UNIVERSIDADE FEDERAL DE SANTA CATARINA
PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

Evandro Blume

ALGORITMO DE OTIMIZAÇÃO PARALELO: UM MODELO PROPOSTO E IMPLEMENTADO

Dissertação submetida à Universidade Federal de Santa Catarina como parte dos requisitos para a obtenção do grau de Mestre em Ciência da Computação

Prof. José Mazzucco Junior, Dr.
Orientador

Florianópolis, agosto de 2002

Algoritmo de Otimização Paralelo: um Modelo Proposto e implementado

Evandro Blume

Esta dissertação foi julgada adequada para a obtenção do Título de Mestre em Ciência da Computação, Área de Concentração: Sistemas de Computação, e aprovada em sua forma final pelo Curso de Pós-Graduação em Ciência da Computação – CPGCC.

_____

Prof. Fernando A. Osthuni Gauthier, Dr. – Coordenador

Banca Examinadora

_____

Prof. José Mazzucco Junior, Dr. – Orientador

_____

Prof. Luis Fernando Friedrich, Dr. – INE, UFSC

_____

Prof. Thadeu Botteri Corso, Dr. – INE, UFSC

Dedico este trabalho a
Aline e Paula, minhas filhas;
Werno e Ceres, meus pais;
Liliane, minha esposa,
e a seus pais,Clarindo e Ecila.

AGRADECIMENTOS

A DEUS, por tudo que me é proporcionado todos os dias e por ter permitido a conclusão deste trabalho.

Ao Prof. Dr. José Mazzucco Junior pela oportunidade, orientação, compreensão, amizade e paciência no período em que elaboramos este trabalho.

À Liliane, minha esposa, que muito contribuiu com seu carinho, seu apoio, seu incentivo e, principalmente, por ter compreendido os diversos obstáculos que enfrentou durante o período em que eu estava desenvolvendo este trabalho.

Às minhas filhas Aline e Paula, pelo carinho e afetividade que passam em qualquer momento em que estamos juntos ou pensamos nelas. Pequeninas pessoas fundamentais na trajetória de minha vida e deste trabalho.

Aos meus pais Werno e Ceres, pelo carinho, incentivo e encaminhamento que deram para que eu pudesse realizar mais este passo.

Aos meus irmãos, que muitas vezes, mesmo não percebendo, tiveram influência no incentivo para o término deste trabalho.

Aos meus tios e padrinhos Gilberto e Berenice, bem como a seus familiares, que souberam me acolher e dedicar seu carinho, quando estava longe de minha casa.

A meus demais familiares que, sem dúvida nenhuma, esperam minha felicidade e sucesso.

Aos colegas e professores, pelos bons momentos de convivência e aprendizado durante a primeira fase desta jornada.

Aos meus amigos e colegas da UNIJUÍ, que me apoiaram e ajudaram na realização deste trabalho, especialmente àqueles que contribuíram diretamente para o mesmo.

À UNIJUÍ, Universidade Regional do Noroeste do Estado do Rio Grande do Sul, que me proporcionou condições para efetuar tais estudos.

RESUMO

A busca de soluções para problemas de otimização das informações nas organizações por meio do computador constituiu a base deste trabalho. No que tange à Ciência da Computação, essa busca certamente requer a construção de algoritmos eficientes e exatos, mas nem sempre encontram-se boas soluções para muitos problemas de ordem prática, principalmente no que diz respeito ao tempo de execução. Existem problemas, dentre os quais estão os de otimização combinatorial, que diferem dos outros porque apresentam uma grande dificuldade para se obter soluções exatas, num tempo computacional aceitável.

Existem técnicas, especialmente as metaheurísticas, tais como *Tabu Search*, *Simulated Annealing*, Algoritmos Genéticos e Redes Neurais, que vêm conseguindo sucesso na solução de problemas de otimização combinatorial e, mesmo não apresentando soluções exatas, têm mostrado bastante eficiência com suas soluções aproximadas.

Este trabalho propõe um novo método, baseado no algoritmo *Simulated Annealing* (SA), modificado para trabalhar com múltiplas faixas de temperatura, de forma que os processos são executados de forma paralela, trocando informações de seus melhores resultados entre os processos existentes a cada início de uma nova faixa.

Os experimentos são executados com instâncias euclidianas do Problema Caixeiro Viajante, que é um problema de otimização combinatorial de difícil solução, apresentando resultados bastante satisfatórios quando comparado com o SA de múltiplas faixas, executado de forma seqüencial.

ABSTRACT

The trying of solving informational optimization problems in organizations through the computer is the objective of this work. Regarding to computer sciences, this research certainly requires the development of efficient and exact algorithms. However, for several practical problems we do not find good solutions, mainly regarding to the execution time of such algorithms.

There are problems such as combinatorial optimization, that are different from others for presenting great difficulty in obtaining exact solutions in an acceptable computational time.

There are techniques, especially the met-heuristics such as tabu search, simulated annealing, genetic algorithms and neural networks, that are getting success in the solution of combinatorial optimization problems, and even not presenting exact solutions, they showed a lot of efficiency with their approximate solutions.

This work proposes a new method based on the simulated annealing algorithm that has been modified to deal with multiple ranges of temperature in which the processes are executed in a parallel way, changing information on their best results between the existing processes in each new range of temperature.

The experiments were executed with the Euclidean instances of the Traveling Salesman Problem, a combinatorial optimization problem with difficult solution that showed satisfying results if compared with the simulated annealing with multiple ranges being executed in sequence.

SUMÁRIO

LISTA DE TABELAS

LISTA DE FIGURAS

CAPÍTULO 1

Introdução

Este primeiro capítulo tem a finalidade de apresentar, em linhas gerais, o presente trabalho. Assim, inicialmente, relata-se a origem da pesquisa, destacando-se a definição do problema abordado; a seguir apresenta-se a síntese dos objetivos e as limitações da pesquisa deste trabalho. Também, pode-se observar o item referente à justificativa e, finalmente, mostra-se a seqüência dos capítulos que compõem esta pesquisa.

1.1 Origem do trabalho

Diante do cenário cada vez mais agressivo da globalização, em que uma boa administração dos custos e maior atenção à produtividade tornam-se fundamentais para a sobrevivência da organização, surge a necessidade de investimentos na qualificação dos processos que, decisivamente, agregam valor à tomada de decisões e potencializam a correta leitura dos resultados efetivos.

No decorrer das últimas décadas, observa-se uma aplicação crescente de métodos científicos na resolução destes processos. O desenvolvimento tecnológico tem contribuído, fortemente, para a evolução de tais processos, principalmente, na área de otimização.

Partindo-se desta premissa, a base para a geração de condições de avaliação por parte dos gestores em relação às ações de sua responsabilidade é a manutenção de um esquema (fluxo) de informações. Mais que isto, é o sistema global de informações geradas e reproduzidas na organização e/ou para a organização.

A necessidade de se estar informado pressupõe, em primeiro lugar, a geração dos dados (alimentar o sistema com informações que representem o evento funcional) e, em segundo, a extração (processamento) eficiente dos valores representativos do conjunto dos dados. Desta forma, as respostas do sistema, obrigatoriamente, fluem em sintonia com o próprio objetivo do projeto.

A situação atual do mercado mundial já confirma as previsões dos especialistas e predefine a sua tendência natural. Todos os setores da economia (indústria, comércio e serviços), serão regidos, oportunamente, sob a ótica do consumidor (seja ele final ou intermediário). O que importa é a satisfação do consumidor. E é a partir do seu julgamento que a organização se sobressairá enquanto vencedora diante de vários competidores do mercado.

Endende-se que o vencedor é a organização - equipe de pessoas, processos e produtos - melhor preparada para a competição. Esta qualificação pressupõe, necessariamente, uma *equipe* consistente e motivada para os objetivos propostos, uma capacidade de *flexibilidade* funcional capaz de surpreender a concorrência e os seus clientes com *conhecimento e tecnologia*. Em outras palavras, a organização sabe onde quer e como chegar, sabe dos obstáculos, conhece suas potencialidades e fraquezas e conhece, também, seu concorrente. O elemento conhecimento, por sua vez, compreende *informação* e capacidade de gerá-la e usá-la eficaz e eficientemente.

Não há como negar que há uma demanda cada vez maior por informações. Os gerentes dependem cada vez mais das informações nas suas atividades decisórias. Além disso, o volume de informações necessárias está crescendo e, por conseguinte, a qualidade e agilidade são exigidas também.

Literalmente, a informação é o resultado de um processo de manipulação e refinamento de dados anteriormente coletados (entrada). O formato e a abrangência desta informação (saída) está, ou deverá estar, de acordo com o nível administrativo (operacional, tático ou estratégico) e dela surgir, ou deverá derivar a orientação para as ações dentre os cenários comuns aos níveis já citados.

A importância de um Sistema de Informações confirma-se partindo do princípio de que as informações que fluem em torno (dentro e fora) da organização representam a sua realidade administrativa/econômica/financeira. Considerar que as

informações refletem a realidade da empresa é tão elementar quanto é o ferramental para a coleta, manipulação e extração destas informações de forma otimizada, contribuindo para uma correta leitura (entendimento) da situação e, conseqüentemente, para a ratificação das decisões deliberadas.

Nestes termos, os sistemas de informações deverão ser idealizados sobre uma harmonia muito bem afinada de todo o conjunto de processos da organização. E, o que interage nesses processos ou subsistemas são os *sistemas de informações para a tomada de decisões*, num sentido mais global e sintético da visão administrativa.

Dentre as áreas que podemos trabalhar estas informações estão os serviços, que de alguma maneira necessitam de um conjunto de restrições. Por exemplo, qual o melhor itinerário que um caminhão de uma transportadora deverá fazer para que o trajeto percorrido seja o menor possível? Também a observação da capacidade de carga do caminhão, selecionando o melhor conjunto de itens indivisíveis a serem transportados em um veículo de espaço e capacidade limitados. Qual a melhor combinação para que os custos de uma indústria sejam os menores observando-se a mão-de-obra, a matéria-prima, as máquinas, etc; e evidenciando o seqüenciamento de operações da indústria? Qual a melhor combinação para um quadro de horário escolar, salas, professores, etc? Esses são apenas alguns exemplos de problemas que podem ser encontrados no dia-a-dia das organizações e que envolvem processos de combinação e otimização das informações ao mesmo tempo, sendo assim, denominados problemas de otimização combinatorial.

Áreas como a ciência da computação, a estatística, as engenharias, as ciências econômicas e a administração, entre outras, já propuseram vários problemas que são inerentes à otimização combinatorial. Dos muitos problemas propostos para estudos, a partir dos quais podemos medir a complexidade computacional dos problemas combinatoriais, um dos mais conhecidos é o do Caixeiro Viajante (PCV) ou *Traveling Salesman Problem* (TSP). A definição clássica do PCV é que um caixeiro viajante deve visitar n cidades, passando em cada cidade uma única vez e retornando à cidade de origem. Dadas as distâncias entre todos os pares de cidades, a tarefa do vendedor é encontrar uma rota que minimize a distância total percorrida (LAWER 1985). Aparentemente, o PCV é um problema de fácil entendimento e de estrutura simples,

porém, à medida que o número de cidades aumenta, o grau de dificuldade de resolução do problema torna-se cada vez maior. Essas características são inerentes à maioria dos problemas de otimização combinatorial, que de modo simples e concreto, exemplificam a enorme velocidade de crescimento fatorial.

Resolver problemas de otimização combinatorial consiste em encontrar a "melhor" solução ou a solução "ótima" dentre um número finito ou um número infinito de soluções possíveis e que atendam a um objetivo específico. Uma forma de resolver tais problemas seria, simplesmente, enumerar todas as soluções possíveis e guardar aquela de menor custo. Entretanto, para qualquer problema de um tamanho minimamente interessante e útil, este método torna-se impraticável, já que o número de soluções possíveis é muito grande. Portanto, técnicas mais apuradas são necessárias. Tais problemas apresentam uma peculiaridade com relação aos outros, que é a grande dificuldade de obtenção de soluções exatas num tempo computacional aceitável.

Com o desenvolvimento teórico da ciência da computação, muitas conclusões foram tiradas em relação aos problemas de otimização combinatorial, sendo uma delas a classificação, em que cada problema apresentava uma complexidade particular, sendo que através de um algoritmo exato o esforço necessário para resolvê-lo era exponencial em relação ao tamanho do mesmo. Assim, um algoritmo de complexidade polinomial será capaz de resolver todos os problemas desta classe, ou então nenhum destes poderá ser resolvido em tempo polinomial. Com os resultados apresentados pelos estudos, os problemas ditos NP-Completos passaram a ser de grande interesse, principalmente depois que se constatou que a maioria dos problemas de otimização combinatorial pertence a essa classe de problemas.

Um dos mais clássicos problemas pertencente às classes dos NP-Completos é o PCV, sendo um paradigma para teste de novos algoritmos. Portanto, o fato de um problema ser identificado como NP-Completo é aceito com forte evidência contra a existência de algoritmos polinomiais, justificando a utilização de heurísticas, com o objetivo de se obterem soluções aproximadas e de boa qualidade.

Assim, muita importância vem sendo dada ao uso de heurísticas que forneçam soluções de boa qualidade em tempo de processamento compatível com as necessidades das organizações. As técnicas de metaheurísticas, tais como Tabu search, Simmlated

Annealing, Computação Evolucionária (Algoritmos Genéticos, Programação Evolucionária, etc) e Redes Neurais são de grande importância para a solução destes problemas de otimização combinatorial, mostrando-se bastante eficientes em diversos problemas.

Porém, podemos dizer que é muito pequeno o número de organizações que fazem uso de alguma técnica de otimização em seus processos. Isso se dá, muitas vezes, pela falta de conhecimento e esclarecimento sobre estas técnicas.

As técnicas de otimização, seus modelos teóricos e os métodos desenvolvidos por pesquisadores, por muito tempo, permaneceram esquecidos em livros e artigos sobre o assunto, ficando no esquecimento suas implantações, sendo pouco utilizados pelas organizações para maximizar os resultados de suas respectivas atividades. No entanto, esse esquecimento vem lentamente se modificando, pois aos poucos as organizações vêm adotando os modelos de otimização nos seus fluxos de trabalho, gerando assim, por conseguinte, grande redução nos custos em determinadas tarefas e fluxo. O sucesso dos negócios nos dias atuais cada vez mais depende do desempenho da organização no mercado em que atua, tanto no aspecto da agilidade em seus processos quanto no financeiro, sobrevivendo aqueles que conseguirem obter desempenho superior aos seus concorrentes.

As organizações, aos poucos, vão aceitando a utilização de técnicas de otimização que devem ser assumidas como parte de seu plano estratégico, tendo cada vez mais um papel de grande relevância.

À medida que o volume e as necessidades de novas informações aumentam, os sistemas de gerenciamento de informações baseados no princípio da logística requerem cada vez mais técnicas que propiciem uma melhor utilização dos recursos disponíveis, ocasionando uma redução nos custos logísticos. Para possibilitar que se alcancem estes objetivos, o uso de novos métodos serve como ferramentas importantes e muitas vezes indispensáveis.

Nos dias de hoje, encontra-se um grande número de abordagens que são as técnicas de otimização local, agregando-se aos novos métodos de aproximação, os quais, normalmente, utilizam como base os algoritmos *Tabu Search*, Genético e *Simulated Annealing* que, mesmo não apresentando soluções exatas, têm alcançado

resultados bastante satisfatórios diante dos problemas a que são submetidos. Com estruturas muito simples e de fácil implementação, esta abordagem vem se sobrassaindo em relação aos algoritmos exatos, que têm suas estruturas mais complexas usando modelos matemáticos da programação linear, através dos métodos *brach-cound* e *sutting-planes*, e ainda incorporam características dos problemas tratados, porém com resultados considerados satisfatórios, apesar do excessivo tempo computacional consumido, mesmo para instâncias consideradas pequenas.

O sucesso surpreendente alcançado por estes algoritmos e sua utilização nas soluções de problemas gerais de otimização, principalmente nas combinatoriais, tem levado a intensas pesquisas na área, observando que mesmo com as modernas técnicas desenvolvidas, ainda não foi possível encontrar solução eficiente para um grande número de problemas.

1.2 Objetivo do trabalho

O tema da pesquisa enquadra-se na temática mais geral de resolução de problemas de otimização combinatorial, tendo como objetivo final deste trabalho avaliar o comportamento de um método que tem como base a abordagem *Simulated Annealig* (SA), aplicado ao PCV. O PCV é um problema clássico de otimização combinatória da classe do problemas NP-Completo.

Especificamente, pretende-se fazer uma análise comparativa da performance do algoritmo gerado pelo método baseado na implementação básica do *Simulated Anealing,* porém com os valores da temperatura retirados de múltiplas faixas, conforme Araujo(2001), com o mesmo método incorporando em seu algoritmo técnicas de processamento paralelo.

1.3 Justificativa do trabalho

Nas últimas duas décadas foi feito um grande investimento na área de otimização combinatorial, pois muitos dos problemas desta área são ditos intratáveis,

isto é, pertencem à classe de problemas NP-completos. Justificando assim, o aumento de interesse pelo uso de métodos heurísticos, os quais se tornaram uma importante área de pesquisa e aplicações. A existência de uma solução ótima não é o principal foco de atenção dos pesquisadores da área. O esforço computacional necessário para encontrar o ótimo é que passa a exercer um papel central no projeto de algoritmos para essa classe de problemas.

Enfocando o esforço computacional gasto para executar as tarefas de otimização, e aceito que a utilização destas técnicas traz ganhos significativos às organizações, podemos reduzir os custos logísticos para a efetiva utilização destas técnicas, com a utilização do processamento paralelo. Porém, o objetivo aqui não é enfatizar somente um possível ganho temporal, mas além disso, a organização, supervisão e controle do processo que a programação paralela pode trazer.

1.4 Limitações do tema

Em obras de MARTIN & OTTO (1995) e HELSGAUN (2000) que apresentam algoritmos relativamente eficientes para resolução do PCV, também não existe descrição mais detalhada que possa servir de orientação para o desenvolvimento de novos algoritmos. Isso impossibilita um trabalho mais abrangente na tentativa de comparação direta dos métodos que se mostram mais eficientes.

Não é pretensão deste trabalho apresentar algum recorde reduzindo o tempo de processamento, ou alcançar uma solução ótima para um determinado número de cidades, mesmo porque, para instâncias com um número muito grande de cidades, o tempo de processamento pode levar dias ou até meses, mesmo com a proposta de paralelização dos algoritmos.

A avaliação será sobre o desempenho do método aplicado ao clássico Problema do Caixeiro Viajante, pesquisando de forma comparativa o desempenho do método que tem como base a abordagem *Simulated Annealing*, sendo que um algoritmo será o SA básico introduzindo múltiplas faixas de temperatura, implementado por Araujo (2001), comparado com o modelo proposto, utilizando o mesmo algoritmo e executando múltiplos processos de forma paralela.

1.5 Estrutura do trabalho

O presente trabalho é apresentado em oito capítulos, mais a bibliografia empregada.

No capítulo 2 é apresentada a fundamentação teórica sobre a complexidade dos problemas, incluindo o do caixeiro viajante. Também fazem parte desse capítulo alguns conceitos sobre os principais procedimentos básicos utilizados na solução de problemas dessa classe.

No capítulo 3 é introduzida a fundamentação conceitual da abordagem *Simulated Annealing*, onde os principais elementos básicos desse método são revistos.

No capítulo 4 são apresentados alguns tópicos relevantes da computação paralela, descrevendo seus conceitos básicos, arquiteturas, programação concorrente e sistemas distribuídos.

No capítulo 5 é descrito o paradigma de programação por troca de mensagens, sendo apresentadas as principais abordagens existentes.

No capítulo 6 são feitas algumas considerações sobre abordagens atuais para solucionar instâncias do Problema do Caixeiro Viajante, que reforçam as argumentações do modelo proposto, assim como as alterações introduzidas no algoritmo e sua implementação.

No capítulo 7 é apresentada a metodologia utilizada para a obtenção dos resultados, bem como as argumentações sobre decisões e ajustes de parâmetros. Também faz parte desse capítulo uma análise comparativa entre os resultados computacionais.

Finalmente, no capítulo 8, são apresentadas as considerações finais e algumas sugestões que podem ser exploradas em futuras pesquisas.

CAPÍTULO 2

A Complexidade dos Problemas

Neste capítulo é apresentado, formalmente, o modelo do problema a ser estudado. Inicialmente, uma fundamentação teórica sobre os problemas e suas complexidades é abordada. Em seguida, conceitua-se o problema do caixeiro viajante inserido nesse contexto, apresentando a formulação matemática, sua complexidade e aplicações práticas.

2.1 Complexidade dos Problemas

2.1.1 Introdução

A complexidade computacional de um algoritmo diz respeito aos recursos computacionais, espaço de memória e tempo de máquina requeridos para solucionar um problema.

Geralmente, existe mais de um algoritmo para resolver um problema. A análise de complexidade computacional é, portanto, fundamental no processo de definição de algoritmos mais eficientes para a sua solução. Apesar de parecer contraditório, com o aumento da velocidade dos computadores, torna-se cada vez mais importante desenvolver algoritmos mais eficientes, devido ao aumento constante do “tamanho" dos problemas a serem resolvidos.

O desenvolvimento da Teoria da Complexidade Computacional tem levado a novos estudos sobre a dificuldade inerente de resolver um número cada vez maior de problemas. E também tem contribuído para sua solução, fornecendo métodos rigorosos de avaliação de algoritmos e classificação de problemas. Um dos campos de conhecimento mais férteis, no qual surgem a cada dia novos problemas, é o campo da Otimização Combinatória, em que diversos problemas da maior relevância prática podem ser modelados (CAMPELLO & MACULAN, 1994).

A busca de solução para problemas de elevado nível de complexidade computacional tem sido um desafio constante para pesquisadores das mais diversas áreas. Particularmente, as áreas da otimização, da pesquisa operacional, da ciência da computação, da matemática e das engenharias, é onde se tem defrontado, freqüentemente, com problemas altamente combinatórios, cuja solução ótima, em muitos casos, ainda está limitada a pequenas instâncias.

Nas últimas duas décadas houve um grande investimento na área de Otimização Combinatória, pois muitos dos problemas desta área são tidos como intratáveis, isto é, pertencem à classe de problemas NP-completos. Justifica-se assim, o aumento de interesse pelo uso de métodos heurísticos, os quais se tornaram uma importante área de pesquisa e aplicações, pois a existência de uma solução ótima não é o principal foco de atenção dos pesquisadores da área. O esforço computacional necessário para encontrar o ótimo, passa assim a exercer um papel central no projeto de algoritmos para essa classe de problemas.

Segundo LEWIS & PAPADIMITRIOU (2000), em termos computacionais, os problemas podem ser classificados em duas categorias: aqueles que podem ser resolvidos por algoritmos, e aqueles que não podem, ou ainda, os problemas tratáveis (computáveis) e os não tratáveis (não computáveis). Mesmo com o potencial computacional existente, muitos problemas, apesar de solúveis em princípio, não podem ser resolvidos em qualquer sentido prático por computadores, mesmo pelos mais velozes ou por combinação desses (em redes, em paralelo, etc.), devido às excessivas exigências de tempo e/ou de espaço de memória.

A resolução desses problemas pela teoria atual da ciência da computação é impedida pela dificuldade de produzir um algoritmo, sendo que as técnicas tradicionais

disponíveis são insuficientes para resolver esse tipo de problema. Isto porque um problema é computável se existir um procedimento que o resolva em um número finito de passos, ou seja, se existe um algoritmo que leve à sua solução (ROUTO, 1991). Um exemplo destes problemas não computáveis está na própria matemática, conforme cita LEWIS & PAPADIMITRIOU (2000), “uma vez que já foi provado que não se pode criar um algoritmo que gere as deduções para todas as verdades da matemática”.

As grandes dúvidas que pairam sobre os problemas de elevado nível de complexidade vêm fazendo com que a ciência da computação aponte várias alternativas aos pesquisadores, para que esses disponham de fundamentação teórica suficiente a fim de decidir se em um determinado problema é possível desenvolver um algoritmo que forneça uma solução exata, em tempo aceitável, ou se devem contentar-se apenas com uma solução aproximada.

Uma análise da complexidade do problema é fundamental no processo de definição de algoritmos mais eficientes para a solução. Mesmo com o aumento do poder computacional, torna-se cada vez mais importante desenvolver algoritmos mais eficientes, tanto no âmbito computacional, para transformar um algoritmo seqüencial em paralelo de forma eficiente, quanto em relação ao aumento constante do "tamanho" dos problemas a serem resolvidos.

A complexidade computacional de um algoritmo diz respeito aos recursos computacionais, ao espaço de memória e ao tempo de máquina necessários para se chegar a uma solução do problema. A complexidade não pode ser medida em termos reais, pois o tempo de processamento de um algoritmo varia de um computador para outro. Em geral, existem muitos fatores influenciando o resultado, tais como: eficiência do computador, compilador, recursos da linguagem, sistema operacional, entre outros.

Utilizando métodos empíricos é possível determinar o tempo de um algoritmo, isto é, obter o tempo que ele leva para fazer determinada tarefa, por meio da execução propriamente dita do algoritmo, considerando-se entradas diversas, em que as operações são todas executadas seqüencialmente e a execução de toda e qualquer operação torna-se uma unidade de tempo.

Entretanto, é possível obter-se uma ordem de grandeza do tempo de execução por meio de métodos analíticos. O objetivo desses métodos é determinar uma expressão

matemática que traduza o comportamento do tempo do algoritmo, que, ao contrário do objetivo do método empírico, visa aferir o tempo de execução de forma independente do ambiente utilizado.

