quando comparada com outras máquinas paralelas. Todavia, apesar do baixo custo, devido à utilização de componentes pré-existentes no ambiente, os *clusters* são geralmente ineficientes pois os computadores normalmente são dedicados a um número restrito de usuários, o que pode acarretar ociosidade durante grande parte do tempo. Além disso, devido à crescente demanda por poder computacional requerida por aplicações cada vez maiores, esses sistemas não são capazes de atender às necessidades apresentadas por diversas aplicações paralelas.

Um *grid* computacional [64, 66] é formado por uma coleção de recursos distribuídos geograficamente, heterogêneos e compartilhados. Geralmente, esse conjunto de recurso é chamado de *site*, mas nesta tese ele foi traduzido para sítio. Esses sítios são conectados por uma rede de alta velocidade, e oferecem

poder computacional suficiente para executar aplicações que não podem ser executadas em *cluster* de computadores devido à insuficiência de poder computacional. Esse ambiente computacional também é classificado como multi-computador, e é citado como um exemplo dos metacomputadores [48, 64].

Assim, esse novo ambiente emerge com o objetivo de fazer com que a computação de alto desempenho também seja acessível aos usuários que não possuem, necessariamente, recursos suficientes localmente para suprir suas necessidades de poder computacional ou armazenamento. Contudo, a exploração eficiente desse tipo de ambiente ainda é um desafio, principalmente devido ao seu comportamento dinâmico e instável. Ao contrário dos sistemas distribuídos tradicionais, esse novo ambiente precisa considerar questões como segurança, acesso uniforme aos recursos geograficamente distribuídos (independentemente da localização física dos mesmos), descoberta e agregação dinâmica de recursos e qualidade de serviços. O Capítulo 4 apresenta a plataforma *grid* e todos os seus detalhes arquiteturais.

Com a plataforma de execução definida, outra decisão importante a ser tomada no projeto de uma metaheurística paralela diz respeito ao modelo de programação, veja Seção 3.5.

3.5 Modelo de Programação Paralela

No campo das metaheurísticas, o modelo de programação paralela determina como a busca global é controlada. Existem basicamente dois modelos de programação paralela: centralizado e distribuído. No modelo centralizado, também chamado mestre-trabalhador, a busca é controlada por apenas um processo (o mestre). Os dados são manipulados diretamente pelo mestre ou armazenados em uma memória compartilhada. No primeiro caso, os processos trabalhadores comunicam com o mestre para obter o trabalho a ser feito e enviar os resultados obtidos. No segundo caso, os processos trabalhadores obtêm as tarefas a serem feitas a partir da memória compartilhada, onde eles também armazenam os resultados obtidos.

O modelo distribuído é caracterizado pela presença de vários processos responsáveis pelo controle da busca global. Nesse caso, a informação é compartilhada entre os trabalhadores pela troca de mensagem. Esse modelo tem a vantagem de que a decisão a ser tomada pode levar em consideração o estado global do sistema. Todavia, todas as decisões são mais demoradas para serem tomadas, pois todos participam de qualquer decisão.

A escolha do modelo de programação apropriado depende fortemente da arquitetura da máquina paralela na qual a aplicação será executada.

Atualmente, o modelo mestre-trabalhador é o modelo mais comumente usado para implementar metaheurísticas paralelas em plataformas distribuídas, tais como *clusters* e *grids*. Esse padrão é especialmente atrativo para ambientes distribuídos porque ele apresenta maior facilidade na migração de aplicações desenvolvidas para ambientes mono-processado para ambientes distribuídos, beneficiando-se de todos os recursos do ambiente.

Além disso, o processo mestre pode realizar a tarefa de processamento juntamente, ou não, com a tarefa de balanceamento de carga. Entretanto, quando o ambiente distribuído a ser considerado é um ambiente de larga escala, no qual os recursos são heterogêneos e não dedicados, como os *grids* computacionais, esse modelo apresenta fatores que podem afetar o desempenho da aplicação.

Por exemplo, a sobrecarga de comunicação entre o mestre e os processos trabalhadores tende a ter um impacto negativo no desempenho, particularmente nos ambientes computacionais em que os recursos podem ser conectados através de enlaces de rede com alta latência ou baixa vazão. Outro ponto negativo é a existência de um único processo mestre, pois isso pode tornar-se um gargalo que aumenta com o número de processos trabalhadores (ou dos recursos computacionais).

Assim sendo, faz-se necessário desenvolver estratégias de paralelização que sejam capazes de se adaptar às características dos novos ambientes computacionais, garantindo que a aplicação paralela possa se beneficiar eficientemente de toda a capacidade do ambiente. Esta tese propõe uma estratégia de paralelização hierárquica distribuída, capaz de naturalmente diminuir o tempo de execução da aplicação por dividir e distribuir as funcionalidades do processo mestre, como será descrito no Capítulo 6.

3.6

Considerações Finais

Este capítulo abordou os principais tópicos a serem definidos ao implementar uma metaheurística paralela. É importante que todas as decisões tomadas sejam feitas em relação às características da aplicação e da plataforma usada na execução pois, diferentemente dos algoritmos sequenciais, os algoritmos paralelos são fortemente dependentes da máquina paralela.

Algumas ferramentas de programação foram descritas nesse capítulo, com ênfase na plataforma MPI porque foi a ferramenta escolhida para ser usada nas implementações desta tese. Em seguida, foram apresentadas as possíveis estratégias de paralelização usadas em metaheurísticas, e os modelos padrão de paralelização. Por último, algumas arquiteturas utilizadas na execução de

algoritmos paralelos foram apresentadas.

Como dito anteriormente, o usuário precisa conhecer a plataforma usada para que possa tomar as decisões certas. Por isso, o Capítulo 4 é dedicado ao ambiente *grid* computacional, em especial à sua arquitetura em camadas, suas vantagens e desvantagens.