2.1.2 Complexidade do algoritmo

A complexidade de um algoritmo pode ser expressa por uma função que relaciona o tamanho de uma instância com o tempo necessário para resolvê-la. Essa função normalmente expressa a quantidade das operações elementares realizadas. A medida da complexidade é, então, o crescimento assintótico dessa contagem de operações. Por exemplo, num algoritmo, para achar o elemento máximo entre *n* objetos, a operação elementar seria a comparação das grandezas dos objetos, no entanto, para valores grandes de *n*, a operação elementar seria a comparação das grandezas efetuadas pelo algoritmo. A noção de complexidade de tempo é descrita a seguir (ROUTO, 1991).

Seja *A* um algoritmo, $\{E_1, ..., E_m\}$ o conjunto de todas as entradas possíveis de *A* denote por t_i, o número de passos efetuados por *A*, quando a entrada for E_i definem-se

complexidade do pior caso = max E_i ? E $\{ t_i \}$,
complexidade do melhor caso = min E_i ? E $\{ t_i \}$,
complexidade do caso médio = S $_{1 = i = m}$ $p_i . t_i$,

onde p_i é a probabilidade de ocorrência da entrada E_i.

A complexidade de pior caso, normalmente representada por O( ), consiste basicamente em assumir o pior dos casos que pode ocorrer, sendo muito usada e, normalmente, a mais fácil de determinar. Ela fornece um limite superior para o número de passos que o algoritmo pode efetuar, em qualquer caso. Por isso mesmo, é a mais empregada, o que faz com que o termo complexidade se refira à complexidade do pior caso. Por exemplo, para encontrar um objeto dentro de uma de dez caixas existentes, sendo que apenas uma delas contém o objeto dentro e as outras estão vazias, a

complexidade no pior caso será O(10). Pois no pior caso o objeto será encontrado na décima caixa. No entanto, a complexidade de mehor caso consiste em assumir que vai acontecer o melhor, tendo aplicações em poucos casos, por essa razão pouco utilizada. Por exemplo, tendo uma lista de números na qual se quer achar algum deles dentro da lista, assume-se que a complexidade é ? (1), pois o número estaria logo no início da lista. Já a complexidade de caso médio é o método mais difícil de determinar, pois necessita-se de análise estatística, e como tal, de muitos testes. No entanto, esse método é muito usado, pois é também o que representa mais corretamente a complexidade do algoritmo.

2.1.3 Complexidade de tempo dos algoritmos

Podemos expressar de forma abstrata a eficiência de um algoritmo, descrevendo o seu tempo de execução como uma função do tamanho do problema. Esse método, chamado de complexidade de tempo, compara dois algoritmos e descreve o seu comportamento temporal em razão do tamanho do conjunto de dados de entrada, Talg = f(n), onde *n* é o tamanho dos dados.

O que interessa é o comportamento assintótico de f(n), ou seja, como f(n) varia com a variação de *n*. É interessante saber como o algoritmo se comporta com uma quantidade de dados realística para o problema e o que acontece quando há uma variação nesses dados. Por exemplo, se existem dois algoritmos Aa e Ab para a solução de um problema, sendo a complexidade de Aa expressa por fa(n) = n^2 e a de Ab por fb(n) = 100*n, isso significa que o algoritmo Aa cresce quadraticamente (uma parábola), e que o algoritmo Ab cresce linearmente (uma reta).

Usando os algoritmos para um conjunto de 30 dados, tem-se o segundo algoritmo com Tb=3000, sendo pior que o primeiro com Ta=900. No entanto, se o conjunto de dados subir para 30.000, então tem-se Ta=900.000.000 e Tb=3.000.000. Isto ocorre porque o comportamento assintótico dos dois é bem diferente, conforme mostrado na figura 2.1.

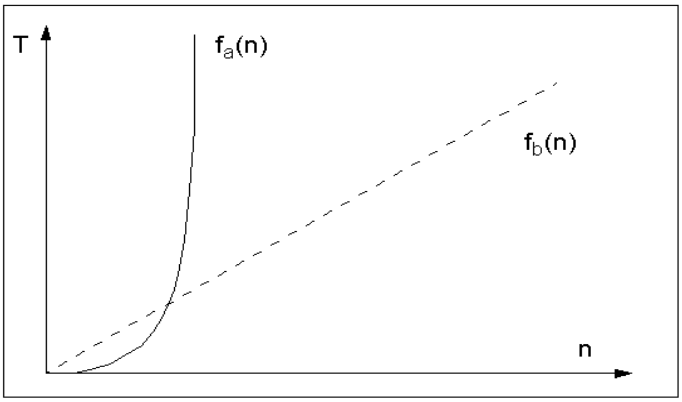

Figura 2.1 - Comportamento Assintótico.

O gráfico da figura 2.1. mostra qual é o aspecto essencial que deve ser expresso pelo cálculo de complexidade. Ou seja, qual é o comportamento assintótico predominante de um algoritmo em função do tamanho do conjunto de dados a ser processado. Se é linear, polinomial, logaritmo ou exponencial.

É comum expressar a complexidade de um algoritmo através da ordem de grandeza da função que o modela. Destaque-se que para para o cálculo do comportamento de algoritmos foram desenvolvidas diferentes medidas de complexidade.

A mais importante delas, e que é usada na prática, é chamada de Ordem de Complexidade ou Notação-O ou Big-Oh. Esse último define-se por: T(n) = O(f(n)) se existem constantes *c* e n0 tais que T(n)c.f(n) quando n0. A definição indica que existe uma constante *c* que faz com que c.f(n) seja sempre pelo menos tão grande quanto T(n), desde que **n** seja maior que um n0. Em outras palavras: A Notação-O fornece a ordem de complexidade ou a taxa de crescimento de uma função, em que se tem a letra O(Ômicron), seguida por uma função entre parênteses, que representa a ordem de grandeza.

A tabela 2.1 representa algumas ordens de grandezas de funções conhecidas:

Ordem de Grandeza	Função
O(1)	Constante
O(log n)	Logarítmica
O(n)	Linear
O(nlogn)	nlogn
$O(n^2)$	Quadrática
$O(n^3)$	Cúbica
$O(n^c)$, c real	Polinomial
$O(c^n)$, c real e > 1	Exponencial
O(n!)	Fatorial

Tabela 2.1 - Ordens de grandeza

A ordem de grandeza de um algoritmo ou sua complexidade O(f(n)) é definida quando o número de operações elementares executadas para obter a solução à instância do problema de tamanho *n* não exceder uma constante f(n), para *n* suficientemente grande, ou seja, diz-se que g(n) é O(f(n)) se existirem duas constantes K e n' tal que |g(n)| = K |f(n))|, para todo n = n'.

Na análise de complexidade de algoritmos, a ordem de grandeza das funções tem uma grande importância. Isto porque é mais fácil e, normalmente, válido, trabalhar com a ordem de grandeza ao invés de computar a função exata do número de operações necessárias, mesmo que para uma mesma função f(n) possa existir uma infinidade de funções limitantes superiores. Apesar disso, o que se procura é a menor função limitante superior para caracterizar a sua ordem de grandeza da complexidade.

Definição: Um algoritmo tem complexidade de tempo polinomial quando: Sejam *f,g:Z+->R.* Dizemos que *g* **domina** *f* (ou *f* é **dominada** por *g*) se existe *m* pertencente a *R+* e *k* a *Z+*, tais que *|f(N)| <= m|g(N)|* para todo *N* pertencente a *Z+*, onde *N >=k,* onde f(N) é dita dominada por g(N), se o limite de |f(N)|/|g(N)| com N tendendo a infinito é igual a um valor real m. Ou ainda, se sua função de complexidade g(n) é um polinômio ou se é limitada por outra função polinomial.

Algoritmos com complexidade polinomial são computacionalmente tratáveis, requerendo um tempo de execução limitado por uma função polinomial, sendo assim solúveis na prática (LEWIS & PAPADIMITRIOU, 2000). No entanto, existe um grande número de problemas para os quais seus algoritmos não são limitados por um polinômio, suas taxas de crescimento são explosivas à medida que *n* cresce, o que torna esses algoritmos inviáveis e que resulta nem sempre contenham fatores exponenciais, como é o caso da função fatorial.

Definição: Diz-se que um algoritmo tem complexidade de tempo exponencial quando sua função de complexidade g(n) não é limitada por uma função polinomial.

Na tabela 2.2 podem-se observar as diferenças na rapidez de crescimento de várias funções típicas de complexidade. Os valores expressam os tempos de execução quando uma operação elementar no algoritmo é executável em um décimo de microssegundo (ROUTO, 1991):

	Valores para n		
Função	20	40	60
N	0,0002 s	0,0004 s	0,0006 s
n log n	0,0009 s	0,0021 s	0,0035 s
n^2	0,004 s	0,016 s	0,036 s
n^3	0,08 s	0,64 s	2,26 s
2^n	10 s	127 dias	3.660 séculos
3^n	580 min	38.550 séculos	$1,3 \times 10^{14}$ séculos

Tabela 2.2 - Comparação de várias funções de complexidade

Observando a tabela 2.2, quando *n* cresce, o crescimento do tempo gasto é extremamente rápido nas funções de complexidade exponencial, inviabilizando qualquer algoritmo de encontrar uma solução exata para o problema. Os algoritmos de complexidade exponencial são considerados inaceitáveis e ineficientes no aspecto

computacional, ao contrário dos algoritmos de complexidade polinomial, que são considerados eficientes.

2.1.4 Classificação dos problemas

A ciência da computação enfrenta grandes problemas quando pesquisa a eficiência dos algoritmos e as dificuldades existentes no seu desenvolvimento e implementação desses algoritmos para solucionar problemas de complexidade computacional. Uma dessas dificuldades está na descoberta de que o esforço necessário para resolver, precisamente, um problema no computador, pode variar enormemente. No que tange a este problema, existe um grande esforço dos pesquisadores para provar a existência ou não de algoritmos que solucionem problemas de elevado nível de complexidade computacional.

Segundo LEWIS & PAPADIMITRIOU (2000), o que tem poupado os cientistas de tentativas inúteis, é a proposição de métodos matemáticos que irão ajudar a provar que um problema de interesse pertence ou não a uma determinada classe. Saliente-se que essa é uma grande contribuição da ciência da computação para a resolução desses problemas.

A integração entre problemas e linguagem está somente em dizer que são coisas iguais, porém, tratadas sob aspectos diferentes. Linguagens são mais apropriadas em conexão à máquina de Turing, enquanto problemas são instruções mais claras de práticas computacionais, como tarefas de interesse prático, para as quais devem ser desenvolvidos algoritmos.

Em tese, a grande maioria dos problemas pode ser resolvida por um algoritmo em tempo polinomial, desde que não se faça exigência quanto ao tipo de máquina que irá executá-lo. A idéia de que o algoritmo pode ser executado em tempo polinomial vem da relação existente entre os problemas de decisão e os algoritmos, através da teoria das linguagens, pela qual sempre é possível exibir um algoritmo que o resolva.

A teoria da linguagem descreve que existem modelos matemáticos com características, por exemplo, não-determinísticas, que não podem ser executados por máquinas reais. Logo, a grande maioria dos problemas, inclusive aqueles que

apresentam complexidade com crescimento exponencial, podem ser resolvidos por algoritmos com características não-determinísticas. O fato é que existem máquinas reais e máquinas que não são reais, que existem apenas teoricamente. As máquinas reais, como os computadores, só podem executar algoritmos em que os passos são determinados; mais precisamente, os computadores só podem executar algoritmos determinísticos, ao contrário dos modelos como a máquina de Turing, que podem também executar algoritmos não-determinísticos.

Existe um grande número de problemas que os algoritmos são capazes de resolver, porém, para muitos deles, não se pode chegar a uma solução em um tempo aceitável. Também há problemas com ordem de grandeza exponencial que, em termos computacionais, são intratáveis. Um problema é referido como intratável quando é difícil encontrar a sua solução, a ponto de ser necessário um tempo exponencial para descobri-la, ou quando a resposta é tão longa que se precisa de um tempo exponencial para descrevê-la. Isso prova que qualquer algoritmo que o resolva não possui complexidade polinomial. Por outro lado, se existir um algoritmo de complexidade polinomial que resolva o problema em tempo oportuno, e esta solução for provada, então considera-se que se tem um problema tratável.

Em relação à complexidade dos seus algoritmos, os problemas são divididos em várias classes, tais como **P**, **NP** e **NP-Completo**, descritas nas sessões seguintes.

Definição: A classe P é formada pelo conjunto de todos os problemas que podem ser resolvidos por um algoritmo determinístico em tempo polinomial, ou ainda pelos procedimentos para os quais existe um polinômio *p(n)* que limita o número de passos do processamento se este for iniciado com uma entrada de tamanho *n*.

Segundo LEWIS & PAPADIMITRIOU (2000), essa parece ser a única proposta séria da teoria da computação, apesar de ainda existirem algumas argumentações contestando a definição citada anteriormente.

A classe de problemas polinomiais tem propriedades de fechamento em relação à adição, à multiplicação e à composição. Assim, um algoritmo polinomial que gera

entrada para outro algoritmo polinomial, forma um algoritmo composto, também polinomial. Estes problemas são da classe P.

Observe-se que mesmo não se conhecendo o algoritmo polinomial de um problema, não significa, necessariamente, que o mesmo não pertença à classe P, devendo ser exibida formalmente a prova de que todo o algoritmo possível para resolver tal problema não é polinomial. O problema da SAT (*Satisfiability Problem*) consiste em determinar se a expressão é verdadeira para uma determinada atribuição das variáveis lógicas. Dado x_1 = V, x_2 = F, x_3 = F, o fato de a resposta exigir o teste de cada uma das cláusulas que conduz à resposta F, e o problema do ciclo hamiltoniano, que consiste em um caminho fechado v_1,v_2,....,vn, v_1, são dois problemas para os quais não se conhece algoritmo exato e tempo polinomial. Esses problemas são, em sua maioria, os chamados problemas de decisão, para os quais todos os algoritmos conhecidos são de complexidade exponencial (LEWIS & PAPADIMITRIOU, 2000). Existem problemas que, apesar de pertencerem a essa classe, não são problemas de decisão, como os de otimização, mas que é possível, através de mecanismos adequados, transformá-los em tal.

Antes de definir a classe NP, descreve-se a noção de não-determinismo. Diz-se que um algoritmo é não determinístico se, além de todas as regras de um algoritmo determinístico, ele pode fazer escolhas de forma não determinística. Informalmente, pode-se definir a classe NP em termos de um algoritmo não determinístico. Tal algoritmo é dividido em duas partes. A primeira parte corresponde à escolha da provável solução (podendo fazer uso de escolhas não determinísticas). A outra parte consiste em verificar, de forma determinística, a viabilidade de uma solução.

> **Definição:** A classe NP é formada pelo conjunto de todos os problemas computáveis, cujas soluções até então conhecidas são de ordem exponencial e não se sabe se existe uma solução melhor, de complexidade polinomial.

Obviamente, todo problema que admite um algoritmo de rapidez polinomial pertence a NP, isto é, P ? NP, pois todo algoritmo determinístico é equivalente a um

não-determinístico. Outro entendimento para a classe NP é o de que ela consiste em todos os problemas de decisão, para os quais existe uma justificativa à resposta SIM, cujo passo de verificação pode ser realizado por um algoritmo polinomial.

Um problema NP-completo é um problema "representante" de uma classe de problemas NP.

Não se sabe se P = NP, embora a maioria dos pesquisadores acredite que esta igualdade não seja válida. Essa é talvez a maior conjectura na área de ciência da computação. Uma das maiores razões de se acreditar que P ? NP, é a existência da classe de problemas NP-completo. Um problema é NP-completo se é NP e se existe uma redução polinomial de qualquer problema NP para este problema, de tal forma que os problemas da classe são redutíveis a ele em tempo polinomial. Por exemplo, o problema de caixeiro-viajante é um poblema redutível ao problema de ciclo hamiltoniano. Dizemos então que o problema de ciclo hamiltoniano é um problema NP-completo. Dessa maneira, se existir um algoritmo polinomial para um problema NP-completo, então todos os problemas de NP podem ser resolvidos polinomialmente. Parece intrigante a existência de tal conjunto, mas, de fato, COOK (1971) mostrou que o problema da satisfatibilidade (SAT) também pertence à classe NP-completo.

Há um grande número de problemas do mundo real, no qual ainda não se conhecem algoritmos em tempo computacional adequado para resolvê-los, e cuja dificuldade intrínseca ainda não foi provada. No entanto, problemas como Caixeiro Viajante, Problema da Mochila, Ciclo Hamiltoniano, SAT, dentre outros, para os quais já se tem provas formais dessas dificuldades, são problemas considerados intratáveis pertencentes à classe NP-Completo. LEWIS & PAPADIMITRIOU, (2000), por exemplo, relacionam inúmeros problemas conhecidamente intratáveis.

Até agora foi demonstrado que, se um algoritmo de complexidade polinomial puder ser encontrado para qualquer um dos problemas NP-completos, então todos os problemas NP-completos serão na verdade problemas P. Por outro lado, se for provado que um dos problemas requer um algoritmo de solução que apresente complexidade exponencial, então todos irão requerer complexidade exponencial (LEWIS & PAPADIMITRIOU, 2000).

A figura 2.2 mostra um simples diagrama da classe NP. Essa classe contém a subclasse P e NP-Completo, sendo P uma subclasse de menor dificuldade, e NPC de maior dificuldade.

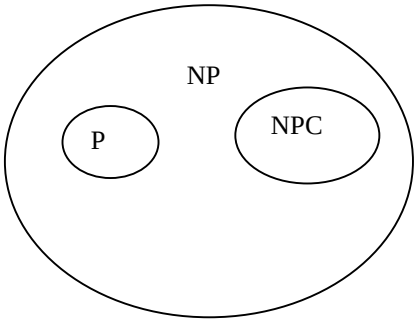

Figura 2.2 - Relacionamento entre P, NP e NPC

2.2 Alguns conceitos básicos

Antes de formular uma definição e os métodos básicos de solução para o problema do caixeiro viajante, é conveniente apresentar algumas conceituações básicas relacionadas ao escopo da formulação e da resolução do mesmo.

2.2.1 Espaço de busca e seus métodos

Tecnicamente, podemos definir o conjunto de todas as possíveis soluções para um problema como sendo o *espaço de busca.* Esse tipo de problema, como é o caso do Caixeiro Viajante, apresentam grandes dificuldades de encontrar uma solucão ótima, por possuírem um gigantesco número de soluções possíveis e, por conseguinte, um vasto espaço de busca.

Um *problema de busca* é caracterizado por um espaço de busca S finito, o conjunto de objetos sobre os quais a busca é conduzida é uma função objetivo, ou seja, f : S ✍ R, estando o mapeamento de *S* para o espaço de valores da função objetivo

R, embora as situações do mundo real apareçam quando R = ? (conjunto dos números reais), e o objetivo for a minimização ou maximização de f. No entanto, R poderia ser um vetor de valores da função objetivo, quando temos problemas de otimização com múltiplos objetivos. Normalmente, sua notação é feita por S ? R^s, observando que o conjunto de todos os mapeamentos de S em R são denotados por R^s, assim, f ? R^s (RADVLIFFE; SURRY, 1995).

A solução para um determinado problema pode encontrar um ponto ou um conjunto de pontos, onde a função f : S ✍ R tem valores máximos ou mínimos que, em termos de um problema de busca, podem ser escritos:

encontrar x^* | $f(x^*) = f(x)$, para todo x ? S, no caso de mínimo.

TANOMARU (1995) descreve três métodos que podem ser pesquisados nesses espaços: numéricos, enumerativos e métodos probabilísticos, todos com seus respectivos derivados e ainda um número grande de métodos híbridos. Os métodos numéricos são vastamente utilizados e apresentam soluções exatas para vários problemas, porém, são ineficientes para otimização de funções multimodais. Além disso, em problemas de otimização combinatorial em espaços discretos, geralmente a função a ser otimizada é não somente multimodal, mas também não diferenciável e/ou não contínua. Métodos enumerativos examinam cada ponto do espaço de busca à procura de pontos ótimos. A idéia pode ser intuitiva, mas é impraticável quando há um número infinito ou muito grande de pontos a examinar. Os métodos probabilísticos, com suas abordagens mais modernas, utiliza a idéia de busca probabilística, entretanto, não são métodos aleatórios baseados em sorte. A computação evolucionária descreve o método *Simulated Annealing*, dentre outros, como uma abordagem conhecida e usual para este método.

2.2.2 Otimização combinatorial

Um problema de otimização combinatorial pode ser um problema de minimização ou de maximização da função objetivo, que para isso deve encontrar a melhor combinação dos objetos, propiciando o melhor desempenho. Ou seja, o problema de escolher a melhor dentre um conjunto de alternativas possíveis.

Sendo *S* o conjunto de soluções possíveis para problemas de otimização combinatória e este sendo finito, podemos então especificar S de duas maneiras: a primeira, uma especificação direta, por meio da simples enumeração de seus elementos, e a segunda uma especificação indireta, através de relações que exploram a estrutura do problema.

A primeira maneira de enumerar todos os elementos de S pode se tornar impraticável no caso de o conjunto ser muito grande. No entanto, são de considerável interesse problemas de otimização em que o conjunto de soluções S seja finito e possa ser caracterizado por meio de relações combinatórias baseadas em propriedades estruturais de *S*, permitindo uma descrição compacta e estratégias de resolução eficientes.

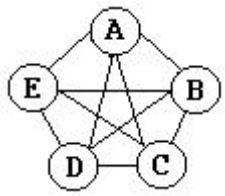

Figura 2.3 – Rede de ligações entre A e B

Considerando o diagrama da figura 2.3, que representa uma rede com os possíveis caminhos ligando o ponto A ao ponto B, nela, cada arco está associado ao custo de percorrer o caminho entre o ponto inicial A e o ponto final B. O problema de Otimização Combinatorial consiste em encontrar o caminho de custo mínimo, partindo de A e chegando a B, passando por todos os pontos.

Como já vimos, na seção 2.2.1, o espaço S denota o conjunto finito de todas as soluções possíveis e *f* a função custo, que é um mapeamento definido por f : S ✍ R. No caso de minimização, o problema é encontrar uma solução x_{opt} ? S que satisfaça $f(x_{opt})$ = f(x), para todo x ? S. No caso de maximização, o problema é encontrar uma solução x_{opt} ? S que satisfaz $f(x_{opt})$? f(x), para todo x ? S. A solução x_{opt} é chamada de solução ótima global, mínima ou máxima, e $f_{opt} = f(x_{opt})$ denota o custo ótimo. Pode haver mais de uma solução com custo ótimo, definindo dessa forma S_{opt} como o conjunto de soluções ótimas.

2.2.3 Pesquisa local e vizinhança

A pesquisa local foi uma das primeiras técnicas propostas, durante os anos 60, para enfrentar com relativo sucesso a intratabilidade computacional dos problemas de otimização combinatorial de difícil solução. Os autores GU & HUANG (1994) consideram a *pesquisa local* como uma das técnicas mais eficientes para solucionar problemas de otimização combinatorial e também como base para muitos métodos heurísticos. Isoladamente, é uma técnica iterativa simples para encontrar boas soluções. Porém, devido à irregularidade no espaço de pesquisa, a mesma, freqüentemente, fica presa a uma solução ótima, porém, local, como mostra a figura 2.4.

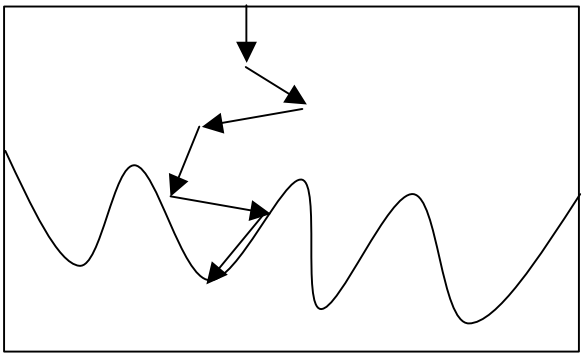

Figura 2.4 – Ótimo Local

A pesquisa local consiste em se mover de uma solução para a outra através da vizinhança dessa solução, de acordo com algumas regras. Dada uma solução corrente x_i ? S e a cada passo i, uma nova solução x_{i+1} é escolhida na vizinhança $N(x_i)$ da solução corrente x_i, de tal modo que, no caso de minimização, $f(x_{i+1}) = f(x_i)$ o processo se repete e x_{i+1} passa a ser a solução corrente, se essa solução existir; caso contrário, x_i é a solução ótima local.

Segundo GU & HUANG (1994), para aplicar a técnica de pesquisa local a um problema, o procedimento é bastante simples, basta especificar a estrutura da vizinhança, definir a função objetiva e o procedimento para obter uma possível solução inicial. Especificar a estrutura de *vizinhança* é fundamental para qualquer método que utilize pesquisa local. Definir uma vizinhança adequada para um dado problema é certamente o fator mais crítico para o sucesso de qualquer tipo de pesquisa local. A vizinhança de uma solução *x (N(x))* contém todas as soluções que podem ser alcançadas de x por um único movimento.

Recentes estudos sobre métodos de pesquisa local para o PCV têm feito maior uso do poder da estrutura de vizinhança. GUTIN & YEO (1998), usando a vizinhança de PUNNEN (1996), construíram vizinhanças de tamanho exponencial polinomialmente alcançável, utilizando grafos para representá-las. Segundo esses autores, quando todas as vizinhanças são polinomialmente alcançáveis, o gráfico correspondente também o é. O diâmetro do gráfico da vizinhança é uma característica importante da estrutura dela no esquema de pesquisa local correspondente. Claramente, vizinhança com gráfico de diâmetro pequeno parece mais potente que aquelas com gráfico de diâmetro grande, ficando, logicamente, de fora aquelas cujos gráficos têm diâmetro infinito. Segundo GUTIN & YEO (1998), a principal propriedade de uma vizinhança *N(t)* imposta usualmente é quando a melhor dentre todas as rotas da vizinhança pode ser alcançada em tempo polinomial.

PAPADIMITRIOU & STEIGLITZ (1982) *apud* LEWIS & PAPADIMITRIOU (2000), também fazem algumas considerações sobre estruturas de vizinhança. Segundo eles, se o número de vizinhos para uma dada solução é muito grande, a qualidade de uma solução específica não diz muito sobre a qualidade de seus vizinhos, e assim o processo de pesquisa não aprende para se mover para regiões mais promissoras do

espaço de pesquisa. Também argumentam que, se o número de vizinhos for muito pequeno, a relação de vizinhança não será exata. Exatidão, neste caso, significa que, para qualquer solução s_i, há pelo menos um caminho (bilateral) finito de soluções vizinhas [x_i, x_{i+1}, x_{i+2},..., x_{opt}] para uma solução globalmente ótima x_{opt}, que vem com uma seqüência monotonicamente decrescente dos valores da função objetiva [$f(x_i)$, $f(x_{i+1})$, $f(x_{i+2})$,..., $f(x_{opt})$]. Se esse não for o caso, uma pesquisa de melhoramento local pode ficar presa numa solução localmente ótima, sem obter o ótimo global.

2.2.4 Noções sobre heurística

O desenvolvimento de conceitos de complexidade computacional e a classificação de problemas difíceis, para os quais não parece existir algoritmo em tempo polinomial, têm fundamentado a utilização de técnicas heurísticas, sem causar grande constrangimento à não garantia de otimalidade. No entanto, a disponibilidade de recursos computacionais mais poderosos tem propiciado muita pesquisa e conseqüente desenvolvimento de métodos heurísticos muito eficientes, o que tem lhes dado um considerável respeito.

COLIN & REEVES (1993) e JOHNSON & MCGEOCH (1997) definem heurística (do grego *heuriskein* = descobrir) como uma técnica que procura boas (ou próximas às ótimas) soluções a um razoável custo computacional, sem ser capaz de garantir a viabilidade do ótimo, ou mesmo, em muitos casos, quão próxima do ótimo está uma particular solução viável encontrada. Heurísticas são procedimentos para resolver problemas através de um enfoque intuitivo, em geral racional, no qual a estrutura do problema possa ser interpretada e explorada inteligentemente para se obter uma solução razoável.

Na própria definição, percebem-se os dois parâmetros básicos e conflitantes de um método heurístico: a qualidade da solução gerada e o esforço computacional despendido. E é exatamente na possibilidade de minimização desse conflito que se encontra a causa para a explosão de interesse neste campo de estudo.

Segundo OSMAN (1991), as heurísticas podem ser divididas em construtivas, de melhoramento, programação matemática, particionamento, restrição do espaço de soluções, relaxação do espaço de soluções, algoritmos compostos e metaheuríticas.

As heurísticas costumam ser procedimentos mais simples e flexíveis que os métodos exatos de solução, permitindo, assim, abordar modelos mais complexos. Este é outro argumento a favor das heurísticas, tendo em vista que, com freqüência, os métodos exatos exigem simplificações nos modelos, tornando-os menos representativos do problema real.

No início dos estudos, a maioria das heurísticas eram específicas para um problema. Mais recentemente, têm sido desenvolvidas técnicas mais gerais, as meta-heurísticas, que, em princípio podem ser aplicadas a todos os problemas. Entre as mais conhecidas estão Busta Tabu, *Simulated Annealing,* Algoritmos Genéticos e Redes Neurais.

Muitos procedimentos heurísticos específicos e outros tantos baseados em meta-heurísticas têm sido desenvolvidos e têm demonstrado grande eficiência, quando aplicados a uma expressiva gama de problemas considerados de difícil solução. Mas, em verdade, o que mede, exatamente, a qualidade de uma heurística, se não temos garantia de otimalidade?

A avaliação de desempenho de heurísticas, no pior caso ou no caso médio, é quase sempre impraticável. Além do mais, esse tipo geral de análise não fornece informações precisas sobre performance em instâncias particulares do problema, motivo por que uma das metodologias mais utilizadas para avaliação de heurísticas é a experimentação empírica sobre problemas-testes. Entretanto, para garantir a significância dos resultados, é preciso ter critérios e parâmetros bem estabelecidos (LAWLER et al, 1985).

Problemas como o do Caixeiro Viajante, por exemplo, requerem a avaliação de um número grandioso de possibilidades para determinar uma solução exata. Em muitos casos, o tempo requerido para essa avaliação pode ultrapassar séculos. Heurísticas têm a função de indicar uma maneira de reduzir o número de possibilidades e obter uma solução num tempo aceitável.

2.3 O problema do caixeiro viajante

Nesta seção é apresentado o conceito do PCV, sua definição, formulação matemática, complexidade e algumas aplicações práticas.

2.3.1 Introdução

O termo Caixeiro Viajante foi utilizado pela primeira vez na década de 30 nos meios matemáticos. O autor do termo é desconhecido, porém o principal precursor de sua divulgação foi Merril Flood, motivado por propor um desafio intelectual por modelos matemáticos como o da teoria dos jogos (HOFFMAN & WOLFE, 1985).

O Problema do Caixeiro Viajante, PCV ou TSP - *the Traveling Salesman Problem* foi citado pela primeira vez, em 1954, por DANTZIG, FULKERSON e JOHNSON (1954), em um artigo publicado no *Jornal da Sociedade de Pesquisa da América* , sendo considerado o principal evento da história da Otimização Combinatória, o qual relatava uma solução para 49 cidades através de métodos de programação linear (HOFFMAN & WOLFE, 1985). O PCV tem, até hoje, se mantido no topo das pesquisas no que diz respeito aos problemas de otimização combinatorial, podendo ser considerado um dos problemas mais estudados, em diferentes áreas, como por exemplo, Matemática, Física, Biologia e Inteligência Artificial, fazendo surgir uma vasta literatura sobre esse problema. O motivo que despertou grande interesse por parte dos pesquisadores da área deve-se ao fato de ser um problema simples de descrever, mas muito difícil de resolver, além de possuir inúmeras aplicações práticas (JOHNSON & MCGEOCH ,1997).

No entanto, o Problema do Caixeiro Viajante é interessante não somente sob o ponto de vista teórico. Muitas aplicações práticas, que vão desde a fabricação de *chips* VLSI (KORTE, 1988 *apud* JOHNSON & MCGEOCH 1997) até a cristalografia de raio-X (BLAND & SHALLCROSS, 1989 *apud* JOHNSON & MCGEOCH, 1997), podem ser modeladas como o PCV ou como variações suas, havendo portanto, uma necessidade de algoritmos. O número de cidades em aplicações práticas pode variar de

algumas dezenas até alguns milhões. LAWLER et al. (1985) fazem excelentes abordagens sobre uma vasta coleção de publicações sobre este problema.

O PCV pertence à classe de problemas NP-completo, ou seja, o tempo gasto para resolvê-lo pode ser exponencial em relação ao tamanho da instância (GAREY & JOHNSON, 1979). Devido a essa disparidade, a resolução do problema utilizando métodos heurísticos ganha maior importância, principalmente quando aplicado a instâncias grandes do problema, para as quais poucas alternativas têm restado aos pesquisadores.

Para tentar desenvolver algoritmos de otimização que trabalhem bem com instâncias do mundo real, os cientistas vêm há décadas pesquisando técnicas de otimização, combinadas com o contínuo e rápido aumento na velocidade e capacidade dos computadores, juntamente com o mais famoso problema de otimização combinatorial, que tem sido base para testes de numerosas técnicas das mais variadas origens.

Essa vasta pesquisa na área tem levado o registro de solução ótima para instância não trivial do PCV de 318 cidades (CROWDER & PADBERG, 1980), para 2.392 cidades (PADBERG & RINALDI, 1987 *apud* JOHNSON & MCGEOCH 1997), para 7.397 cidades (APPLEGATE et al., 1994 *apud* JOHNSON & MCGEOCH 1997), e mais recentemente para 13.509 cidades (APPLEGATE et al., 1998).

2.3.2 Definição do PCV

A um caixeiro viajante é informado um conjunto de cidades e um custo c_{ij} associado a cada par de cidades *i* e *j* deste conjunto, representando a distância de ir da cidade *i* à cidade *j*. O caixeiro deve partir de uma cidade inicial, passar por todas as demais, uma única vez, e retornar à cidade de partida. Considerando que todas as cidades são interligadas, o problema consiste em se determinar uma rota que torne mínima a distância total. A distância pode ter, dependendo do contexto, outras denominações, tais como tempo ou dinheiro.

Formalmente, o PCV pode ser definido considerando-se um grafo completo. Dado G = (V, A) um grafo onde V é um conjunto de *n* vértices e A é o conjunto de

arcos ou arestas que conectam cada par de cidades *i* e *j* ? V. A cada arco/aresta está associado um custo c_{ij}. O PCV consiste em encontrar a rota de menor custo, passando em cada vértice uma única vez. No caso simétrico $c_{ij} = c_{ji}$ para toda cidade *i, j* ? V, enquanto que o caso assimétrico possui pelo menos um caso em que c_{ij} ? c_{ji}.

Embora a definição do problema seja normalmente referenciada para um problema de minimização, o problema também é encontrado na sua forma de maximização.

2.3.3 Formulação do PCV

Para formular uma notação que represente o número R(n) de rotas para o caso de n cidades, basta seguir a seguinte lógica: considerando que o Caixeiro parte de uma cidade determinada, sendo que essa primeira primeira não afeta o cálculo, a próxima escolhida deve ser retirada do conjunto das (n –1) cidades restantes. A seguinte, obviamente, sairá do conjunto das (n-2) cidades restantes, e assim, sucessivamente. Dessa forma, através de um raciocínio combinatorial simples e clássico, conclui-se que o conjunto de rotas possíveis, do qual o Caixeiro deve escolher a menor, possui cardinalidade dada por:

$$(n - 1) \times (n - 2) \times (n - 3) \times ... \times 2 \times 1$$

Ou seja, o Caixeiro deve escolher sua melhor rota dentre um conjunto de (n – 1)! possibilidades. Além disso, deve ser observado que se a distância entre duas cidades for independente do sentido - distância simétrica, o número de circuitos diferentes será então de (n – 1)! / 2.

Seguindo uma lógica prática, por exemplo, no caso de n = 4 cidades (A, B, C e D), a primeira e a última posição são fixas, de modo que elas não afetam o cálculo; na segunda posição, pode-se colocar qualquer uma das três cidades restantes, B, C e D. Uma vez escolhida uma delas, pode-se colocar qualquer uma das duas restantes na terceira posição; na quarta posição não se tem nenhuma escolha, pois sobrou apenas

uma cidade; conseqüentemente, o número de rotas é 3 x 2 x 1 = 6, o que corresponde a 3!. Considerando que não importa o sentido em que se percorre o roteiro, o número total de rotas fica diminuído pela metade, ou seja, três rotas. De modo que, por dedução simples, pode-se fazer a seguinte notação fatorial: R(n) = (n - 1)!/2.

A formulação matemática é a mesma para ambos os casos, apenas indicando a função objetivo como sendo de *Min* (minimização) ou *Max* (maximização), conforme o caso. Na literatura a formulação é normalmente referenciada usando uma função de *Min:*

Função objetivo: $MIN \sum_{i=1}^{n} \sum_{j=1}^{n} c_{ij}.x_{ij}$

Sujeito a :

$$\sum_{i=1}^{n} x_{ij} = 1, \qquad j = 1, ..., n \qquad (1)$$

$$\sum_{j=1}^{n} x_{ij} = 1, \qquad i = 1, ..., n \qquad (2)$$

$$\sum_{i \in S} \sum_{j \in S} x_{ij} \leq |S| - 1, \qquad \forall S \subset V, S \neq \emptyset \qquad (3)$$

$$x_{ij} \in \{0, 1\} \qquad i, j = 1, ..., n \qquad (4)$$

A variável inteira $x_{ij} = 1$ indica que a cidade *j* é visitada logo após a cidade *i*, caso contrário $x_{ij} = 0$. A variável *n* representa o número de cidades do problema, *S* é um subconjunto do conjunto {1, 2, ..., *n*}, e o símbolo "| |" denota a cardinalidade do conjunto. A função objetivo representa a minimização do somatório das distâncias entre as cidades da rota. As restrições (1) e (2) garantem que, para cada cidade $i \in n$, há exatamente uma conexão de chegada e uma partindo para outra cidade. A restrição (3) garante a não existência de sub-rotas (rotas que não incluam todas as *n* cidades) e a restrição (4) define *x* como variável binária.

2.3.4 A complexidade do PCV

O problema do caixeiro é um clássico exemplo de problema de otimização combinatória. A primeira alternativa em que podemos pensar para resolver esse tipo de problema, é reduzi-lo a um problema de enumeração, no qual acharíamos todas as rotas possíveis e, usando um computador, calcularíamos o comprimento de cada uma delas, para então avaliarmos qual a menor. Porém, o uso dessa técnica de força bruta não é recomendável para a maioria dos casos, principalmente quando o número de cidades é muito grande, como já vimos na seção2.2.

O PCV tem o número de *computações* executadas no melhor algoritmo conhecido para o problema, crescendo exponencialmente em função do tamanho da instância. Nesse contexto, por *computações* subentendem-se operações primitivas (atribuição, soma, etc.), conforme Cormen et al., (1996). Ainda podemos ressaltar que o PCV pertence à classe dos problemas NP-Completo.

Para exemplificar a complexidade do problema no que se refere ao tempo gasto para sua execução, será dado um exemplo retirado do Grupo de Pesquisa em Matemática, do Instituto de Matemática da UFRGS (GPM/UFRGS, 2001).

Parta-se da suposição da existência de um computador capaz de fazer um bilhão de adições por segundo. Com efeito, no caso de 20 cidades, ele precisaria de apenas 19 adições para dizer qual o comprimento de uma rota; logo, seria capaz de calcular 10^9 / 19 = 53 milhões de rotas por segundo. Contudo, essa astronômica velocidade é um nada frente à imensidão de (19! / 2) de rotas de que se precisará examinar. Com efeito, o valor aproximado desse número é 6,0 x 10^{16}. Conseqüentemente, o computador precisaria de

$$6{,}0 \times 10^{16} / 53{,}0 \times 10^{6} = 1{,}13 \times 10^{9} \text{ segundos}$$

para completar sua tarefa, o que equivale a cerca de 36 anos, aproximadamente.

Observa-se que o aumento no valor do *n* provoca uma lentíssima diminuição na velocidade com que o computador calcula o tempo de cada rota (ela diminui apenas de um sexto quando *n* aumenta de 5 para 25), entretanto provoca um aumento imensamente grande no tempo total de cálculo. Em outras palavras: a inviabilidade computacional é

devida à presença do fatorial na medida do esforço computacional do método. Com efeito, se essa complexidade fosse expressa em termos de um polinômio em *n*, como, por exemplo, n^5, o computador proposto anteriormente seria perfeitamente capaz de suportar o aumento de *n*, conforme mostra a tabela 2.3.

n	Rotas/s	(n – 1)! / 2	cálculo total	n^5	cálculo total
5	250 milhões	12	insignificante	3.125	Insignificante
10	110 milhões	181.440	0,0015 seg.	100.000	Insignificante
15	71 milhões	$4{,}35 \times 10^8$	10 min.	759.375	0,01 seg.
20	53 milhões	$6{,}0 \times 10^{16}$	36 anos	3.200.000	0,06 seg.
25	42 milhões	$6{,}2 \times 10^{23}$	235×10^6 anos	9.765.625	0,23 seg.

Tabela 2.3 – Esforço computacional

2.3.5 Aplicações

O número de problemas que podem ser modelados como um problema de caixeiro viajante é muito grande. Além disso, o PCV possui muitas aplicações práticas. Nesta seção, algumas aplicações práticas serão observadas, com o objetivo de justificar a importância desse problema, pois além da facilidade na sua formulação, não há a exigência de nenhum fundamento matemático para entendê-lo e nenhum grande talento para encontrar boas soluções para problemas grandes, o que torna o trabalho de seus estudiosos quase que um divertimento recreativo (LAWLER et al., 1985).

APPLEGATE et al. (1998) relata em seu trabalho inúmeras aplicações para o PCV, além nos itens descritos a seguir, que servem para mostrar como são vastas as áreas em que se podem utilizar algoritmos de otimização baseados no PCV .

2.3.5.1 Seqüenciamento de tarefas I

Suponha que *n* tarefas devam ser processadas seqüencialmente em uma determinada máquina. O tempo de preparo da máquina para processar a tarefa *j* imediatamente após a tarefa *i* é designado por c_{ij}. O problema de encontrar uma seqüência de execução para as tarefas, de forma a minimizar o tempo total de processamento, pode ser modelado como um PCV (LAPORTE, 1992).

2.3.5.2 Projeto de redes com restrições de conectividade

Suponha que você está projetando uma rede de computadores com algumas restrições de conectividade. Nesta rede, para alguns computadores importantes deve haver sempre a possibilidade de comunicação entre eles; já outros são menos importantes e podem servir apenas como um nó intermediário para conectar os computadores principais. Com este conjunto de restrições em mãos e o custo para conectar diretamente dois computadores através de fibras óticas é possível projetar a rede de menor custo, respeitando as restrições de conectividade.

Poderia se pensar em adicionar restrições do tipo: mesmo que uma conexão ligando dois computadores fique temporariamente fora do ar, a rede projetada deve permitir uma rota alternativa que ligue todos os computadores principais.

2.3.5.3 Roteamento de veículo

Em geral, o problema de roteamento de veículo consiste em determinar, para uma frota de carros, quais clientes seriam atendidos e por quais veículos e qual a ordem em que cada veículo visitaria seus clientes. Outros problemas podem ser derivados dessa idéia.

a) Distribuição de bens de consumo: Dado um conjunto de fregueses que precisam receber mercadorias, a fábrica tem que decidir a quantidade de carga a ser colocada em cada caminhão e quais caminhões irão atender quais clientes. Além disso, é

preciso otimizar as rotas dos veículos e, em alguns casos, levar em consideração a eventual necessidade de reabastecimento da carga de alguns caminhões. Por exemplo, os clientes podem ser bares e a fábrica pode ser uma fábrica de bebidas.

b) Localização de facilidades: dado um conjunto de clientes que precisam ser atendidos e um conjunto de possíveis locais para instalação de facilidades, deseja-se determinar quais os melhores locais para instalação das facilidades, de forma que todos os clientes sejam atendidos a um custo mínimo. Por exemplo, os clientes podem ser um conjunto de casas, e as facilidades podem ser instalar postos de pronto-socorro. O custo pode estar relacionado com a distância das casas ao pronto-socorro mais próximo.

2.3.5.4 Cristalização com raio-X

Outra aplicação direta do TSP ocorre na análise da estrutura de cristais (JÜNGER et al., 1995). Aqui um difratômetro de Raio-X é usado para obter informações a respeito da estrutura do material cristalino. Para esse fim, um detector mede a intensidade das reflexões de Raio-X do cristal em várias posições. Considerando que a medição por si própria pode ser executada rapidamente, há uma considerável sobrecarga no tempo de posicionamento, pois até 30.000 posições devem ser realizadas para alguns experimentos. Neste exemplo, o posicionamento envolve a movimentação de quatro motores. O tempo necessário para o movimento de uma posição para outra pode ser computado de forma precisa. A seqüência em que as medidas são tomadas para várias posições é irrelevante. Conseqüentemente, a melhor seqüência para as medidas a fim de minimizar o tempo de posicionamento total tem de ser determinada. O problema pode ser modelado como um TSP simétrico.

2.3.5.5 Projeto de circuitos integrados

Os seguintes tipos de problemas ocorrem repetidamente no projeto de sistemas de *hardware.* Tais sistemas possuem muitos módulos, cada um contendo sua própria pinagem, isto é, uma certa quantidade de pinos. A posição física de cada módulo é pré-

determinada. Para interconectar um dado conjunto de pinos por fios, precisa-se estar sujeito a determinadas condições: (i) no máximo dois fios podem ligar dois pinos (devido ao seu tamanho e possíveis mudanças futuras no *layout*); (ii) o comprimento total dos fios deve ser minimizado (para facilidade e organização da fiação). Desse modo, o problema da fiação torna-se um PCV com (n+1) cidades.

2.3.5.6 Seqüenciamento de tarefas II

Em certas fábricas, um produto final é criado a partir da execução de uma seqüência de pequenas tarefas. Essas tarefas podem possuir regras de precedência entre si, e particularidades que exigem um ou outro tipo de máquina, em determinado estado, para sua execução. Considere-se, particularmente, uma seqüência de *n* tarefas numa única máquina. As tarefas podem ser realizadas em qualquer ordem e o objetivo é completá-las no menor tempo possível. Destaque-se que a máquina deverá estar no estado S_j (que pode ser: rotação, temperatura, pressão, espessura de corte, cor de pintura, ou qualquer outro estado) para fazer a tarefa j e que o estado inicial e final para a máquina é S_0.

Considere-se que o tempo utilizado para completar a tarefa j diretamente após a tarefa i seja,

$$t_{ij} = c_{ij} + p_j,$$

onde c_{ij} é o tempo requerido para transformar a máquina do estado S_i para S_j e p_j é o tempo real para realizar a tarefa j (com $p_0 = 0$). Para um dado ciclo de permutação p de {0, ..., n}, o tempo requerido para completar todas as tarefas é

$$\sum_{i=0}^{n} \left(c_i p_{(i)} + p p_{(i)} \right) = \sum_{i=0}^{n} c_i p_{(i)} + \sum_{j=1}^{n} p_j$$

Visto que a soma de todos os p_j‘s é uma constante, segue que o problema de seqüenciamento de tarefas é um PCV.

2.3.5.7 Controle de robôs

Para que seja possível fabricar alguma peça, um robô tem que realizar uma seqüência de operações (corte de ranhuras, etc.). A tarefa resume-se em determinar uma seqüência para realizar as operações necessárias, a fim de minimizar o tempo de processamento. Surge uma dificuldade nessa aplicação, uma vez que há restrições de precedência. Dessa forma, temos o problema de encontrar o caminho hamiltoniano mais curto (onde as distâncias irão corresponder aos tempos necessário na mudança de posicionamento, bem como de escolha de ferramentas) de forma a satisfazer certas relações de precedência. Recebendo como dado de entrada a matriz de distâncias entre cada cidade *i, j*, o problema de caminho hamiltoniano pode ser definido como o problema de encontrar um caminho que, partindo de uma cidade inicial, passe uma única vez por todas as demais cidades, sem retornar ao ponto de partida.

2.3.5.8 Atribuições de freqüências em telefonia celular

Considere agora uma empresa que tem várias torres (antenas) que cobrem uma determinada região por onde trafegam usuários de telefone celular. Quando duas antenas muito próximas recebem as mesmas freqüências, a região que sofre a cobertura dessas antenas apresenta muita interferência. O objetivo é atribuir as freqüências (de quantidade limitada) para as antenas de maneira a minimizar a interferência total da atribuição.

CAPÍTULO 3

Simulated Annealing

Este capítulo aborda o método de meta-heurística conhecido por *simulated annealing*, integrante dos chamados métodos computacionais inteligentes (TANOMARU, 1995), apresentando inicialmente uma introdução do método, o modelo original da física, a partir do qual foi inspirado, para, então, introduzir sua estrutura e seus elementos básicos. Finaliza com algumas considerações teóricas que garantem a eficiência desse método.

3.1 Introdução

Simulated Annealing é um método heurístico para encontrar boas soluções para problemas difíceis de otimização. A técnica foi aplicada pela primeira vez a problemas de otimização combinatória por KIRKPATRICK et al. (1983), e posteriormente por CERNY (1985). Eles mostraram que conceitos da física para simular processos de recozimento, propostos inicialmente por METROPOLIS et al. (1953), poderiam ser estendidos para resolver problemas de otimização em geral, especialmente problemas de otimização combinatória.

O método *Simulated Annealing* pode ser facilmente explicado como sendo uma extensão da simples e familiar heurística de busca local. Esse tipo de busca requer somente a definição de um esquema de vizinhança e um método de avaliação do custo de uma solução particular. O algoritmo busca iterativamente a vizinhança da solução corrente para obter uma solução de melhor qualidade, o qual passará a ser a nova

solução. Quando não existirem soluções na vizinhança corrente que melhorem a qualidade da solução, o algoritmo termina em um ótimo local, o que, dependendo da solução inicial, pode ser muito pobre.

A armadilha do ótimo local faz da busca local uma heurística pobre para muitos problemas de otimização combinatorial. A figura 3.1 ilustra essa situação de mínimo local. Uma propriedade desejável de qualquer algoritmo é a habilidade de achar uma boa solução, independente do ponto de partida.

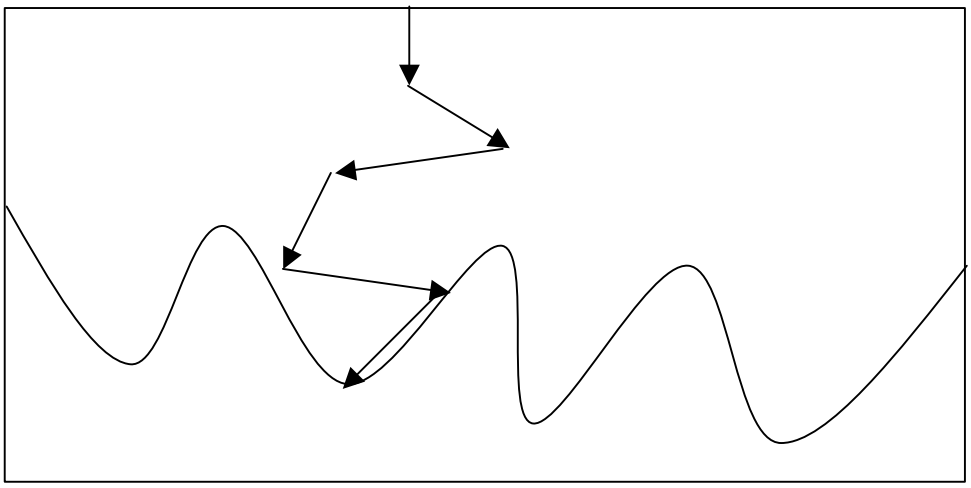

Figura 3.1 - Ilustração da estratégia da pesquisa local

Uma forma de escapar da armadilha do ótimo local é reiniciar a busca local de várias soluções iniciais diferentes e utilizar a melhor solução encontrada como solução do algoritmo. Tal estratégia reduz a dependência da solução inicial, mas apresenta um outro problema que consiste em determinar quando parar o algoritmo. Observa-se que repetidas buscas locais convergem assintoticamente para a solução ótima usando todas as soluções como ponto de partida, o que não é viável em problemas grandes.

3.2 Analogia física

O algoritmo *simulated annealing* na sua versão original, é utilizado em termodinâmica na simulação do processo de recozimento de um sólido, quando se

pretende alcançar o seu estado com energia mínima. Em seguida, é apresentado o algoritmo como uma ferramenta utilizada na resolução de problemas de otimização.

O nome recozimento (*annealing*) é dado ao processo de aquecimento de um sólido até o seu ponto de fusão, seguido de um resfriamento gradual e vagaroso, até que se alcance novamente o seu enrijecimento. Nesse processo, o resfriamento vagaroso é essencial para se manter um equilíbrio térmico no qual os átomos encontrarão tempo suficiente para se organizarem em uma estrutura uniforme com energia mínima. Se o sólido é resfriado bruscamente, seus átomos formarão uma estrutura irregular e fraca, com alta energia, em conseqüência do esforço interno gasto.

Computacionalmente, o recozimento pode ser visto como um processo estocástico de determinação de uma organização dos átomos de um sólido que apresente energia mínima. Em temperatura alta, os átomos se movem livremente e, com grande probabilidade, podem mover-se para posições que incrementarão a energia total do sistema. Quando se baixa a temperatura, os átomos gradualmente se movem em direção a uma estrutura regular e, somente com pequena probabilidade, incrementarão suas energias.

Segundo METROPOLIS et al. (1953), quando os átomos se encontram em equilíbrio, em uma temperatura *T*, a probabilidade de que a energia do sistema seja *E*, é proporcional à $e^{-E/kT}$, onde *k* é conhecida como constante de Bolzmann. Desta forma, a probabilidade de que a energia de um sistema seja (*E* + *dE*) pode ser expressa por:

$$prob\,(E + dE) = prob\,(E)\,prob\,(dE) = prob\,(E)\,e^{-dE/kT}$$

Em outras palavras, a probabilidade de que a energia de um sistema passe de *E* para *(E + dE)* é dada por $e^{-dE/kT}$. Na expressão $e^{-dE/kT}$, como *k* é uma constante, observa-se que à medida que *T* diminui, a probabilidade da energia do sistema se alterar é cada vez menor. Nota-se, também, que, nessas condições, quanto menor o valor de *dE*, maior a probabilidade de a mudança ocorrer.

O processo de Metropolis é constituído basicamente por duas etapas. Na primeira, procede-se à elevação inicial da temperatura a um estado de energia máxima e, na segunda, verifica-se o seu abaixamento sucessivo e suficientemente lento para que as

partículas do sistema se combinem de forma a atingirem o estado de energia mínima, isto é, de tal forma que seja atingido o equilíbrio térmico.

A simulação da evolução das soluções é baseada em técnicas de Monte Carlo e na geração de estados sucessivos. Supõe-se como ponto de partida um estado possuindo energia E_i a partir do qual, recorrendo a um mecanismo apropriado, é gerado um outro estado com energia E_j . Se a diferença de energias E_j -E_i for inferior ou igual a zero, o novo estado é aceito como estado atual. Se a condição anterior não se verificar, o novo estado poderá ser ainda aceito com uma provável função da diferença de energias entre estados sucessivos e da temperatura do banho. A expressão a seguir, apresenta o processo de cálculo dessa probabilidade.

$$p = e^{-\frac{E_i - E_j}{K_B T}}$$

T Temperatura

K_B Constante de Boltzman

O processo de arrefecimento simulado tem a particularidade de permitir o relaxamento transitório da optimalidade durante o processo de pesquisa. Esse relaxamento é admitido na tentativa de evitar a convergência para óptimos locais.

Se a redução da temperatura for suficientemente lenta, o sólido pode encontrar o equilíbrio térmico em cada temperatura. No algoritmo de Metropolis, isto é obtido gerando um grande número de transições em uma determinada temperatura. O equilíbrio térmico é caracterizado pela distribuição de *Boltzmann*. Essa distribuição dá a probabilidade de sólidos estando no estado i com energia E_i na temperatura T, e é dado por

$$P_T\{X=i\} = \frac{1}{Z(T)} \exp\left(-\frac{E_i}{k_B T}\right),$$

onde X é uma variável estocástica, denotando o estado corrente do sólido. Z(T) é a *função partição,* que é definida como

$$Z(T) = \sum_j \exp\left(\frac{-E_j}{k_B T}\right),$$

onde o somatório abrange todos os possíveis estados.

O método computacional que imita esse processo de recozimento de um sólido é chamado de *Simulated Annealing.*

3.3 Descrição do algoritmo

Simulated Annealing é considerado um tipo de algoritmo conhecido como de busca local. Ele se constitui em um método de obtenção de boas soluções para problemas de otimização de difíceis resoluções. Desde a sua introdução como um método de otimização combinatorial, esse método vem sendo vastamente utilizado em diversas áreas, tais como projeto de circuitos integrados auxiliado por computador, processamento de imagem, redes neuronais, etc.

A sua semelhança com o método original, no qual foi inspirado, é muito grande. Na sua apresentação, nos trabalhos independentes de KIRKPATRICK et al. 1983 e CERNY, 1985, ele é mostrado como um modelo de simulação de recozimento de sólidos. Como proposto em METROPOLIS et al. (1953), pode ser utilizado em problemas de otimização, onde a função objetivo, a ser minimizada, corresponde à energia dos estados do sólido.

Antes de passar para a descrição do algoritmo propriamente dito, será definido o problema de otimização que se pretende resolver. Para tanto, seja *S* o espaço total de soluções de um problema combinatorial, em que *S* é o conjunto finito que contém todas as combinações possíveis que representam as soluções viáveis para o problema. Seja *f* uma função de valores reais, definida sobre *S*, $f : S \rightarrow R$, o problema se constitui em encontrar uma solução (ou estado) $i \in S$, tal que *f(i)* seja mínimo.

Uma das formas mais simples de tentar resolver o problema, utilizando busca local em *S*, conhecida como algoritmo descendente, é iniciar o processo de busca por uma solução, normalmente, tomada de forma aleatória. Uma outra solução *j* é então

gerada, na vizinhança desta, por meio de um mecanismo apropriado e dependente do problema. Caso uma redução do custo dessa nova solução seja verificada, ou seja, $f(j)<f(i)$, a mesma passa a ser considerada a solução corrente e o processo se repete. Caso contrário, a nova solução é rejeitada e uma outra gerada. Esse processo se repete até que nenhum melhoramento possa ser obtido na vizinhança da solução corrente, após um número determinado de insistências. O algoritmo retorna, então, o valor da última solução corrente, considerada uma solução de mínimo local.

O grande problema desse método, muito simples e rápido, é que o mínimo local encontrado pode estar longe de ser um mínimo global, o que se traduziria em uma solução inaceitável para o problema. Uma estratégia muito simples de aprimorar a solução obtida, através desse tipo de algoritmo, seria escolher a menor solução dentre um conjunto de soluções obtidas de execuções sucessivas, realizadas a partir de diferentes soluções iniciais.

O método *Simulated annealing* não utiliza essa estratégia. Esse método tenta evitar a convergência para um mínimo local, aceitando, às vezes, uma nova solução gerada, mesmo que essa incremente o valor de *f*. O aceite ou a rejeição de uma nova solução, que causará um incremento de *d* em *f*, em uma temperatura *T*, é determinado por um critério probabilístico, através de uma função *g* conhecida por função de aceite. Normalmente, essa função é expressa por

$$g(d, T) = e^{-d/T}$$

Caso $d = f(j) - f(i)$ for menor que zero, a solução *j* será aceita como a nova solução corrente. Caso contrário, a nova solução somente será aceita se

$$g(d, T) > random(0, 1)$$

É grande a semelhança com o método original de simulação de recozimento na termodinâmica, como já aludido, pois o parâmetro *d* corresponde à variação da energia de um estado para outro *(dE)* e o parâmetro de controle *T* corresponde à temperatura. Uma vez que, agora *T* é imaginário, a constante *K*, que aparecia na expressão original multiplicando *T*, é considerada igual a *1*.

Da mesma forma que no processo físico, a função *g (d , T)* implica que a probabilidade de aceite de uma nova solução é inversamente proporcional ao incremento de *d*, e quando *T* é alto, a maioria dos movimentos (de um estado para outro ou de uma solução para outra) é aceita, entretanto, à medida que *T* se aproxima de zero, a grande maioria das soluções são rejeitadas.

Procurando evitar uma convergência precoce para um mínimo local, o algoritmo inicia com um valor de *T* relativamente alto. Esse parâmetro é gradualmente diminuído e, para cada um dos seus valores, são realizadas várias tentativas de se alcançar uma melhor solução nas vizinhanças da solução corrente.

O algoritmo *Simulated Annealing* oferece uma forma de escapar do ótimo local, analisando a vizinhança da solução corrente e aceitando a solução que traga melhora, mas que aceite também soluções que piorem a atual solução corrente com uma probabilidade que é menor quanto maior for a distância entre essa solução e a solução corrente. A condição para aceitar ou rejeitar um movimento que aumente a função de custo (ou seja, solução pior) é determinada por uma seqüência de números randômicos, mas com uma probabilidade controlada. A probabilidade de aceitar um movimento que aumente a função de custo é chamada de função de aceite e é normalmente representada por exp (- ? / T), onde ? é a diferença entre as soluções, e T é um parâmetro de controle que corresponde à temperatura, fazendo uma analogia com *annealing* físico. Essa função de aceite implica que os pequenos aumentos da função de custo são provavelmente mais aceitos que os grandes e que, quando a temperatura T é alta, mais movimentos são aceitos, mas, quando a temperatura T se aproxima de zero, muitos movimentos que aumentam a função de custo serão rejeitados.

Assim, o algoritmo *Simulated Annealing* é iniciado com um valor de temperatura T relativamente alto para evitar que prematuramente fique preso a um mínimo local. O algoritmo então prossege tentando um certo número de movimentos na vizinhança em cada temperatura, enquanto o parâmetro temperatura é gradualmente reduzido. A figura 3.2 é uma ilustração de como o algoritmo *Simulated Annealing* caminha para fugir de um mínimo local:

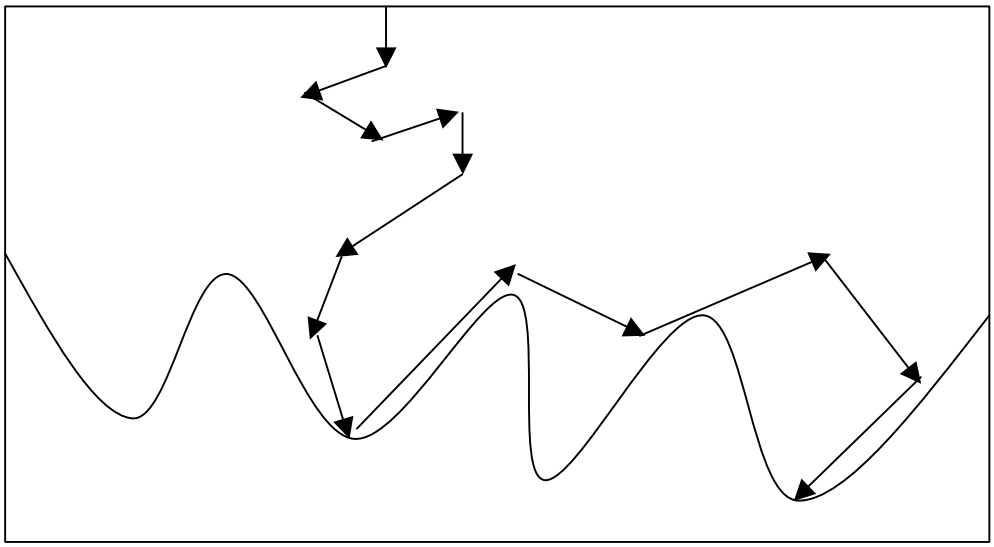

Figura 3.2- Ilustração da estratégia do *Simulated Annealing*

Um pseudocódigo é apresentado na figura 3.3.

Procedimento *Simulated Annealing*

```
início
        S ✍ S0;
        T ✍ T0;
        enquanto temperatura elevada faça
        para interações para equilíbrio faça
            gerar uma solução S' de N(S);
                avaliar a variação de energia – ?E = f(S') – f(S);
                    se ?E < 0 then
                            S ✍ S';
                     senão
                            gerar u ? random[0,1];
                            se u < exp(-?E/KB . T) then
                                    S ✍ S';
                     fimse
                fimse
            fimpara
            reduzir a temperatura T;
        fimenquanto
fimprocedimento
```

Figura 3.3 – Pseudocódigo do algoritmo *Simulated Annealing*

3.4 Aspectos do algoritmo

Na construção de algoritmos SA é importante a especificação de alguns parâmetros genéricos (REEVES, 1985), tais como: valores inicial e final de temperatura, número de iterações e regra de decréscimo de temperatura, os quais definem uma estratégia de resfriamento.

Para estabelecer de forma empírica o *valor inicial de temperatura* a ser utilizado, deve ser considerado que o valor deva ser "alto", desta forma a maioria ou, eventualmente, todas as transições serão aceitas. Na prática, isto pode requerer algum conhecimento da magnitude da vizinhança na solução específica. Não havendo este conhecimento, escolhe-se um valor que pareça ser alto, executa-se o algoritmo por um tempo curto e observa-se a taxa de aceitação. Se essa taxa for adequadamente alta, o valor testado para temperatura pode ser usado, significando que uma taxa de aceitação alta pode variar de uma situação para outra, mas, em muitos casos, valores entre 40% e 60% são bons resultados (REEVES, 1985). É possível a utilização de métodos mais sofisticados para a escolha do valor inicial de temperatura, levando em consideração o conceito de entropia (RODRIGUES & ANJO, 1993). Quanto à temperatura final, na teoria, o processo deveria continuar até que a temperatura fosse igual a zero. Na prática, é suficiente que a parada ocorra quando a chance de ocorrer aumentos na função custo (para minimização) seja negligenciável.

A prescrição de resfriamento (*cooling schedule*), que inclui a regra de decréscimo de temperatura e o número de iterações, tem sido implementada de várias maneiras, mas as formas tradicionais seguem um modelo baseado em cadeias de Markov (REEVES, 1985) (AARTS & KORST, 1990).

No procedimento de *annealing* a temperatura é fixa até que o equilíbrio seja encontrado, com o aspecto adicional de decidir quando isso ocorre; uma vez considerado que este estado foi alcançado, a temperatura é reduzida e o procedimento repetido; o número de transições pode ser muito grande a cada temperatura e o número de etapas pode ser relativamente grande também.

Já a temperatura é reduzida lentamente a cada transição, o que é menos complicado e mais usado na prática. Em qualquer um dos casos, é necessário decidir o formato da curva de resfriamento.

Entre os métodos de resfriamento mais populares e simples estão a prescrição geométrica, t ¬ a t, onde *a* é uma constante próxima a 1 e a prescrição na forma t ¬ (t / (1+ b t) onde *b* é uma constante com valor próximo a zero. O número de iterações depende dos valores atribuídos às temperaturas inicial e final e das regras de decréscimo da mesma, encontrando-se fórmulas consideradas adequadas por alguns autores.

Em termos gerais, tem sido mostrado que o algoritmo converge para um conjunto de soluções ótimas globais quando o tempo tende para o infinito. Esta propriedade de convergência assintótica é provada por meio de um modelo de cadeia de Markov. Entretanto, quando implementando o algoritmo, considerações práticas devem ser dadas ao programa de resfriamento para assegurar convergência para uma solução de boa qualidade, em um tempo razoável.

3.4.1 Resfriamento geométrico

KIRKPATRICK et al. (1983) propôs o método de resfriamento geométrico que ainda permanece largamente utilizado. Esse programa não é baseado em qualquer justificativa teórica, bem como não se preocupa com a analogia física. Cabe salientar que há muitos parâmetros a serem ajustados para um determinado problema, mas, uma vez encontrados bons parâmetros para o problema, poucos ajustes serão necessários para outras instâncias do mesmo.

O parâmetro da temperatura inicial T_0 deve ser suficientemente grande para permitir virtualmente que todas as transições sejam aceitas. Porque e o valor apropriado de T_0 varia de problema para problema e de instância para instância, uma metodologia genérica é proposta para se obter uma temperatura suficientemente quente para todos os problemas. Executando o *Simulated Annealing*, inicialmente com uma temperatura pequena, a taxa de aceite ?(T) pode ser determinada pela proporção entre o número de transições aceitas e o número *l* de transições propostas usando o teste usual de Metropolis. Se o valor de ?(T) é maior que a taxa de aceite inicial desejada $?_0$, então a temperatura inicial T_0=T; caso contrário, T é incrementado por um fator ? > 1, T = ?T.

O processo é repetido k vezes até que o valor de T forneça uma taxa de aceite ?(T) > ?$_0$, ponto em que T_0 recebe o valor corrente de T.

Uma sugestão para o valor inicial de T é: T = ln f(x_0), onde f(x_0) é o valor da função objetivo da solução inicial. O valor de l sugerido é: l ? ?h?|$N(x,?)$|?, onde |N(x,?)| é o tamanho da vizinhança e h é uma constante multiplicativa. Se o valor de l for muito pequeno, então a medida da taxa de aceite torna-se muito imprecisa; caso contrário, a taxa de aceite terá grande precisão, mas poderá haver um grande desperdício de tempo. Se o valor de ? for muito grande, haverá o risco de ultrapassar em muito a temperatura inicial ideal, caso contrário, também pode haver um gasto excessivo de tempo.

A maneira pela qual a temperatura é reduzida dá o nome ao programa de resfriamento. A temperatura é reduzida pela multiplicação por um fator fixo ? < 1:

$$T_{k+1} = ? T_k$$

Sendo que, de acordo com a teoria, a temperatura deve ser reduzida lentamente, o valor de ? é usualmente selecionado entre 0.8 e 0.99, com tendência a valores próximos de 1 (dando uma redução de temperatura mais lenta). O valor de ? usado no algoritmo afeta o tempo de execução e a qualidade desejada para a solução. Se o valor dele for pequeno, então a temperatura resfria rapidamente, levando o algoritmo a parar rapidamente, mas levando a soluções pobres. Se ? estiver próximo de 1, então a temperatura reduz lentamente, exigindo maior tempo de execução e obtendo soluções de melhor qualidade.

O programa de resfriamento geométrico tenta reestabelecer o equilíbrio após cada redução de temperatura, executando um número L fixo de iterações de movimentos pela vizinhança. O número de transições requeridas para restabelecer esse equilíbrio depende da variação da temperatura e do tamanho da instância do problema. O programa de resfriamento geométrico ignora a variação da temperatura e estabelece um tamanho suficiente, que pode ser atrelado ao tamanho do problema:

$$L ? ?c?|N(x,?)|?,$$

onde |N(x,?)| é o tamanho da vizinhança e *c* é uma constante multiplicativa.

O valor de c afeta o algoritmo de duas maneiras. Se *c* é pequeno, então cada cadeia consistirá em um pequeno número de transições propostas. Assim a temperatura irá resfriar rapidamente e o algoritmo terminará em um pequeno intervalo de tempo. Entretanto, cadeias pequenas não permitem ao sistema restabelecer o equilíbrio. Por outro lado, valores de c grandes exigem muito tempo de execução e, em conseqüência, esperam-se soluções de melhor qualidade.

O algoritmo *Simulated Annealing* deve parar quando estiver congelado. Como determinar quando o sistema está congelado não é claro, mas deve significar que há uma pequena ou nenhuma possibilidade de escapar da solução corrente. O método aqui utilizado é parar quando a solução corrente permanece a mesma para um número *s* de cadeias consecutivas. Essa condição de parada deve ocorrer somente em baixas temperaturas. O valor de *s* pode afetar o algoritmo provocando parada prematura, resultando em uma solução que não é a melhor que o algoritmo pode encontrar. Se o valor de *s* não afeta o algoritmo, então a condição de parada usada não está atuando o suficiente para reconhecer que o sistema já está congelado.

3.4.2 Resfriamento com tempo polinomial

O resfriamento com tempo polinominal, proposto inicialmente por AARTS & KORST (1990), tem mostrado teoricamente que o mesmo produz tempo de execução limitado por uma função polinomial do tamanho da instância do problema. Como a maioria dos programas de resfriamento, ele não garante qualidade da solução final produzida. A preocupação básica é atingir um equilíbrio aproximado em uma dada temperatura.

Se forem gerados alguns ensaios com uma dada temperatura T, obtém-se o valor m_1, denotando o número de transições propostas de *i* para j com f(j) ? f(i), e m_2 o número de transições para o qual f(j) > f(i); Seja $? \overline{?f}^{(?)}$ a variação média do custo para

as m_2 transições onde o custo aumenta, então a taxa de aceite ? pode ser aproximada pela seguinte expressão:

$$, \; ? \; ? \; \frac{m_1 \; ? \; m_2 \, . \exp ? \frac{? \; ? \; \overline{? f}^{(?)} \; ?}{T} ?}{m_1 \; ? \; m_2} \qquad (1.0)$$

derivando:

$$T \; ? \; \frac{\overline{? f}^{(?)}}{\ln ? \frac{m_2}{? m_2 \, . ? \; ? \; m_1 (1 ? \; ?)} ?} \, . \qquad (2.0)$$

Sabendo que a temperatura a ser calculada deve ser suficientemente alta para se obter uma taxa de aceite próxima de 1.0, então podem se realizar *m* ensaios aceitando todas as transições, e computa-se m_1 e m_2 sem a aplicação do critério de Metropolis, calculando $? \; \overline{? f}^{(?)}$. De posse desses valores e da taxa de aceite desejada ?, deve-se usar a fórmula 2.0 para a obtenção de T_0.

A função de redução da temperatura deve ter seu passos de redução, tais que sejam pequenos o suficiente para permitir que as cadeias de Markov sejam as menores possíveis para permitir a condição de “quase equilíbrio” dada abaixo:

$$? \, k \; ? \; 0 : || q(T_k) \; ? \; q(T_{k ? 1}) || ? \; ? \, ,$$

para algum valor positivo ?. Evidentemente, assume-se que a condição de quase equilíbrio existe em T_0.

Então, para duas temperaturas sucessivas, espera-se que as distribuições estacionárias estejam próximas. Isto pode ser quantificado por:

$$? i ? S: \frac{1}{1 ? ?} ? \frac{q_i(T_k)}{q_i(T_{k?1})} ? 1 ? ? \text{ , k=0,1,...,}$$

para algum número pequeno positivo, que pode corresponder a ? da condição de “quase equilíbrio”.

A partir daí, AARTS & KORST (1990) chegam à seguinte expressão:

$$T_{k?1} ? \frac{T_k}{1 ? \frac{T_k . \ln ?1 ? ? ?}{? f ?_{T_k} ? f_{opt} ? 3? _{T_k}}} \text{ , k=0,1,..}$$

Como nem sempre se conhece a f_{opt}, a solução ótima, pode-se omitir o termo $\langle f \rangle_{Tk} - f_{opt,}$ deixando que sua ausência seja contrabalançada pelo parâmetro ?.

O algoritmo deve terminar quando o custo médio para a temperatura tendendo a zero tender para o valor ótimo.

O algoritmo deve terminar na k-ésima iteração que satisfaça a condição:

$$\frac{T_k}{\langle f \rangle_?} \frac{? \langle f \rangle_T}{? T} \Bigg|_{T ? T_K} ? ?_S ,$$

onde $?_s$ é um número positivo pequeno, chamado *parâmetro de parada*.

Se nesse processo de resfriamento usarem-se cadeias de Markov de tamanho fixo igual ao tamanho da vizinhança, visitar-se-ão cerca de 2/3 das diferentes soluções dessa vizinhança, segundo AARTS & KORST (1990).

Usando a função de redução de temperatura dada anteriormente da seguinte forma:

$$T_{k?1} ? \frac{T_k}{1 ? ? _k T_k} \text{ , k=0,1,..}$$

$$\text{onde } ?_k ? \frac{\ln(1??)}{3?_{T_K}}, \text{ k=0,1, ...}$$

Se K é o primeiro inteiro que satisfaz o critério de parada, então K= O(ln(|S|).

Esse resultado é muito importante e sua prova pode ser encontrada em AARTS & KORST (1990). Um passo da prova consiste em mostrar que K é expresso como função de $T_{k,}$e o outro passo consiste em estabelecer um limite inferior para T_k.

Assim o algoritmo requer O(?.L.ln|S|) passos, onde L é o tamanho de cada cadeia de Markov, ln|S| denota o limite superior do número de cadeias de Markov e ? denota o tempo computacional para uma transição. Para a maioria dos problemas combinatórios, L e ? podem ser escolhidos para serem polinomiais. Como ln|S| é polinomial, então o *Simulated Annealing* executa em tempo polinomial.

3.4.3 Resfriamento com tempo linear

LUNDY & MEES (1986) sugerem uma função de redução de temperatura da seguinte forma:

$$t_k ? \frac{t_k}{1?? .t_k},$$

onde ? é um número pequeno que controla a redução de temperatura. Quando ele for pequeno, a queda é mais lenta, quando ? aumenta, a queda é mais rápida, gerando soluções de qualidade inferior às geradas para um ? menor.

Como o critério de parada pode ser expresso em termos de um valor mínimo para a temperatura ou em termos do "congelamento" do sistema, a proposta deste programa é:

$$t ? \frac{?}{\ln??|S|?1?/??},$$

onde S é o espaço de soluções. Deseja-se produzir soluções que estejam a ? da solução ótima com probabilidade ?.

Uma regra de parada simples consiste em se estabelecer um percentual de iterações a ser executado sem que haja melhora na solução, assim não haverá desperdício de tempo. No entanto, qualquer regra deve ser bem ajustada de forma a permitir uma queda suficiente da temperatura , a fim de que a convergência seja assegurada.

Nesta situação, uma condição que se considera importante não é a temperatura, mas a taxa de aceitação. Quando a média da taxa de aceitação atinge valor zero, significa que a variação do valor da função objetivo, visitada naquela iteração, é muito superior ao valor da temperatura. A partir daí, o algoritmo está em busca local, o que pode ser prorrogado por um certo tempo. Um valor que pode ser experimentado é executar *n.ln n* reduções de temperatura após a taxa de aceitação média atingir zero em uma dada temperatura. O critério adotado no programa de tempo polinomial pode ser colocado simultaneamente, ou seja, quando a média dos valores da função objetivo de uma temperatura para outra sofrer redução inferior a um parâmetro $?_t$, o que permite o algoritmo parar antes da *n.ln n* estabelecidas.

Além dos parâmetros genéricos mencionados no parágrafo anterior, um outro aspecto importante a ser considerado é a escolha da estrutura de vizinhança a ser considerado. Este aspecto é dependente do problema. Com freqüência, a estrutura de vizinhança deve ser escolhida de forma a tornar econômica, sob um ponto de vista computacional, a transição de uma solução para a subseqüente.

3.5 Considerações finais

A importância teórica e prática dos problemas de otimização combinatorial tem motivado um grande esforço de pesquisa na obtenção de soluções aceitáveis em tempo adequado de computação e no entendimento teórico, tanto dos problemas em si como dos algoritmos utilizados em sua resolução.

A utilização de algoritmos tendo como paradigma o *Simulated Annealing* é devida, em grande parte, a sua estrutura simples. A analogia com conceitos já estabelecidos em outras teorias incentivou um grande número de pesquisas teóricas, existindo uma fundamentação matemática já desenvolvida e documentada na literatura.

O esforço requerido pelo *Simulated Annealing* pode variar muito, dependendo do problema e do tamanho da instância, de alguns segundos para um problema pequeno a muitas horas para um problema grande. Diante desse pressuposto surge a necessidade de se investigarem várias modificações no algoritmo básico para acelerar sua convergência para uma solução de boa qualidade. AARTS & KORST (1990) identificaram três categorias gerais de abordagens para acelerar o algoritmo *Simulated Annealing*, que são: um algoritmo seqüencial mais eficiente, aceleração do *hardware* e projeto de algoritmo paralelo no qual este trabalho tem o enfoque principal.

Algumas acelerações do *hardware* sugeridas na literatura normalmente são muito difíceis e de alto custo. A aceleração mais acessível consiste na utilização de máquinas de uso geral com maior poder de processamento, porém, AARTS & KORST (1990) mostram que a alternativa de desenvolvimento de *Simulated Annealing* paralelo tira um grande proveito da disponibilidade de diversas máquinas em ambiente de redes.

CAPÍTULO 4

Computação Paralela

Este capítulo apresenta alguns tópicos relevantes sobre computação paralela, descrevendo seus conceitos básicos, arquiteturas, programação concorrente e sistemas distribuídos.

4.1 Introdução

A literatura apresenta várias definições para computação paralela, dentre as quais cabe citar a sugerida por QUINN (1987): “computação paralela é o processamento de informações que enfatiza a manipulação concorrente dos dados, que pertencem a um ou mais processos que objetivam resolver um único problema”; a apresentada por ALMASI (1994): “computação paralela constitui-se de uma coleção de elementos de processamento que se comunicam e cooperam entre si e com isso resolvem um problema de maneira mais rápida”; e por ANDREWS (1991), que define programação paralela como sendo “a atividade de se escrever programas computacionais compostos por múltiplos processos cooperantes, atuando no desempenho de determinada tarefa.”

Embora ainda conceitualmente intrigante, a computação paralela já se torna essencial no projeto de muitos sistemas computacionais, seja pela própria natureza inerentemente paralela apresentada por um sistema, seja pela minimização do tempo de processamento, ou mesmo pela busca de uma estruturação melhor e/ou mais segura de um sistema.

Segundo MAZZUCO (1999), atualmente a computação paralela vem auxiliando diversas outras tarefas no projeto de muitos sistemas computacionais, não atuando mais somente na eficiência de um processo no que tange ao tempo de execução, mas também, na busca de uma estruturação melhor e/ou mais segura para o sistema. Como exemplo podemos citar o gerenciamento de processos e usuários de uma rede, criando processos paralelos supervisores.

Com o aumento espantoso das informações e a necessidade de trabalhar estas informações de maneira rápida, a computação paralela torna-se uma aposta para o futuro. Além dos custos da tecnologia serem altos, os limites de natureza física impedem um contínuo incremento de performance, então, temos que pensar também em quantidade de processadores, e não só em qualidade.

A computação paralela apresenta, no entanto, muitas questões para serem resolvidas antes de passar a um uso mais generalizado. Um dos problemas mais sérios reside no fato dos algoritmos tradicionais, bastante intrincados na população da computação, terem de ser totalmente revistos antes que se possa explorar corretamente esta nova estrutura de computação, sendo necessário reavaliar todos os princípios na elaboração dos algoritmos, para que a solução paralela seja efetivamente mais rápida que a solução seqüencial.

Além disso, a necessidade de novas maneiras de organização do processamento computacional objetiva a eficiência através da quebra do paradigma de execução seqüencial do fluxo de execuções, conforme a filosofia de Von Neumann, citado por AMORIM, (1988), evitando, entre outras, a grande perda de performance ocasionada pela necessidade de tornar seqüenciais programas que são inerentemente paralelos.

Os sistemas paralelos oferecem um incremento à performance de programas lentos, soluções naturais para programas intrinsecamente paralelos e uma possível modularidade dos programas. Em geral, o paradigma apresenta algumas dificuldades a serem consideradas, como a dificuldade de programação, a necessidade de balanceamento de carga, a comunicação e o sincronismo entre os processos.

Conforme TANENBAUM (1992), os sistemas multiprogramados com um único processador são capazes de processar mais de um programa simultaneamente, compartilhando o tempo do processador entre os diversos processos, ao passo que os

sistemas multiprocessados compartilham os diversos processadores para executar cada um dos processos existentes, necessitando de um número igual ou superior de processadores em relação ao número de processos para caracterizar um processamento paralelo (TREW e WILSON, 1991).

Associado a esses conceitos, está a utilização de sistemas computacionais distribuídos, permitindo que as tarefas possam ser executadas em um *cluster*, explorando desta maneira uma maior potência computacional com relevante redução de custos.

4.2 Tipos de paralelismo

Desenvolver concorrentemente processos computacionais implica que, em um determinado instante, dois ou mais processos foram iniciados e ainda não terminados. Dentro dessa definição, temos dois tipos de paralelismo, denominados paralelismo físico e lógico ou ainda, respectivamente, real e pseudo paralelismo (TREW e WILSON, 1991) (KIRNER, 1991).

Paralelismo Lógico: Neste caso, tem-se apenas um elemento de processamento, e os processos são executados intercaladamente, de maneira que apenas um está ativo a cada instante. A figura 4.1 apresenta um gráfico que demonstra a execução de três processos (p1, p2 e p3) em relação ao tempo, exemplificado este tipo de paralelismo.

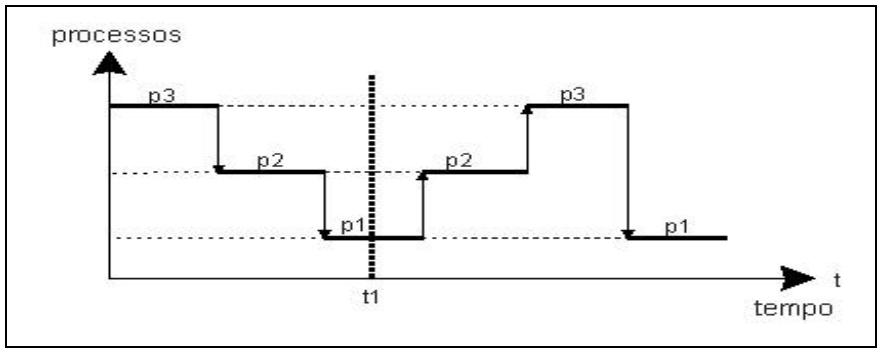

Figura 4.1 –Paralelismo Lógico.

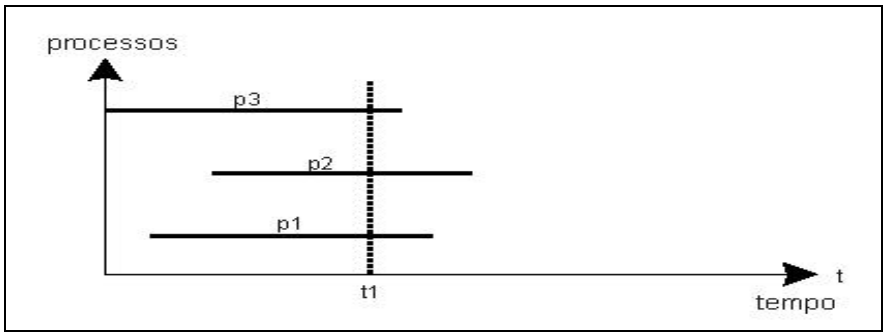

Figura 4.2 – Paralelismo Físico.

Paralelismo Físico: Neste caso, tem-se um elemento de processamento para cada processo executado, e todos eles podem estar ativos simultaneamente, como pode ser visto na figura 4.2.

Se existem *n* elementos de processamento, e *m* processos sendo executados concorrentemente, onde $n<m$, forma-se uma situação de paralelismo misto, em que encontram-se, ao mesmo tempo, paralelismo físico e lógico.

O paralelismo físico ainda pode ser dividido em três tipos: espacial, temporal (pipeline) ou combinado.

O paralelismo físico espacial relaciona-se à execução simultânea dos processos, podendo ser síncrono - quando os elementos de processamento são ligados a uma única unidade de controle e, portanto, executam a mesma instrução sincronamente, ou assíncrona, quando os elementos de processamento são independentes entre si. Já o paralelismo temporal, ou pipeline, implica a execução de eventos sobrepostos no tempo. Uma determinada tarefa é subdividida em uma seqüência de subtarefas, sendo cada uma delas executada por um estágio especializado de hardware e software que opera paralelamente com os outros estágios do pipeline, enquanto que o paralelismo combinado equivale a vários estágios pipeline sendo executados em paralelo.

Baseando-se nas definições apresentadas anteriormente, é possível definir três tipos de estilo de programação dentro da computação: a programação seqüencial, que se caracteriza pela execução de várias tarefas, uma após a outra; a programação concorrente, caracterizando-se pela iniciação de várias tarefas, sem que as anteriores tenham necessariamente terminado (sistemas multi ou uniprocessadores); e a

programação paralela caracterizada pela iniciação e execução das tarefas em paralelo (sistemas multiprocessadores).

4.3 Concorrência e paralelismo

O termo paralelismo, na literatura sobre informática, é normalmente utilizado para designar paralelismo físico, ou seja, tem-se múltiplos elementos de processamento para executar os processos, sendo que todos eles podem estar ativos simultaneamente. É importante então, ressaltar a diferença entre paralelismo e concorrência.

A concorrência implica mais de um processo iniciado e não ainda terminado. Não existe uma relação com o número de elementos de processamento utilizados. Obter paralelismo na execução desses processos só é possível quando existe mais de um elemento de processamento, de modo que os processos possam estar em execução em um determinado instante.

Quando existe apenas um elemento de processamento e vários processos estão sendo executados, de maneira concorrente, existe um pseudo-paralelismo. O usuário tem a falsa impressão de que os processos estão sendo executados ao mesmo tempo, mas o que realmente acontece é o compartilhamento do elemento de processamento entre os processos em execução. Isto é, em um determinado instante de tempo, apenas um processo está em execução, enquanto os demais estão aguardando a liberação do processador (MOLINA, 1998).

4.4 Nível de paralelismo ou granularidade

O nível de paralelismo pode ser definido como o tamanho das unidades de trabalho submetidas aos processadores, ou ainda como a quantidade de trabalho realizado entre interações do processador (KIRNER, 1991; ALMASI, 1994).

Diversas definições de Granularidade podem ser encontradas na literatura, sendo divididas em três níves: fina, média e grossa, ou ainda: alta, média e baixa, respectivamente.

Granulação fina relaciona paralelismo em nível de instruções ou operações e implica um grande número de processadores pequenos e simples com comunicação entre eles, muito rápida e confiável. Granulação grossa relaciona o paralelismo em nível de processos e programas, e geralmente aplica-se a plataformas com pouco nível de interação. A granulação média situa-se em um patamar entre as duas anteriores, implicando procedimentos que são executados em paralelo.

4.5 *Speedup* e eficiência

Aumentar o desempenho do processo é uma das metas da computação paralela. Dois parâmetros são utilizados para medir as vantagens na utilização da computação paralela: *speedup* e eficiência (KIRNER, 1991; QUINN, 1987).

Speedup é a medida que tem por objetivo determinar a relação existente entre o tempo dispensado para executar um algoritmo em um único processador (T1) e o tempo gasto para executá-lo em ‘p’ processadores (Tp).

$$speedup ? \frac{T1}{Tp}$$

Quando o speedup se iguala a ‘p’ indica que o caso ótimo foi atingido, significando que o aumento da capacidade de processamento é diretamente proporcional ao número de processadores, definindo o conceito de sistemas escalares. Casos onde o speedup é maior que o ótimo são possíveis, sendo conhecidos por anomalia do *speedup* ou *speedup* superlinear.

A comunicação entre processos, granulosidades inadequadas e partes não paralelizáveis de programas, são fatores que afetam diretamente o resultado do *speedup*.

A eficiência, por sua vez, relaciona speedup e número de processadores, ou seja, identifica a utilização do processador. No caso ótimo (100% de utilização) a eficiência deve equivaler a 1 e o *speedup* a ‘p’; já em casos reais a eficiência é menor que 1 e o *speedup* menor que ‘p’.

$$eficiência \ ? \ \frac{speedup}{p}$$

Nos sistemas paralelos surge o efeito de Amdahl, causado pelas sobrecargas nos programas paralelos, devido a fatores como sincronismo, comunicação e ativação de processos. Essa sobrecarga tende a diminuir com o aumento das tarefas (as de maior complexidade).

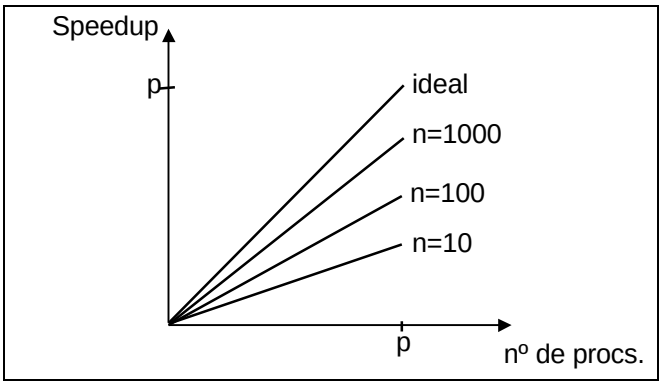

Figura 4.3 - Relação entre *Speedup* e número de processos

4.6 Arquiteruras paralelas

Quando nesse âmbito é referenciado o termo arquitetura, podem-se descrever as arquiteturas computacionais como: uma arquitetura sequencial, uma arquitetura com mecanismos de paralelismo de baixo nível, como *pipeline* de intruções, que implica a execução de eventos sobrepostos no tempo, ou ainda arquiteturas paralelas, as quais conforme DUNCAN (1990), são definidas da seguinte forma: "Uma arquitetura paralela fornece uma estrutura explícita e de alto nível para o desenvolvimento de soluções utilizando o processamento paralelo, através da existência de múltiplos processadores, esses simples ou complexos, que cooperam para resolver problemas através de execução concorrente".

Existem muitas maneiras de se organizar computadores paralelos. Para que se possa visualizar melhor todo o conjunto de possíveis opções de arquiteturas paralelas, é interessante classificá-las. Segundo BEM-DYKE (1993), uma classificação ideal deve ser:

Hierárquica: iniciando em um nível mais abstrato, a classificação deve ser refinada em subníveis à medida que se diferencie de maneira mais detalhada cada arquitetura;

Universal: um computador único, deve ter uma classificação única;

Extensível: futuras máquinas que surjam, devem ser incluídas sem que sejam necessárias modificações na classificação;

Concisa: os nomes que representam cada uma das classes devem ser pequenos para que a classificação seja de uso prático;

Abrangente: a classificação deve incluir todos os tipos de arquiteturas existentes.

4.6.1 Classificação de Flynn

Segundo FLYNN (1972), o processo computacional deve ser visto como uma relação entre fluxos de instruções e fluxos de dados. Um fluxo de instruções equivale a uma seqüência de instruções (em um processador) executadas sobre um fluxo de dados aos quais essas instruções estão relacionadas (ALMASI, 1994 DUNCAN, 1990 FLYNN, 1972).

Baseando-se nas possíveis unicidade e multiplicidade de fluxos de dados e instruções, dividem-se as arquiteturas de computadores em quatro classes:

SISD - *Single Instruction Stream/Single Data Stream* (Fluxo único de instruções/Fluxo único de dados): corresponde ao tradicional modelo von Neumann. Um processador executa seqüencialmente um conjunto de instruções sobre um conjunto de dados.

SIMD - *Single Instruction Stream/Multiple Data Stream* (Fluxo único de instruções/Fluxo múltiplo de dados). Envolve múltiplos processadores (escravos) sob o

controle de uma única unidade de controle (mestre), executando simultaneamente a mesma instrução em diversos conjuntos de dados.

MISD - *Multiple Instruction Stream/Single Data Stream* (Fluxo múltiplo de instruções/Fluxo único de dados). Envolve múltiplos processadores executando diferentes instruções em um único conjunto de dados. Geralmente, nenhuma arquitetura é classificada como MISD, isto é, não existem representantes desta categoria.

MIMD - *Multiple Instruction Stream/Multiple Data Stream* (Fluxo múltiplo de instruções/Fluxo múltiplo de dados). Envolve múltiplos processadores executando diferentes instruções em diferentes conjuntos de dados, de maneira independente.

A classificação de Flynn não é abrangente o suficiente para incluir alguns computadores modernos (por exemplo, processadores vetoriais e máquinas de fluxo de dados), falhando também, no que concerne a extensibilidade da classificação. Outro inconveniente desta classificação é a falta de hierarquia. A classificação MIMD, por exemplo, engloba quase todas as arquiteturas paralelas sem apresentar subníveis.

4.6.2 Classificação de Duncan

DUNCAN (1990), em sua classificação, exclui arquiteturas que apresentem apenas mecanismos de paralelismo de baixo nível, os quais já se tornaram lugar comum nos computadores modernos.

A classificação de Duncan é apresentada na Figura 4.4, e a seguir a classificação Assíncrona/MIMD será discutida mais a fundo, tendo em vista as diretrizes deste trabalho.

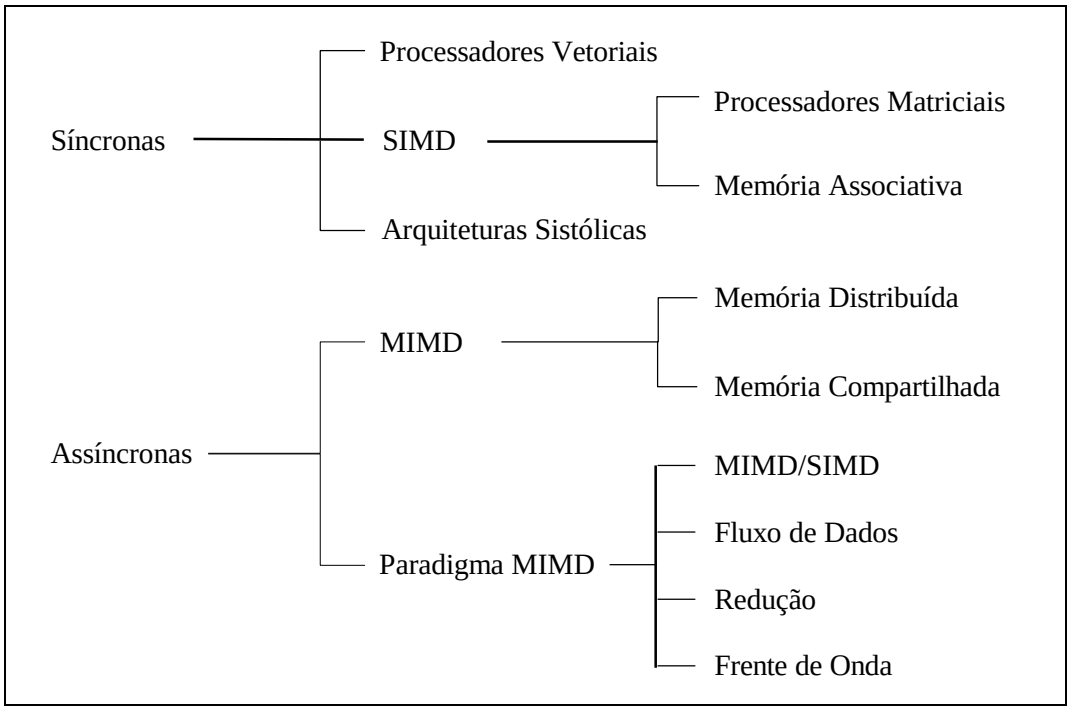

Figura 4.4 - Classificação de Duncan

Arquiteturas Síncronas: são aquelas que coordenam suas operações concorrentes sincronamente em todos os processadores, através de relógios globais, unidades de controle únicas ou controladores de unidades vetoriais (DUNCAN, 1990). Tais arquiteturas apresentam pouca flexibilidade para a expressão de algoritmos paralelos (BLECH, 1994).

Arquiteturas Assíncronas: caracterizam-se pelo controle descentralizado de *hardware*, de maneira que os processadores sejam independentes entre si. Essa classe é formada basicamente pelas arquiteturas MIMD, sejam convencionais ou não (DUNCAN, 1990).

MIMD: que relacionam arquiteturas compostas por vários processadores independentes, onde se executam diferentes fluxos de instruções em dados locais desses processadores.

Este tipo de organização pressupõe algoritmos de granulação mais grossa (implicando plataformas com poucos processadores complexos), com pouca comunicação entre processos (em virtude da sobrecarga de comunicação ser maior), enquanto arquiteturas síncronas utilizam algoritmos de granulação mais fina. Além disso, as arquiteturas MIMD oferecem grande flexibilidade para a construção de

algoritmos paralelos (BLECH, 1994). Apesar de independentes, os processos em execução nos diversos processadores devem cooperar entre si, tornando necessárias a comunicação e o sincronismo entre esses processos. A implementação dos métodos de comunicação e sincronismo depende da organização de memória, que pode ser centralizada ou distribuída (Figura 4.5).

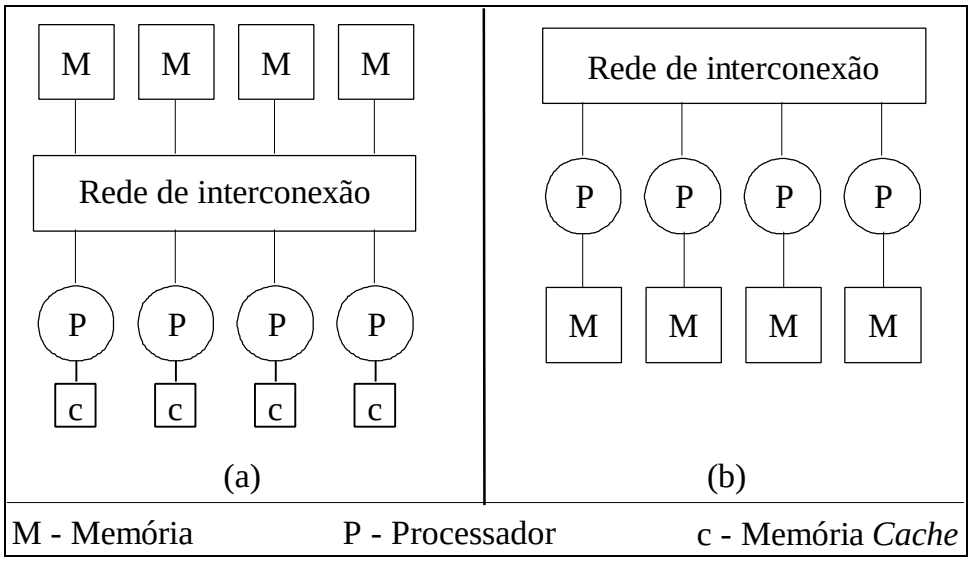

Figura 4.5 - Arquiteturas MIMD (a) Memória Centralizada (b) Memória Distribuída

Arquiteturas de memória centralizada (ou multiprocessadores) caracterizam-se pela existência de uma memória global e única, a qual é utilizada por todos os processadores (fortemente acoplados), de maneira que através do compartilhamento de posições desta memória ocorre a comunicação entre os processos. Para evitar que a memória se torne um gargalo, as arquiteturas de memória centralizada devem implementar mecanismos de *cache*, além de garantirem a coerência dos dados (através de mecanismos de *hardware* e *software*).

Apesar das complicações geradas pela necessidade de gerenciamento de memória *cache*, as arquiteturas de memória centralizada apresentam grande flexibilidade de programação, como é descrito adiante, de maneira que essas arquiteturas tornem-se uma opção interessante para o programador paralelo.

Em arquiteturas de memória distribuída, cada processador possui sua própria memória local, sendo então fracamente acoplado. Em virtude de não haver compartilhamento de memória, os processos comunicam-se via troca de mensagens, que

se trata da transferência explícita de dados entre os processadores. Este tipo de organização é também conhecida como multicomputador.

A distinção entre as duas organizações de memória citadas deve ser feita com dois referenciais em mente: *hardware* e *software* (ALMASI, 1994 TANENBAUM, 1995 BLECH, 1994).

Do ponto de vista de programação, a diferença básica reside nas primitivas de comunicação entre processos. A comunicação em memória centralizada é baseada no compartilhamento de posições de memória, e para que os dados se mantenham consistentes, é necessário um método de controle de acesso a essas variáveis compartilhadas, como por exemplo: semáforos, monitores, etc. Esses mecanismos de programação já são bem conhecidos pelo programador em geral.

No caso de memória distribuída, a comunicação é feita através de troca de mensagens, o que gera algumas complicações, tais como: o controle de fluxo, controle sobre mensagens perdidas, *bufferização* e bloqueio de rotinas. Então, do ponto de vista do programador, a melhor escolha é a memória compartilhada. É interessante ressaltar que se pode usar o paradigma de troca de mensagens em memória compartilhada, o que aumenta o leque de opções para o programador.

Em relação ao *hardware*, essa situação se inverte. Arquiteturas de memória distribuída são mais fáceis de se construir e são naturalmente ampliáveis (MCBRYAN, 1994) (BLECH, 1994). Portanto, do ponto de vista do projetista, a melhor escolha é uma arquitetura de memória distribuída.

O caso ideal é a existência de uma arquitetura que seja fácil de projetar e programar. Com esse objetivo em mente foi criada a idéia das arquiteturas de memória centralizada distribuída (ou memória compartilhada virtual). Nesse caso, constrói-se uma plataforma de memória distribuída e, via mecanismos implementados em *hardware* e *software*, emula-se um ambiente de memória centralizada (MCBRYAN, 1994 TANENBAUM, 1995 BLECH, 1994).

4.6 Programação concorrente

Dentro da computação estão definidos três estilos de programação:

Programação seqüencial: caracteriza-se pela execução de várias tarefas, uma após a outra;

Programação concorrente: caracteriza-se pela iniciação de várias tarefas, sem que as anteriores tenham necessariamente terminado (sistemas multi ou uniprocessadores);

Programação paralela: caracteriza-se pela iniciação e execução das tarefas em paralelo (sistemas multiprocessadores).

Um programa seqüencial é composto por um conjunto de instruções que são executadas seqüencialmente, sendo que a execução dessas instruções é denominada de processo. Um programa concorrente especifica dois ou mais programas seqüenciais que podem ser executados concorrentemente como processos paralelos (ANDREWS, 1983).

A programação concorrente existe para fornecer ferramentas para a construção de programas paralelos, de maneira que se consiga melhor desempenho e melhor utilização do hardware paralelo disponível. A computação paralela apresenta muitas vantagens em relação à computação seqüencial.

Um programa seqüencial é constituído basicamente por um conjunto de construções já bem dominadas pelo programador em geral, como por exemplo, atribuições, comandos de decisão (*if... then... else*), laços (*for... do*), entre outras. Um programa concorrente, além dessas primitivas básicas, necessita de novas construções que lhe permitam tratar aspectos decorrentes da execução paralela dos vários processos.

Segundo ALMASI (1994), para a execução de programas paralelos, deve haver meios de definir um conjunto de tarefas a serem executadas paralelamente, ativar e encerrar a execução dessas tarefas, bem como coordenar e especificar a interação entre elas.

A fase de definição da organização das tarefas paralelas é de extrema importância, pois o ganho de desempenho adquirido com a paralelização depende fortemente da melhor configuração das tarefas a serem executadas concorrentemente.

Definido o algoritmo, é necessário um conjunto de ferramentas para que o programador possa representar a concorrência, definindo quais partes do código serão executadas seqüencialmente e quais serão paralelas.

Além disso, processos cooperando para a resolução de determinado problema devem comunicar-se e sincronizar-se, a fim de que haja interação entre eles. A maneira como se implementa a comunicação entre processos depende da arquitetura onde se executa a aplicação: em caso de memória centralizada, utilizam-se variáveis compartilhadas; em caso de memória distribuída, é utilizada troca de mensagens.

4.7 Comunicação e sincronismo

A comunicação é necessária para que processos interagindo na resolução de determinada aplicação troquem informações. E quando há comunicação, devem existir operações de sincronização, para fornecer controle de acesso e controle de seqüência.

Primitivas de sincronização (e conseqüentemente de comunicação) devem estar presentes para que os processos possam se comunicar uns com os outros e, dessa maneira, interferirem uns nos outros. A sincronização e a comunicação são implementadas com o uso de variáveis compartilhadas ou através da passagem de mensagens.

Observa-se em ALMASI (1994) que esses mecanismos de comunicação foram desenvolvidos para atuar em um dos dois grandes grupos: nas arquiteturas de memória compartilhada ou nas de memória distribuída.

4.7.1 Comunicação e sincronismo em memória compartilhada

O controle de acesso e a seqüência em memória compartilhada são implementados através da utilização de variáveis compartilhadas entre os diversos processos concorrentes.

Para a implementação de exclusão mútua e controle de seqüência, que se utilizam de variáveis compartilhadas, vários métodos podem ser empregados, dos quais destacam-se: espera ocupada, semáforos e monitores (ALMASI, 1994 ANDREWS, 1983 TANENBAUM, 1992 QUINN, 1987 KIRNER, 1991).

Espera Ocupada: Utilizando-se uma variável compartilhada cujo valor pode ser manipulado (modificado), através de uma primitiva indivisível, cria-se um modo simples para a sincronização de processos concorrentes. Um processo, ao querer entrar em uma região crítica, deve executar esta primitiva, continuamente, até conseguir permissão de entrada. Uma das desvantagens da espera ocupada é o gasto supérfluo de tempo de CPU.

Semáforos: Este mecanismo de sincronização genérico é composto por duas operações atômicas: up e down. Essas operações atuam sobre uma variável compartilhada de modo a permitir o controle de acesso e a exclusão mútua. Em adição, oferecem um meio de sincronização de nível mais elevado do que a espera ocupada, evitando o desperdício de CPU, no entanto, apresentam como desvantagem a desestruturação do código.

Monitores: Estes oferecem uma maneira estruturada de implementar a exclusão mútua. Um monitor consiste em algumas variáveis permanentes que armazenam o estado do recurso compartilhado e de alguns procedimentos que implementam operações sobre esses recursos. Essas variáveis podem ser acessadas somente pelos procedimentos internos a cada monitor, e a execução desses procedimentos é feita de maneira mutuamente exclusiva.

4.7.2 Comunicação e sincronismo em memória distribuída

Tanto a comunicação como o sincronismo feito em memória distribuída devem ser implementados através de troca de mensagens entre processos (ALMASI, 1994 ANDREWS, 1983 TANENBAUM, 1992 QUINN, 1987 KIRNER, 1991).

Quando esse tipo de comunicação e sincronização é utilizado, processos enviam e recebem mensagens. A comunicação é efetuada porque um processo, após receber uma mensagem, obtém valores de algum processo emissor. Dessa mesma forma, a sincronização é efetuada porque uma mensagem pode ser recebida somente depois de ter sido enviada, o que impede a ocorrência dos eventos em ordem inversa.

As primitivas send/receive utilizadas na comunicação podem ser bloqueantes ou não bloqueantes.

O processo de comunicação pode ser organizado segundo abstrações distintas:

Comunicação Ponto a Ponto: caracteriza-se pelo uso de uma operação send/receive bloqueante, de modo que os processos se sincronizem. Neste processo a comunicação é unidirecional.

Rendezvouz: neste mecanismo dois processos concorrentes executam as primitivas send/receive duas vezes, apresentando comunicação síncrona e bidirecional (AXF, 1990).

RPC (Remote Procedure Call): caracteriza-se pela execução de um procedimento não local a um determinado processo, utilizando uma sintaxe semelhante à chamada de procedimentos locais, sendo que o processo requisitante de serviço é bloqueado até que se obtenham os resultados desejados. O procedimento é iniciado quando é recebida uma requisição de execução. Normalmente, o procedimento remoto retorna valores, tornando a comunicação bidirecional.

CAPÍTULO 5

Programação por Troca de Mensagens

Neste capítulo é descrito o paradigma de programação por troca de mensagens, sendo apresentadas as principais abordagens existentes, tais como: conceitos de *Message Passing*, ambiente de programação, rotinas básica e suporte MPI.

5.1 Introdução

No âmbito da computação paralela, destacam-se as plataformas paralelas de memória distribuída, que podem ser computadores paralelos ou máquinas paralelas virtuais (BLECH, 1994 e ZALUSKA, 1991).

A cooperação entre processadores que trabalham em conjunto na execução de uma determinada tarefa é essencial. Quando possuímos memória distribuída, onde cada processador possui seu próprio dispositivo de memória local, devem ser definidas as primitivas que possibilitem que os processos se comuniquem e se sincronizem (BEN-ARI 1985 e QUINN, 1987).

Para isso, tem-se um conjunto dessas primitivas, as quais permitem que os processadores troquem mensagens entre si, requisitando dados explicitamente de outros processadores. Estas primitivas caracterizam o paradigma da troca de mensagens.

Para a autora ANGELA QUEALY (1994), troca de mensagens é um método para a comunicação de processos quando não há compartilhamento de memória. Ela é necessária porque:

- Memórias são locais aos processadores;

- Não há compartilhamento de variáveis;
- É a única maneira de se conseguirem dados de outras memórias.

O paradigma da troca de mensagens apresenta-se apenas como uma das alternativas viáveis e torna-se cada vez mais popular, podendo-se descrever alguns fatores que justifiquem a sua aceitação (DONGARRA, 1995):

a) Generalidade é a grande variedade de plataformas em que podem ser executados os processos, como em arquiteturas paralelas de memória distribuída e sistemas distribuídos.

b) Adequação a ambientes distribuídos ampliáveis: tem capacidade de aumentar seu poder de processamento à medida que são ampliados os componentes de um sistema.

Tal paradigma não deve se tornar obsoleto tão logo, pois sempre se fará necessária de alguma maneira a troca de mensagens entre os processos.

5.2 Ambiente de programação

Um programa que utiliza a troca de mensagens para comunicação entre processos de um sistema pode ser definido como um conjunto de programas seqüenciais, distribuído em vários processadores que se comunicam através de um conjunto de instruções, as quais formam o ambiente de troca de mensagens (DONGARRA, 1995).

Em um ambiente de programação via troca de mensagens tem-se uma linguagem seqüencial, a qual será utilizada para escrever os programas seqüenciais que serão executados nos processadores e uma biblioteca de troca de mensagens, a qual fornece as ferramentas necessárias para a ativação e cooperação entre os processos paralelos.

Para a programação desses sistemas abordam-se dois paradigmas: o paradigma MPMD (Multiple Program – Multiple Data) e o paradigma SPMD (Single Program – Multiple Data). No primeiro, MPMD, temos os códigos fontes distintos em cada um dos processadores, e no SPMD, o programa nos diversos processadores tem o mesmo

código fonte e cada processador deve executar de maneira independente uma determinada parte do programa (BAL, 1989).

Os ambientes de passagem de mensagem foram desenvolvidos inicialmente para máquinas com processamento maciçamente paralelo (Massively Parallel Processing - MPP), onde cada fabricante desenvolveu seu próprio ambiente, sem se preocupar com a portabilidade do software gerado.

Para resolver o problema de portabilidade, grupos de pesquisa desenvolveram ambientes de passagem de mensagens independentes da máquina ou plataforma a ser utilizada. Esses ambientes foram chamados de plataformas de portabilidade, que são ambientes de programação portáteis implementados em várias arquiteturas paralelas e sistemas distribuídos.

Para que as aplicações pudessem ganhar a portabilidade e serem executadas em todos os equipamentos para os quais o ambiente foi desenvolvido independente da máquina que está sendo utilizada, definiu-se um conjunto de funções para implementá-las em várias plataformas de hardware. Estes ambientes podem ser utilizados em sistemas heterogêneos, onde duas ou mais máquinas distintas cooperam entre si para resolver um problema. Com isso, ganhou-se popularidade e aceitação, já que disponibilizam de um ambiente paralelo com custo relativamente baixo quando comparado às máquinas paralelas.

Dentre as plataformas de portabilidade, podem-se citar dois representantes que merecem destaque no cenário computacional atual: o PVM (Parallel Virtual Machine) e o MPI (Message Passing Interface) (MCBRYAN, 1994 DONGARRA, 1995 KITAJINA, 1995).

O PVM destaca-se por ser considerado, por alguns autores, um padrão de fato para plataformas de portabilidade, enquanto o MPI é uma tentativa de padronização, realizada por diversas organizações mundiais. Por ser utilizada nesse trabalho, a plataforma de portabilidade MPI será descrita com maiores detalhes na seção 5.4.

5.3 Rotinas para troca de mensagens

Para uma mensagem passar de um processo para o outro, faz-se necessário um conjunto de rotinas capaz de operar e controlar essa troca de informações (QUEALY 1994). Essa descrição será genérica, pois a sintaxe varia de acordo com o ambiente em que se está trabalhando.

As rotinas se fazem necessárias sempre que uma informação tenha que ser transferida entre processos e esses não possuírem memória compartilhada, devendo ser transportada da memória local de um processo para a memória local do outro, via troca de mensagens. Por exemplo, se um processo que está sendo executado no processador Pn necessita de uma informação que está no processo executado pelo processador P0, então essa informação será transferida da memória local de P0 para a memória local de Pn. Esse modelo é mostrado na figura 5.1 (QUEALY 1994):

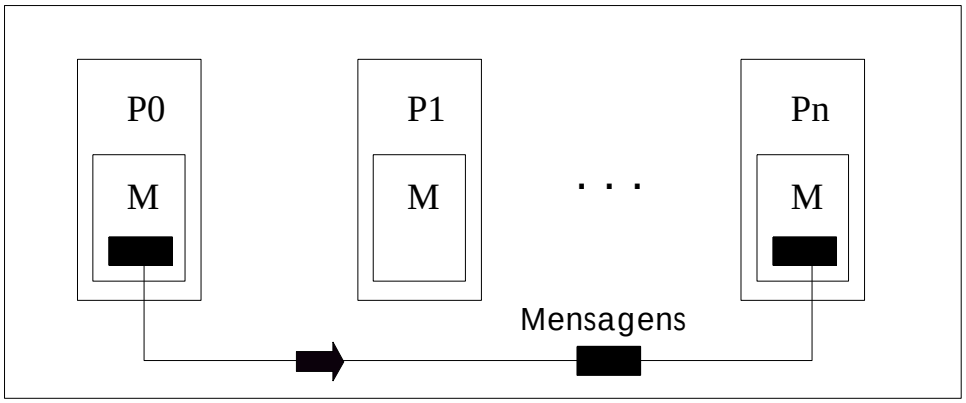

Figura 5.1 – Transferência de uma Mensagem.

Para uma troca de mensagens entre processos, há um conjunto de informações que devem ser consideradas [Macdonald, 1996]. Essas informações são:

- qual processo está enviando a mensagem;
- quais os dados que formam esta mensagem;
- qual o tipo, ou tipos dos dados;
- qual o tamanho da mensagem;
- qual processo vai receber a mensagem;
- aonde os dados serão armazenados no processo receptor;

- qual a quantidade de dados suportada pelo receptor.

Uma informação que também deve fazer parte da mensagem é um identificador, a partir do qual a mensagem possa ser selecionada pelo processo receptor.

Uma mensagem pode ser transmitida ou recebida de diferentes maneiras, conforme sua forma de comunicação, síncrona ou não, e também em relação ao número de processos envolvidos na operação. Na próxima seção serão discutidas as rotinas de comunicação ponto-a-ponto e coletivas já inseridas no contexto MPI.

5.4 *Message Passing Interface* - MPI

Pelo grande número de plataformas existentes que ocasionam restrições em relação à real portabilidade dos programas e o mau aproveitamento de características de algumas arquiteturas paralelas, surge a necessidade de definir um padrão para ambientes de troca de mensagens, tendo assim, em 1992, o início dos estudos sobre *Message Passing Interface*, sendo criado o comitê MPI, o qual adotou procedimentos para criação de um MPI Fórum.

O MPI define um conjunto básico de rotinas que oferecem os seguintes serviços: suporte para grupos de processos, suporte para contextos de comunicação, suporte para topologia de processos, comunicação ponto-a-ponto e comunicação coletiva (DONGARRA, 1995), (MPI, 1995) (SOUZA, 1996).

5.4.1 Suporte para grupos de processos

O MPI relaciona os processos em grupos, e esses processos são identificados pela sua classificação dentro desse grupo. Todo processo tem uma identificação única atribuída pelo sistema quando ele é inicializado. Essa identificação é contínua e representada por um número inteiro, começando de zero até N-1, onde N é o número de processos.

O Rank é um identificador único do processo, utilizado para identificá-lo no envio (send) ou recebimento (receive) de uma mensagem.

5.4.2 Suporte para contextos de comunicação

O MPI utiliza a combinação de grupo e contexto para garantir segurança na comunicação e evitar problemas no envio de mensagens entre os processos, evitando que grupos de processos não relacionados não recebam mensagens de grupos de outros contextos.

Outro conceito introduzido pelo MPI é o *communicator*. O *communicator* é um objeto local que representa o domínio (contexto) de um conjunto de processos que podem ser contactados, criando contextos distintos para cada grupo de processos.

5.4.3 Suporte para topologia de processos

O MPI fornece primitivas que permitem ao programador definir a estrutura topológica com a qual os processos de um determinado grupo se relacionarão. Como exemplo de uma topologia, pode citar-se uma malha, onde cada ponto de intersecção na malha corresponde a um processo.

5.4.4 Comunicação ponto-a-ponto

Dentre as rotinas básicas do MPI estão as rotinas de comunicação ponto-a-ponto que executam a transferência de dados entre dois processos.

Existem quatro (4) modos de comunicação ponto-a-ponto:

*Synchronous (*Síncrono*):* é a comunicação na qual o processo que envia a mensagem não retorna à execução normal enquanto não haja um sinal do recebimento da mensagem pelo destinatário.

*Ready (*Imediata*):* neste tipo de comunicação, sabe-se, a *priori*, que o recebimento *receive* de uma mensagem já foi efetuado pelo processo destino, podendo-se enviá-la *send* sem necessidade de confirmação ou sincronização, melhorando o desempenho na transmissão.

*Buffered (*Buferizada*):* no modo *buffered* a operação de envio *send* utiliza uma quantidade de espaço específica para o buffer, definida pelo usuário *application buffer*. Isto se tornará necessário quando a quantidade de mensagem a ser enviada ultrapassar o tamanho padrão do buffer de 4Kbytes para um processo.

*Standard (*Padrão*):* é um modo padrão para o envio *send* de mensagens, buscando um meio termo entre eficiência e segurança, usando comunicação *blocking*.

Além dos tipos de comunicação citados acima, temos as chamadas de rotinas de comunicação bloqueantes, quando a finalização da execução da rotina é dependente de determinados eventos, ou seja, espera por determinada ação antes de liberar a continuação do processamento, e as não-bloqueantes, quando a finalização da execução da rotina não depende de determinados eventos, ou seja, o processo continua sendo executado normalmente sem haver espera.

5.4.5 Comunicação coletiva

Comunicação coletiva é a comunicação padrão que invoca todos os processos em um grupo, onde podem comunicar-se entre si. Normalmente, a comunicação coletiva envolve mais de dois processos. As rotinas de comunicação coletiva são voltadas para a comunicação e coordenação de grupos de processos.

Algumas operações coletivas disponíveis no MPI são:

Barrier (Rotina de sincronização): a função desta rotina é retornar depois de ter sido chamada por todos os processos relacionados com o *communicator*, o qual foi

passado como um argumento para o barrier. Uma chamada *barrier* (barreira) não tem efeito na memória local dos processos.

*Broadcast (*Difusão*):* é a comunicação coletiva em que um único processo envia *send* os mesmos dados para todos os processos com o mesmo *communicator*.

*Reduction (*Redução*):* é a comunicação coletiva onde cada processo no *communicator* contém um operador, e todos eles são combinados usando um operador binário que será aplicado sucessivamente.

*Gather (*Coleta*):* a estrutura dos dados distribuídos é coletada por um único processo.

*Scatter (*Espalhamento*):* a estrutura dos dados que está armazenado em um único processo é distribuído a todos os processos.

5.4.6 Considerações finais

Os ambientes de passagem de mensagem foram desenvolvidos com o objetivo de utilizar o potencial dos sistemas distribuídos para o desenvolvimento de aplicações paralelas e permitir a união de plataformas heterogêneas e a portabilidade das aplicações paralelas desenvolvidas.

Visando tornar essas aplicações independentes da arquitetura utilizada, formulou-se a idéia de plataformas de portabilidade. Vários ambientes de passagem de mensagem, como o MPI, foram desenvolvidos seguindo essa idéia e os demais objetivos da computação paralela distribuída.

Esses ambientes, caracterizados pela busca de simplicidade e eficiência, permitem que aplicações paralelas sejam desenvolvidas, utilizando hardware relativamente barato (se comparado às máquinas paralelas) e com software familiar à equipe de desenvolvimento (SOUZA, 1996).

5.5 Programa exemplo

O programa-exemplo descrito abaixo, e que implementa a transferência de uma mensagem entre dois processos, serve para que se tenha uma idéia básica da programação MPI, que possui suporte de programação para as linguagens C e Fortran. Este exemplo está escrito em C.

O programa-exemplo executa de maneira SPMD em dois processadores. São gerados dois processos identificados por ranks 0 e 1, sendo que o processo de rank = 0 envia uma mensagem para o processo de rank = 1, que recebe esta mensagem.

```
#include <mpi.h>  /* Biblioteca MPI */
main(argc, argv)
int argc;
char *argv[];
{
   char msg[20];              /* Mensagem a ser enviada */
   int myrank;                /* Rank de um processo */
   int tag = 99;              /* Identificador da mensagem (tag) */
   MPI_Status status;         /* Variável status, utilizada pela rotina
receive( ) */
   MPI_Init(&argc, &argv);    /* Inicia o MPI */
   MPI_Comm_rank(MPI_COMM_WORLD,  &myrank);        /*
Define o rank do processo */

   /* O processo com rank 0 envia a mensagem, o de rank 1 a recebe
*/
   if (myrank == 0) {
      strcpy( msg, "Hello world");
      MPI_Send(msg,  strlen(msg)  +  1,  MPI_CHAR,  1,
tag,MPI_COMM_WORLD);
     }
      else {
```

```
                    MPI_Recv( msg, 20, MPI_CHAR, 0, tag,
MPI_COMM_WORLD, &status);
                    printf(`` %s \n", msg);
                    }
            MPI_Finalize( );            /* Finaliza o MPI */
          }
```

Figura 5.2 – Programa exemplo MPI escrito em C.

Este é um exemplo extremamente simples, porém, geralmente, só em casos isolados necessita-se da utilização das rotinas mais avançadas do MPI.

CAPÍTULO 6

O Método Proposto

Neste capítulo é descrito o método proposto para paralelamente, otimizar combinatoriais, tendo como base o algoritmo *Simulated Annealing* (SA). Onde, inicialmente, são apresentadas as definições existentes do método, utilizando múltiplas faixas de temperatura e, por fim, o modelo proposto, apresentando o método paralelo para esta abordagem.

6.1 Introdução

O considerado aumento do interesse por abordagens baseadas nas técnicas de otimização de busca local para tratamento de problemas considerados NP-Completos, que podem utilizar como base o algoritmo *Simulated Annealing*, demonstra cada vez mais a importância de se encontrar um **algoritmo aproximado** para resolver esses problemas que podem mostrar soluções viáveis para muitos dos problemas encontrados nos sistemas de informações das organizações.

Os algoritmos como o SA, desenvolvidos para resolver problemas de otimização, cumprem seu papel satisfatoriamente, porém, muitas vezes o tempo de processamento impõe restrições à utilização do algoritmo em determinados problemas reais. Diante disso, surge uma grande possibilidade de avanço com a implementação sendo executada em paralelo, necessitando assim, suporte por parte da implementação, devido ao fato de a seqüência de instruções ser um conceito básico do algoritmo e não permitir que este seja automaticamente paralelizável.

O importante Problema do Caixeiro Viajante (PCV), aliado ao fato de ser um representante de uma classe muito vasta de outros problemas de otimização combinatorial, contém dois ingredientes necessários que atraem muitos pesquisadores, ou seja, tem simplicidade na formulação e é de difícil resolução. Além disso, tem-se mostrado eficaz no desenvolvimento de testes e demonstrações de novas propostas de aplicabilidade.

6.2 Descrição formal do problema

A modelagem do PCV pode ser feita pela análise combinatória ou pela teoria de grafos, na qual a solução consiste em encontrar um circuito hamiltoniano, isto é, um circuito que visite todas as cidades, somente uma vez, obtendo o menor custo/caminho. Definimos custo (*i*,*j*) como o valor do peso da aresta entre os vértices *i* e *j*.

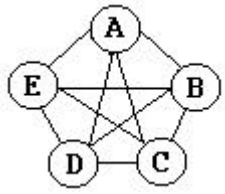

Figura 6.1 – Circuito Hamiltoniano

Desta forma o PCV pode ser representado por um grafo *G* onde os vértices são as *n* cidades a visitar e uma matriz $D = [d_{ij}]$ de distâncias entre cada cidade (*i*,*j*). A função custo é definida por:

$$f(p) = \sum_{i=1}^{n-1} d_{i,\,i+1} + d_{n,\,1}$$

onde a solução *p* é uma permutação de *n* cidades.

Matematicamente, o PCV pode ser formulado como uma matriz de custo $C = [d_{ij}]$, onde d_{ij} representa o custo de ir da cidade *i* para a cidade *j* e encontrar uma

permutação $p(i_1,i_2,i_3,\ldots,i_n)$ de inteiros, de 1 até que *n* minimize a quantidade obtida pela soma. Onde o custo = $c_{i1,i2} + c_{i2,i3} + \ldots + c_{in,i1}$.

O modelo mostrado será utilizado para implementar o algoritmo *Simulated Annealing* para resolver o PCV.

6.3 Método existente

A resolução de problemas de otimização combinatória tem sido tratada na literatura, basicamente, por dois tipos de métodos: aqueles ditos exatos e os chamados métodos aproximados. Os métodos exatos garantem que a solução ótima para o problema é encontrada, porém a desvantagem é que os algoritmos gerados a partir desses métodos são, em geral, computacionalmente custosos, o que limita suas utilizações a instâncias pequenas. Alguns métodos dessa classe que ainda são usados, como por exemplo, *branch-and-bound* e planos de cortes (APPLEGATE et al., 1998), têm suas fundamentações dentro da programação linear e suas variantes.

Os métodos aproximados, apesar de não garantirem soluções ótimas, possibilitam que instâncias maiores sejam tratadas em menor tempo. Por isso são os que têm atraído mais a atenção dos pesquisadores e, portanto, conseguido os avanços mais expressivos. Portanto, neste trabalho serão considerados apenas os métodos baseados nos estudos e implementações feitas por ARAUJO (2001).

Como o SA é um procedimento iterativo, uma dada temperatura é mantida constante por um número predeterminado de repetições, para então ser reduzida através de sua multiplicação por um valor constante, denominado fator de redução. O número de repetições para uma dada temperatura, que na nomenclatura do SA é para atingir o equilíbrio, pode ser visto, dentro do contexto de uma solução para o problema sendo tratado, como sendo o número de vizinhos que devem ser alcançados. Após esse número de repetições, a temperatura é reduzida e um novo ciclo se inicia, até que a temperatura alcance um valor mínimo, normalmente próximo de zero.

Qualquer algoritmo que utilize como paradigma a abordagem SA tem os valores da temperatura retirados de uma única faixa. O algoritmo inicia com uma

temperatura T_0, geralmente alta, e vai diminuindo a cada iteração, através do fator de redução, até atingir um valor de temperatura mínimo, normalmente próximo de zero.

Faixa única significa que os valores assumidos pela temperatura são retirados de uma faixa compreendida entre os valores inicial e final da temperatura, que vai diminuindo cada vez que o sistema se equilibra. Com isso, é possível encontrar a quantidade de vezes que o sistema alcança o equilíbrio, já que ele está relacionado com os valores inicial e final da temperatura e do fator de redução, que matematicamente, de forma simples, pode ser calculada.

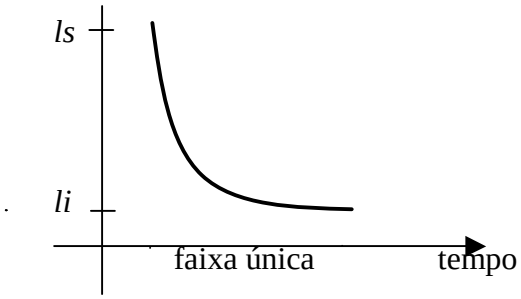

Figura 6.2 – Temperatura retirada de uma única faixa
Modelo básico do SA

No método mostrado por ARAUJO (2001), o valor da temperatura continua constante para um determinado número de repetições, só que agora a temperatura não percorre somente uma faixa de valores entre as temperaturas inicial e final, mas com várias faixas compreendidas entre estes dois valores.

Para tratar com as múltiplas faixas, são definidos dois intervalos. Um superior, que tem os valores *ls* e *lls* representando, respectivamente, o limite superior e inferior do intervalo, e um inferior, com os valores *li* e *lli* representando o limite superior e inferior do intervalo, respectivamente.

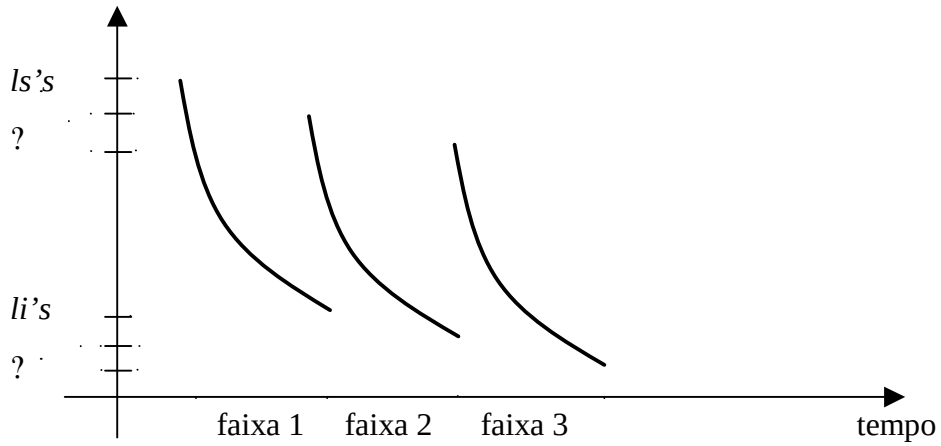

Figura 6.3 - Temperatura retirada de três faixas

No intervalo superior estão os valores iniciais que a temperatura assume em cada uma das faixas. Cada um desses valores que a temperatura assume no intervalo superior é determinado em razão dos valores do limite e do número de faixas que é utilizado. Descrição idêntica tem o intervalo inferior, só que representando os valores mínimos que a temperatura assume em cada faixa. A figura 6.3 ilustra tal situação.

Durante a execução do algoritmo, duas constantes passam a controlar a redução dos valores iniciais e finais da temperatura em cada faixa denominada ? e ?. Os valores das constantes ? e ? serão determinados pelo número de faixas que é utilizado no processo, através das fórmulas 6.3.1 e 6.3.2. Em cada faixa, o valor da temperatura *T* se inicia com *ls* e é reduzido até *li* utilizando a expressão $T = ? * T$, onde ? é o fator redutor. Após concluir a varredura dos pontos ao longo de uma faixa, a temperatura salta, abruptamente, para o limite superior da nova faixa, que é obtido através da expressão $ls = ls * ?$, enquanto que o limite inferior é obtido por $ls = ls * ?$.

$$? \text{ (beta)} = exp\ (\ (\ ln\ (\ lls\ ) - ln\ (\ ls\ )\ )\ /\ nfaixa\ ) \qquad (6.3.1)$$

$$? \text{ (gama)} = exp\ (\ (\ ln\ (\ lli\ ) - ln\ (\ li\ )\ )\ /\ nfaixa\ ) \qquad (6.3.2)$$

Após o teste de aceitação, uma solução melhor pode surgir, tornando-se a solução corrente. Em cada temperatura, um equilíbrio deve ser alcançado por meio de uma pesquisa exaustiva na vizinhança da solução corrente. Como o procedimento é interativo, após cada equilíbrio, a temperatura é reduzida até que alcance T_{Min}, quando então o procedimento se reinicia com mudanças bruscas, nos novos valores da temperatura, que são feitas quando os valores de *ls* e *li* são calculados, já dentro da próxima faixa.

6.4 Método proposto

O método proposto neste trabalho vem incrementar a pesquisa apresentada em Araújo (2001), e não tem como objetivo principal a obtenção de recordes em número de cidades com solução ótima garantida, nem grandes reduções no tempo de processamento. Mesmo sendo um dos propósitos da programação paralela, o tempo de processamento não é o principal alvo do trabalho, e sim, a troca de informações dos melhores resultados entre os processos a cada troca de faixa, possibilitando após *n* iterações um resultado melhor que o do algoritmo seqüencial.

Este trabalho passou por um grande número de fases nas quais foi sendo sucessivamente alterado para satisfazer aquilo que se pretendia. Desde a fase inicial até a implementação prática, muitas questões foram sendo levantadas, um grande número de opções teve de ser avaliada e muitas decisões tomadas.

6.4.1 Descrição do método

O método implementado para paralelizar o algoritmo SA é bastante simples. Cada processo recebe uma cópia do algoritmo e executa-o de forma assíncrona, utilizando os mesmos dados iniciais, porém, independentes e ignorando totalmente a execução dos demais processos, até que a temperatura chegue no limite inferior da faixa *Li's* (figura 6.3). A partir daí cada processo envia a sua solução atual a um processo centralizador, o qual, ao término de todos os processos, seleciona o melhor

resultado e envia novamente para todos os processos que executarão a nova faixa com um novo valor, melhor ou igual ao existente no atual processo, e assim, sucessivamente até o término das faixas, quando a temperatura chegar ao limite inferior *Li* (figura 6.3), quando então o processo centralizador escolhe o melhor resultado.

O tempo gasto na troca de mensagens não inviabiliza este enfoque, pois a troca de mensagens só ocorre a cada troca de faixa, e conforme ARAUJO (2001), o número ideal de faixas para alcançar bons resultados ficaria em torno de sete.

O modelo é representado nas figura 6.4, 6.5, 6.6, 6.7, onde se tem quatro processos executando o algoritmo utilizando três faixas.

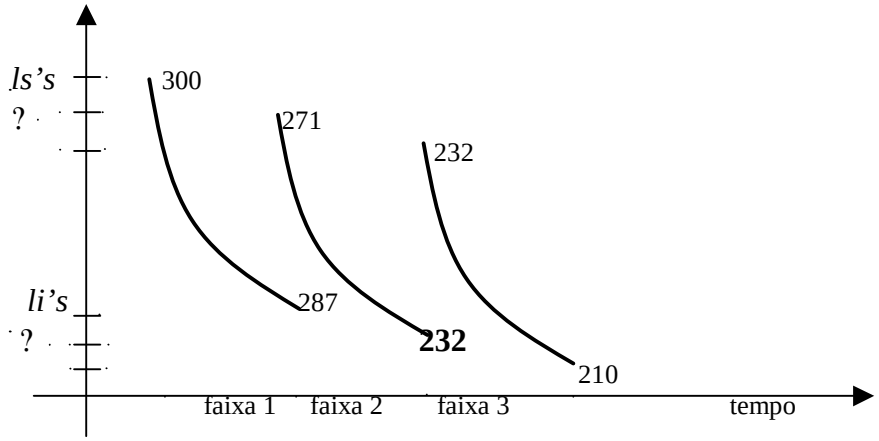

Figura 6.4 - Processo um, com três faixas

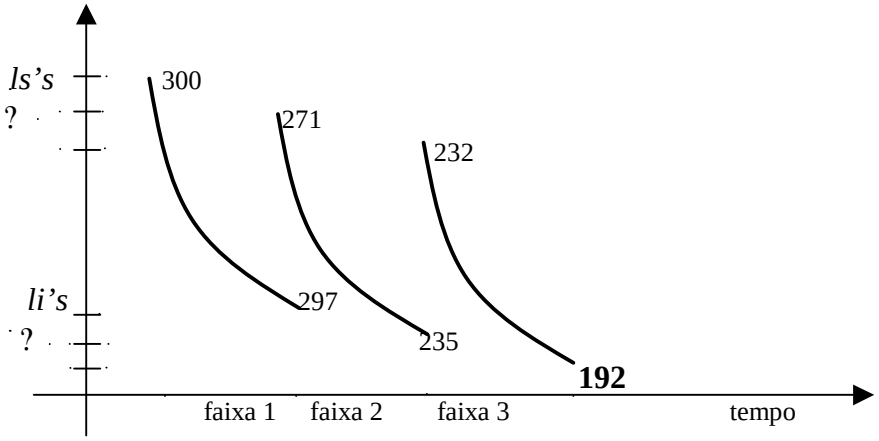

Figura 6.5 - Processo dois, com três faixas

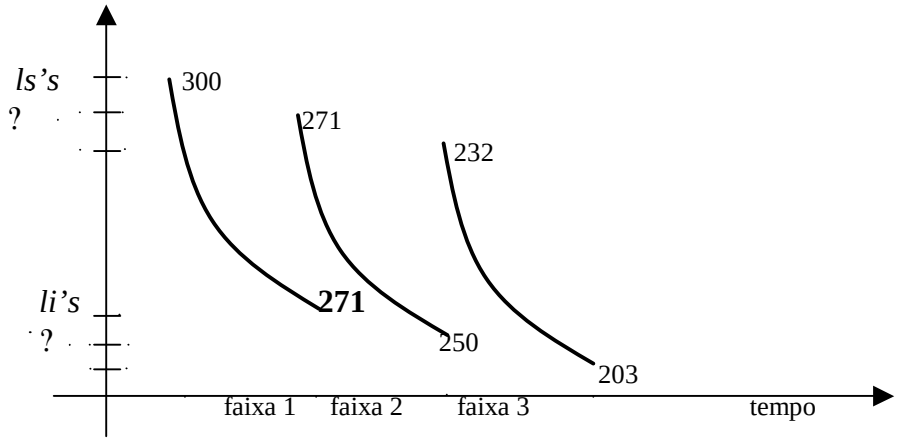

Figura 6.6 - Processo três, com três faixas

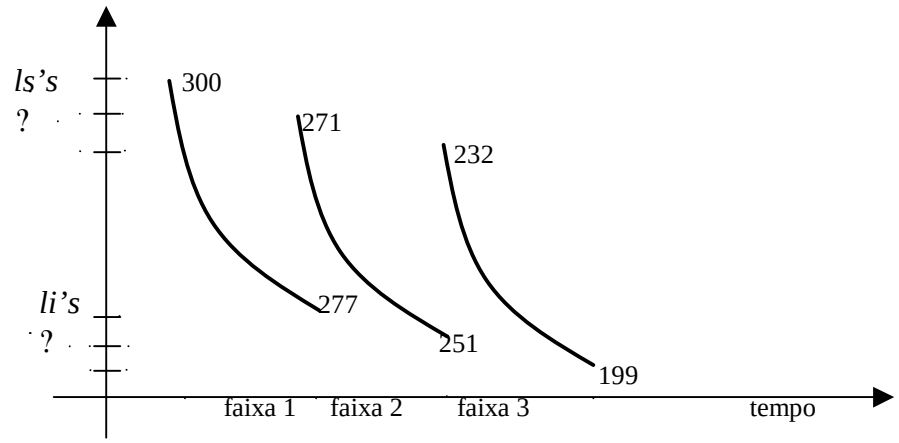

Figura 6.7 - Processo quatro, com três faixas

As figuras 6.4 a 6.7 demonstram o que acontece com os valores no início e final de cada faixa, onde ao final é selecionado o melhor custo entre os quatro processos e é escolhido o melhor, que será distribuído para os quatro processos iniciarem a nova faixa com essa rota.

Procedimento do método proposto

```
início
    ...
    Stemp ✍ S
    CustoTemp ✍ custo(S)
    CustoEnviado ✍ custo(S)
    NroProcessos ✍ MPI(size) - 1
    se MPI(rank) = 0 then
                para FAIXA de 1 até  NFAIXAS faça
                        para N de 1 até NroProcessos faça
                                S ✍ S do processo N
                                se Custo(S) < CustoTemp then
                                        CustoTemp ✍ Custo(S)
                                        Stemp ✍ S
                        fimfaça
                        Se CustoTemp < CustoEnviado then
                                CustoEnviado ✍ CustoTemp
                                para N de 1 até NroPprocesso faça
                                        envia S para S do processo N
                                fimfaça
                fimfaça

   senão
        ...
        T    ✍  ls;
        T_Min ✍ li;
         para FAIXA de 1 até  NFAIXAS faça
                enquanto ( T > T_Min ) faça
                para interações para equilíbrio faça
                        Gerar uma solução S' da N(S);
                                Avaliar variação de energia – ? E ✍ f(S') – f(S)
                                se ? E < 0 then
                                        S ✍ S';
                                senão
                                        Gerar u ?  random[0,1];
                                        se u  <  exp(- ? E / K_B * T) then
                                                S ✍ S';
                                        fimse
                                fimse
                        fimpara
                        Reduzir a temperatura T
                fimenquanto
                envia S para S do processo 0
                se FAIXA < NFAIXAS then
                        S  ✍ S do processo 0
                ls ✍ ls * ? ;
                li ✍ li * ? ;
                T    ✍  ls;
                T_Min ✍ li;
        fimfaçca
    fimpara
fimprocedimento
```

Figura 6.8 - Pseudocódigo do algoritmo proposto pelo método

6.5 Implementação

A implementação do algoritmo foi feita utilizando a linguagem de programação C, padrão ANSI, em ambiente Linux Conectiva, e como suporte para programação paralela foi utilizada a biblioteca de rotinas conhecida por MPI (*Message Passing Interface),* (SNIR et al. 1996 e PACHECO, 1998) introduzida no capítulo 5.

A plataforma para os experimentos foi um cluster com cinco máquinas, sendo um Pentiun III 800Mhz, e as restantes Pentiun IV 1.5Ghz. Todas dedicadas exclusivamente ao experimento.

CAPÍTULO 7

Análise dos Resultados

Neste capítulo é apresentada a metodologia e formas para avaliar o modelo, a origem das instâncias e por fim os resultados obtidos com a implementação do algoritmo básico, porém com múltiplas faixas, em comparação com a do modelo proposto.

7.1 Metodologia

Existem várias maneiras de avaliar a performance de algoritmos aproximados, como a *análise do pior caso, a análise probabilística e a análise empírica.* Todas têm seus pontos fortes e fracos. Porém, para avaliar a performance deste algoritmo que propõe soluções para o PCV, a *análise empírica* parece ser a mais simples e, portanto, a utilizada nesta avaliação.

Neste trabalho, o modelo proposto é avaliado através de um algoritmo derivado do *simulated anneling,* onde as temperaturas são retiradas de múltiplas faixas e tratadas de forma paralela. Esta avaliação atém-se ao desempenho do algoritmo em relação às várias faixas de temperatura, de modo sequencial e paralelo quando aplicado aos mesmos problemas-testes. O conjunto de problemas-testes utilizado para este tipo de análise é de instâncias com dados de situações reais e soluções conhecidas.

Os algoritmos (implementação básica com múltiplas faixas e implementação paralela) são executados várias vezes para os vários tamanhos de instâncias. Em cada execução é feito um número de tentativas igual ao do tamanho da instância, isto é, o

número de cidades. Toda a execução, independente da categoria, tem sua rota inicial gerada pela heurística que mostre melhor desempenho para aquele determinado tamanho de instância, portanto as várias tentativas têm a mesma rota inicial naquela rodada.

7.1.1 Avaliação

Avaliou-se o melhor resultado obtido de todas as execuções para um determinado tamanho de instância dentro da categoria. Este dado fornece a *qualidade da solução*, que é medida, em termos de percentagem, pela diferença entre a solução encontrada e a melhor solução ótima conhecida, e que é dada pela fórmula:

$$Q = \frac{f(.) - f_{opt}}{f_{opt}} \; x \; 100, \qquad (7.1)$$

onde $f(.)$ é o custo da solução encontrada pelo algoritmo, e f_{opt} o custo da melhor solução ótima conhecida para aquela instância.

Outro resultado que também foi avaliado é a *performance,* que mede o número de vezes em que o algoritmo numa determinada faixa alcançou a solução ótima, e se tal acontecer, será verificado através da seguinte fórmula:

$$P = \frac{Num_{opt}}{Num} \; x \; 100, \qquad (7.2)$$

onde Num_{opt} é o número de avaliações que alcançaram o ótimo global , e Num o número total de avaliações.

Por não ser o objetivo final deste trabalho, a eficiência da paralelização do algoritmo não foi avaliada, e sim os valores finais (custo da rota) encontrados a cada processo executado. Porém, foi feita uma pequena comparação do tempo gasto para executar o algoritmo de modo seqüencial comparado com o paralelo.

7.1.2 Origem das instâncias

As diversas instâncias do PCV têm suas origens baseadas nas mais diversas situações, que tanto podem representar situações do mundo real como do imaginário, simplesmente para estudo; e que também podem representar problemas com ou sem restrições. Dentre os problemas, os que mais têm causado interesse, principalmente em função de sua aplicabilidade, em situações práticas, são os que tratam de instâncias euclidianas (o problema é chamado euclidiano), ou seja, as cidades correspondem a pontos no plano, e as distâncias são calculadas na métrica euclidiana, que tem logicamente como restrição a desigualdade triangular e a simetria.

A maioria das instâncias do PCV é de natureza geométrica, portanto obedecendo à métrica euclidiana. A principal fonte de tais instâncias é a TSBLIB, que é uma biblioteca de instâncias para o PCV (TSP em inglês), colecionado por (REINELT, 1991 *apud* HELSGAUN, 2000) e disponível via internet, por intermédio do endereço eletrônico ftp://softlib.rice.edu/pub/tsplib. Lá também podem ser encontrados os resultados das mais recentes pesquisas sobre o assunto. A biblioteca contém instâncias do PCV, de várias fontes e com várias propriedades.

As instâncias utilizadas neste trabalho foram retiradas da TSPLIB, que contém instâncias de diversas origens e que também são as mais utilizadas, pois apresentam soluções ótimas conhecidas para a grande maioria delas.

7.2 Experimentos

Os experimentos do algoritmo *Simulated Annealing* estão baseados no algoritmo com abordagens de forma genérica, tendo uma rotina modificada para trabalhar com múltiplas faixas. Para a realização dos experimentos deste trabalho incorporamos o paralelismo no algoritmo, possibilitando que a cada troca de faixa de temperatura os processos recebam a melhor rota até então conhecida entre os processos cooperantes.

7.2.1 Rota inicial

Para determinar qual o procedimento de construção da rota inicial, vizinho mais próximo, geração randômica ou seqüência inicial das cidades, optou-se por utilizar como rota inicial aquela gerada a partir da leitura seqüencial das coordenadas das cidades, isto é, o primeiro par de coordenadas lido corresponde à primeira cidade da rota, o segundo par de coordenadas lido corresponde à segunda cidade da rota, e assim por diante, até completar a rota com todas as cidades.

A escolha se deu baseada no trabalho de ARAUJO (2001), que descreve os testes feitos em algumas instâncias do problema, onde a rota inicial foi obtida através de procedimentos heurísticos, mesmo havendo uma melhoria significativa no custo da rota inicial para a maioria das instâncias, quando se utilizou a heurística “vizinho mais próximo” ou “geração randômica”, comparado com a gerada seqüêncialmente, não tendo havido nenhum benefício acentuado em relação ao custo da rota final. Diante de tais experimentos já realizados, justifica-se o fato de não se usar nenhuma forma heurística para obtenção da rota inicial na realização dos experimentos deste trabalho, sendo assim, a rota inicial é construída automaticamente, de forma seqüencial, conforme sua ordem quando lida no arquivo. Essas idéias são igualmente traçadas pelos argumentos de LIN & KERNICHAN (1973). Eles dizem que utilizar heurística para construir a rota inicial, em alguns casos, é desperdício de tempo, já que rotas construídas são comumente determinísticas e pode não ser possível ocorrer a obtenção de mais de uma rota inicial.

7.2.2 Desempenho em relação ao número de processos paralelos

Após definirem-se os valores dos parâmetros da implementação do algoritmo pelo método de tentativa e erro realizando-se diversos testes com várias instâncias do problema, iniciaram-se os testes com o modelo já conhecido de múltiplas faixas de temperatura em um processo de modo seqüencial e o modelo proposto, algoritmo com múltiplas faixas de temperatura rodando de forma paralela, efetuando a cada faixa a troca de mensagens com a rota atual de cada processo.

Para executar os experimentos, utilizaram-se instâncias do PCV de tamanho razoável, com 100, 150 e 200 cidades e, portanto, de solução não trivial para algoritmos baseados na abordagem do SA. Os testes foram executados comparando-se os

resultados obtidos com 1, 5, 10 e 15 processos sendo executados. Quando nos referimos a um processo, o mesmo representa a implementação seqüencial do algoritmo já implementado e discutida em ARAUJO (2001), o qual queremos comparar com o processo paralelo, observando quais resultados podemos ter quando trocamos informações sobre a rota existente a cada troca de faixa de temperatura.

Os resultados tabelados em 7.1, 7.2 e 7.3 representam os valores máximos, médios e mínimos obtidos conforme o número de processos paralelos envolvidos, para cada instância com 100, 150 e 200 cidades. Os valores mostrados em cada uma das linhas das tabelas, para as três classes de instâncias, foram obtidos após 150 execuções.

Cabe resaltar que todos os testes executados por um número de processos foram realizados de forma independente, ou seja, para se obter o valor mínimo para a linha correspondente a um processo em andamento, foram feitas, por exemplo, 150 execuções. Observando que para o número de iterações finais ser o mesmo em todos os experimentos, optou-se por igualar o número de processos executados na sua totalidade, ou seja, para a linha correspondente a um processo rodando, foram feitas 150 execuções, já para a linha correspondente a 5 processos, foram realizadas 30 execuções com 5 processos cooperantes em cada um dos testes. E assim, sucessivamente, na linha de 10 processos, 15 execuções com 10 processos e na linha de 15 processos, 10 execuções com 15 processos.

	Valores			
Núm. De Processos	Máximo	Médio	Mínimo	Desvio Padão
1	21412	21320	21285*	34,71
5	21383	21295	21285*	21,73
10	21370	21289	21285*	12,86
15	21356	21287	21285*	8,81

Tabela 7.1 – Instâncias com 100 cidades
* Valor ótimo conhecido.

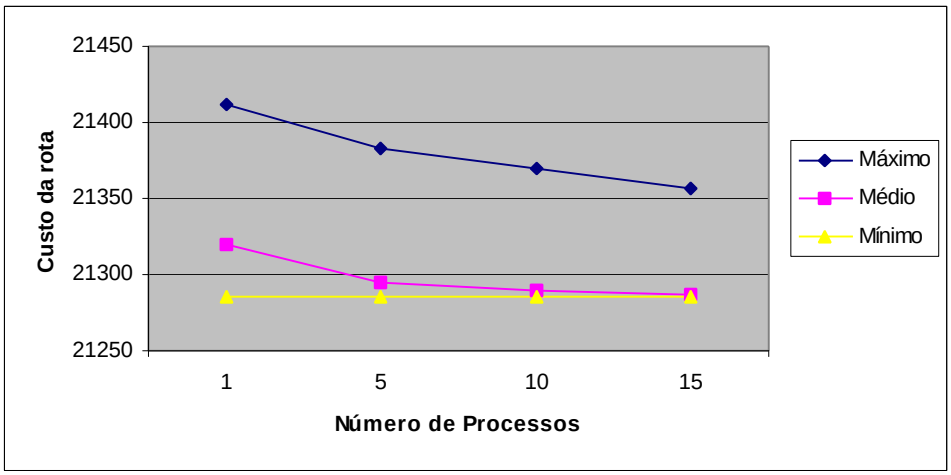

Figura 7.1 – Instância com 100 cidades.

Analisando a figura 7.1 podemos observar que os resultados já evidenciam uma tendência de melhores valores à medida que aumenta o número de processos cooperantes. No entanto, como retrata a figura 7.2, com instância maior, 150 cidades, nota-se a elevação do valor máximo quando chegamos a 15 processos cooperantes. Contudo, em novos testes realizados com esta instância, 150 cidades, observou-se que esta tendência não comprometia os resultados do algoritmo, pois estas elevações no valor máximo encontrado eram esporádicas, levando então à escolha dos primeiros resultados encontrados para tabular neste trabalho.

	Valores			
Núm. De Processos	Máximo	Médio	Mínimo	Desvio Padão
1	26858	26653	26534	73,89
5	26770	26588	26524*	51,37
10	26692	26564	26524*	38.13
15	26748	26553	26524*	32,90

Tabela 7.2 – Instância com 150 cidades
* Valor ótimo conhecido.

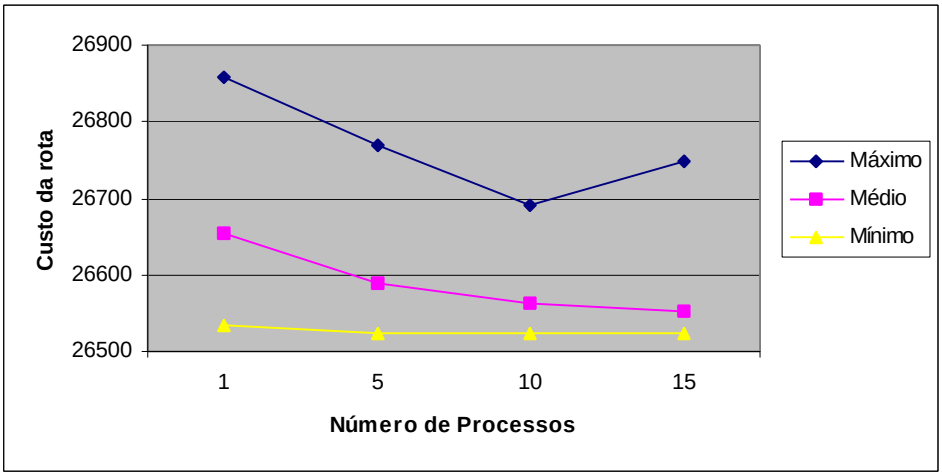

Figura 7.2 – Instância com 150 cidades.

Durante todo o processo dos testes realizados com a instância de 150 cidades, quando executamos o algoritmo com um processo executando-o de modo independente, não alcançamos o valor mínimo conhecido. Contudo, conforme dados tabulados na tabela 7.2 e apresentados graficamente na figura 7.2, já confirmamos a eficácia do algoritmo quando incrementamos a troca de informações durante sua execução, pois, no grande grupo de testes, 150 execuções, sempre encontramos o valor ótimo conhecido para essa instância, quando executamos os testes com processos cooperantes.

Os testes realizados com a instância de 200 cidades mostra uma outra realidade: o que na instância de 150 cidades não era tendencioso, na de 200 cidades não é o mesmo que se observa. Essa tendência evidencia-se à medida que aumentamos o número de

processos cooperantes, decidimos então aumentar nessa instância o número de processos cooperantes dos experimentos, incluindo experimentos com 20 processos.

	Valores			
Núm. De Processos	Máximo	Médio	Mínimo	Desvio Padão
1	29828	29602	29483	81,91
5	29727	29531	29441	60,73
10	29734	29529	29416	62,33
15	29778	29550	29412	32,90
20	29794	29562	29406	28,75

Tabela 7.3 – Instância com 200 cidades

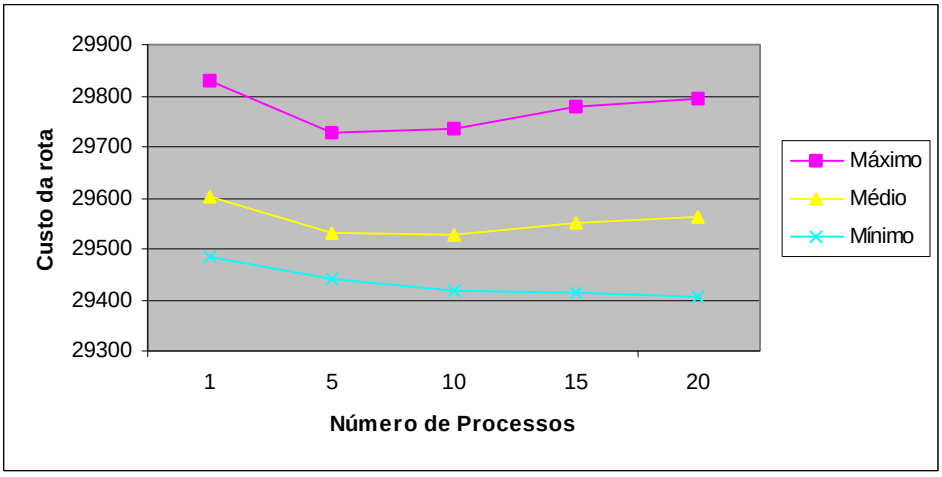

Figura 7.3 – Instância com 200 cidades.

Apesar de não apresentar o mesmo desempenho com os valores máximos e médios, o algoritmo aplicado na instância de 200 cidades mostrou-se eficaz no sentido de alcançar melhores resultados, ou seja, à medida que aumentamos o número de processos obtivemos resultados mais próximos dos valores mínimos conhecidos, como mostra a tabela 7.3 e a figura 7.3.

7.2.3 Acertos por número de processos cooperantes

Mesmo para uma abordagem típica de SA, obter resultados ótimos para instâncias de tamanho razoável não é tão freqüente. Esse modelo mostrou-se bastante otimista ao alcançar muitas soluções ótimas, principalmente à medida que o número de processos cooperantes aumenta. Essa taxa de acertos foi obtida executando-se o algoritmo 150 vezes. Para um processo rodando, foram feitas 150 execuções, para 5 processos, foram realizadas 30 execuções com 5 processos cooperantes em cada um dos testes. E assim sucessivamente, na linha de 10 processos, 15 execuções com 10 processos e na linha de 15 processos, 10 execuções com 15 processos. Execuções sempre feitas com instâncias de 100, 150 e 200 cidades.

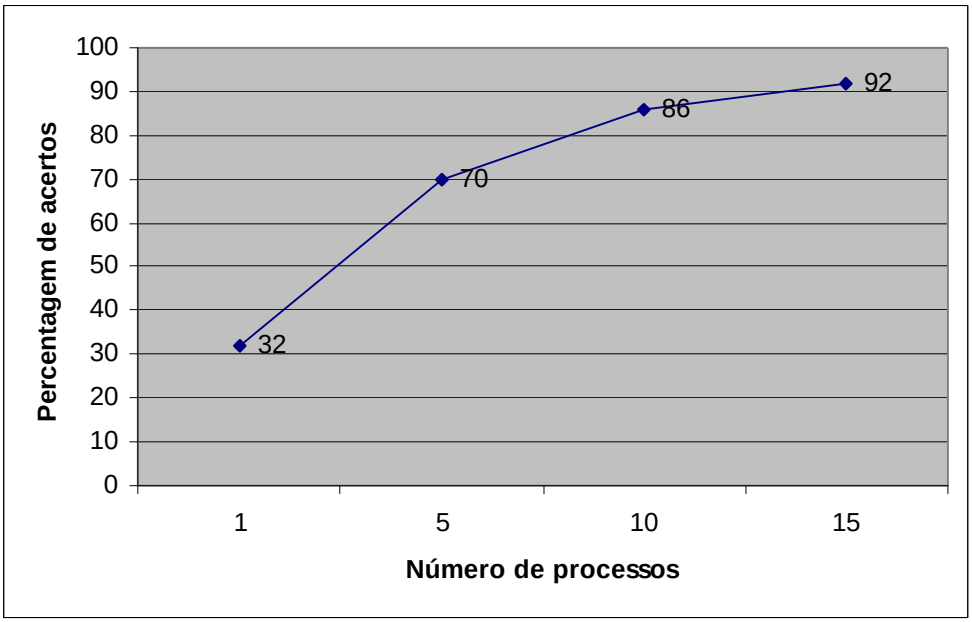

Figura 7.4 – Performance do modelo com instância de 100 cidades.

O gráfico da figura 7.4 reflete o comportamento do modelo, no qual, à medida que o número de processos cooperantes aumenta, aumenta também o percentual de acerto com solução ótima.

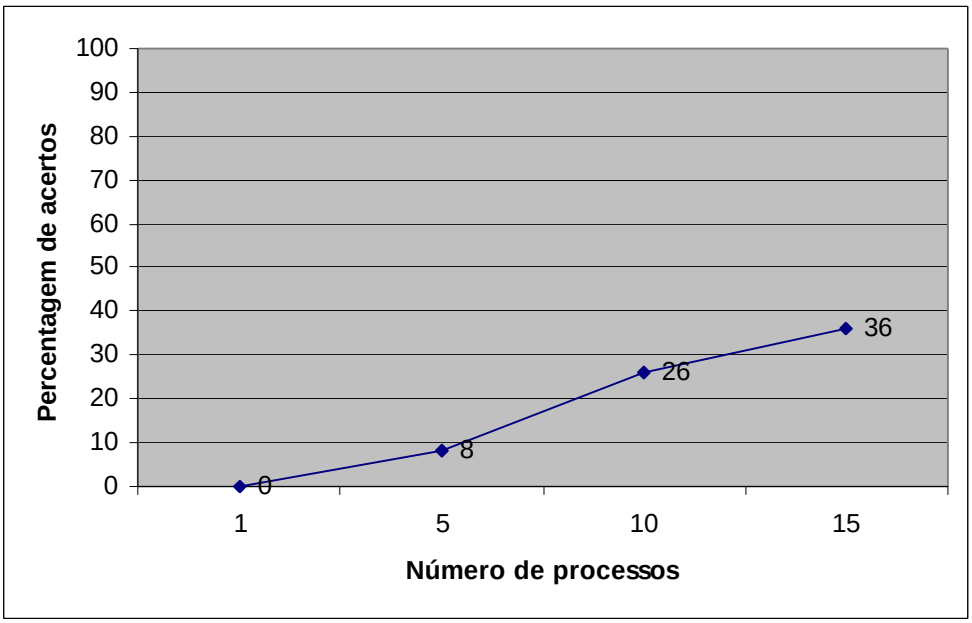

Figura 7.5 – Performance do modelo com instância de 150 cidades.

Comportamento idêntico ao do gráfico da figura 7.4 pode também ser observado no gráfico da figura 7.5, onde o algoritmo foi aplicado a uma instância de 150 cidades. A evidência de melhora está clara não só pelo aumento de acertos, à medida que o número de processos aumenta, mas também, pelo método existente até então não ter encontrado nenhuma solução ótima nos 150 testes realizados.

		Método anterior 8 Faixas 1 Proc.		Método Paralelo 8 faixas 5 Proc.		Método Paralelo 8 faixas 10 Proc.		Método Paralelo 8 faixas 15 Proc.	
Núm. de Cidades	Ótimo Conhecido	% de acerto	% Médio acima ótima	% de acerto	% Médio acima ótima	% de acerto	% Médio acima ótima	% de acerto	% Médio acima ótima
100	21285	32	0,17	70	0,05	86	0,02	92	0,01
150	26524	0	0,49	8	0,24	26	0,15	36	0,11
200	29368	0	0,80	0	0,56	0	0,54	0	0,66

Tabela 7.4 – Desempenho geral do método

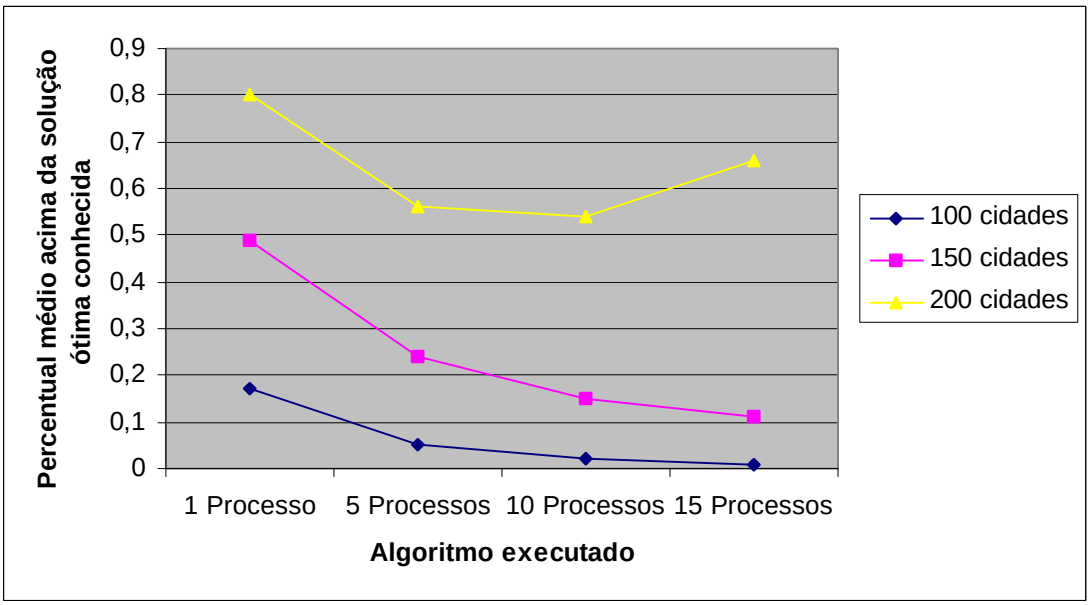

Figura 7.6 – Desempenho geral do método paralelo

Cabe ressaltar que todas as execuções dos experimentos foram feitas dentro do mesmo conjunto de parâmetros, ou seja, para as instâncias maiores não foi feita nehuma alteração diferenciada, senão as definidas pelas suas próprias configurações. Os dados tabelados em 7.4 demonstram o desempenho geral do método e foram obtidos pela aplicação das fórmulas 7.1 e 7.2, correspondentes ao percentual de acertos e ao percentual médio acima do ótimo conhecido, respectivamente.

7.2.4 Tempo de execução

Para encontrar a relação existente entre o tempo dispensado na execução do algoritmo em um único processador (T1) e o tempo gasto para executá-lo em *p* processadores (Tp), conhecido como *speedup*, foram utilizados os mesmos experimentos anteriores, aplicados com a instância de 200 cidades. A média de tempo de várias execuções do algoritmo obteve o seguinte resultado:

T1 = 452 segundos

T5 = 89 segundos $\qquad speedup \; ? \; \frac{T1}{Tp}$

Speedup = 5.08

Conforme citado no ítem 4.5, quando o *speedup* se iguala a *p,* indica que o caso ótimo foi atingido, siginificando que o aumento da capacidade de processamento é diretamente proporcional ao número de processadores. Observando o valor calculado do *Speedup,* que é 5.08, fica claro que é um resultado satisfatório, porém, o esperado no final da execução do algoritmo, que é obter o custo mínimo da rota (menor distância), neste caso, por exemplo, para a instância de 200 cidades, não foi alcançado. Isso ocorre porque, quando o algoritmo é executado seqüencialmente em um único processador, não existem processos paralelos trocando informações, que é o objetivo deste trabalho para melhorar a qualidade dos resultados finais (distância/tamanho da rota).

CAPÍTULO 8

Considerações Finais

São destacados, neste capítulo, a contribuição sobre o tema trabalhado, bem como, a colaboração que o mesmo oferece para a resolução de problemas de um modo geral. Também são sugeridos alguns tópicos não abordados nessa implementação, os quais podem ser de grande valia para futuras pesquisas.

8.1 Conclusões

Com a necessidade de uma boa administração dos custos e maior atenção à produtividade, cada vez é maior a necessidade de investimentos na qualificação dos processos que, decisivamente, agregam valor à tomada de decisões e potencializam a correta leitura dos resultados efetivos.

O desenvolvimento tecnológico tem contribuído, fortemente, para a evolução desses processos, principalmente, na área de otimização, evidenciando uma forte tendência em se utilizar métodos aproximados na resolução desta classe de problemas. Uma abordagem dessa natureza, apesar de não garantir uma solução ótima para um problema, é capaz de oferecer uma solução de boa qualidade. No entanto, um dos fatores que pode levar ao fracasso a utilização destes métodos, é o tempo de processamento. O considerado aumento do interesse por abordagens baseadas nas técnicas de otimização que podem utilizar como base o algoritmo *Simulated Annealing* demonstra cada vez mais a importância de se encontrar um algoritmo aproximado para

resolver esses problemas que podem mostrar soluções viáveis para muitos dos problemas encontrados nos sistemas de informações das organizações.

Os algoritmos como o SA, desenvolvidos para resolver problemas de otimização, cumprem seu papel satisfatoriamente, porém, muitas vezes o tempo de processamento impõe restrições à utilização do algoritmo em determinados problemas reais. Diante disso, surge uma grande possibilidade de avanço com a implementação do algoritmo sendo executado em paralelo. Necessita-se, assim, de suporte por parte do algoritmo, devido ao fato da seqüência de instruções ser um conceito básico do algoritmo e não permitir que esse seja automaticamente paralelizável.

Neste trabalho, foi proposta e implementada a potencialidade de um modelo genérico do SA para resolução dos problemas de otimização, sendo executado tal modelo de modo paralelo, com troca de informações sobre as melhores rotas até então encontradas em outros processos cooperantes.

Para efeitos de testes foi utilizado o PCV, em razão da sua vasta área de aplicabilidade, e pelo fato de o PCV ser um problema de grande interesse dos pesquisadores, contribuindo para a apresentação de novos métodos que resultem em boas soluções.

A qualidade da abordagem proposta neste trabalho, fruto das inovações realizadas, demonstra a competitividade do método, com experimentos realizados com dados de várias instâncias e número de processos cooperantes diferentes, evidenciando uma boa alternativa para problemas que utilizam o algoritmo SA para sua resolução.

Os resultados alcançados pelos testes mostram que o método proposto, quando comparado à implementação básica do SA e à implementação com 8 faixas em 1 processo, alcança a solução ótima com mais freqüência, e em muitos casos só o método proposto alcançou a solução ótima nas instâncias testadas. Esses resultados traçam um horizonte otimista em relação ao potencial do método proposto, já que são os primeiros testes realizados com o algoritmo SA sob várias faixas de temperatura, rodando de modo paralelo com troca de informações durante uma execução. Cabe salientar que nesse sentido existem duas referências sobre múltiplas faixas, a de ARAUJO (2001), o qual implementou o modelo de múltiplas faixas para o SA, e a encontrada em MAZZUCCO (1999), essa porém direcionada a um método híbrido aplicado aos

problemas de programação da produção. Não existe, portanto, ao menos na literatura consultada, qualquer referência teórica ou prática ao método proposto, implementado e testado.

Outro fato relevante, é o fascinante modo proposto de resolver problemas, muito diferente de qualquer tipo de programação a que se estava habituado. Por outro lado, ao contrário do que se acredita habitualmente, pensar em esquemas de programação paralela é entusiasmante. Da mesma forma, não se referindo à programação paralela como uma abstração, mas enquanto possibiliade de abordar um problema, é preferível pensar em vários acontecimentos a serem trabalhados em simultâneo do que pensar na seqüência exata dos acontecimentos.

No entanto, o fator mais entusiasmante da pesquisa foi o de, ao contrário do que muitas vezes é comum neste tipo de trabalho, não se estar poduzindo algo que já tenha sido completamente explorado e sobre o que todas as perguntas tenham sido respondidas, mas, mais do que isso, ter-se entrado em um meio em plena efervescência, com muitos esforços em diversos níveis para encontrar as melhores soluções para os diversos problemas que então foram enfrentados.

Mesmo se acreditando que o método proposto proporcionou conclusões novas acerca do assunto, infelizmente, não é vulgar que isso aconteça no mundo da informática, em que para alguns, tudo parece já ter sido inventado em todos os formatos possíveis. Por outro lado, chega-se também ao fim com a impressão que tudo é muito simples. Afinal, onde está o tempo gasto na elaboração desse trabalho?

8.2 Pontos a explorar

O método estudado abordou diversos assuntos relacionados ao enfoque principal do trabalho. No entanto, decidir quais questões julgadas relevantes seriam abordadas e quais levariam para um novo trabalho, foi um imperativo. Diante disso, ficam lacunas cujo preenchimento sugerimos como tema para futuras pesquisas no assunto:

- Ajuste dos parâmetros:

  Os parâmetros que foram utilizados para os experimentos advieram de um trabalho já existente e não tiveram adaptação para o modelo paralelo, podendo ocorrer mudanças à medida que manipulamos os parâmetros conforme o tamanho da instância e o número de processos cooperantes;

- Processos paralelos para cálculos diversos:

  O algoritmo SA executa uma série de procedimentos independentes que poderiam ser alocados para processos separados do principal, por questão de melhora na performance;

- Tolerância a falhas:

  Como se partiu do princípio de que neste trabalho os processos serão executados em máquinas totalmente dedicadas aos experimentos, e se alguma anomalia acontecesse poderíamos iniciar novamente os testes, deixou-se de lado o estudo da tolerância a falhas;

- Distribuição de carga:

  Do mesmo modo que o problema de tolerância a falhas, esta questão, que aqui poderia ser abordada, ficou para futuros trabalhos;

- Arquiteturas diferentes:

  Os testes foram realizados em um cluster montado com máquinas de arquiteturas e sistema operacional iguais, seria interessante verificar os resultados quanto ao aspecto do tempo em arquiteturas e sistemas operacionais diferentes.

REFERÊNCIAS BIBLIOGRÁFICAS

AARTS, E. AND KORST, J. *Simulated annealing and Boltzmann Machines.* John Wiley & Sons, 1990.

ALMASI, G. S., Gottlieb, A. *Highly Parallel Computing.* 2ª ed. The Benjamin Cummings, 1994.

AMORIM, C. L., et al. *Uma introdução a computação paralela e distribuída.* VI Escola de Computação, 1988.

ANDREWS, G. R. *Paradigms for process Interaction in Distributed Programs,* CM Computing Surveys, v. 23, n. 1, março de 1991, p. 49-90.

ANDREWS, G. R., Schneider, F. B. *Concepts and notations for concurrent programming*, ACM Computing Survey, v. 15, n.1, p.3-43, 1983.

APPLEGATE, D. et al. Microelectronics Technology Alert, July 1998, página HTML Disponível em: <(http://www.crpc.rice.edu/CRPC/news/Archive/mta_7_24_98.html>

ARAÚJO, HAROLDO ALEXANDRE DE. Algoritmo *Simulated Annealing*: uma nova abordagem, (Dissertação de Mestrado) CPGCC / UFSC, 2001.

BAL, H. E., STEINER, J. G., TANENBAUM, A. S. *Programming Languages for Distributed Computing Systems*, ACM Computing Surveys, v. 21, n. 3, 1989, p. 261- 322.

BEN-ARI. *Principles of Concurrent and Distributes Programming.* Prentice Hall, 1985.

BEN-DYKE, A. D. 'Architectural taxonomy, A brief review', University of Birmingham, 1993.

BLAND, R. G. and SHALLCROSS, D. F. *Large traveling salesman problems arising from experiments in X-ray crystallography: A preliminary report on computation*, Operations Research Letters, v. 8, 1989, p. 125-128.

BLECH, R. A. *An overview of parallel processing*, Slides, Parallel Computing with PVM Workshop, Nasa Lewis Research Center. Disponível em: <http://www.lerc.nasa.gov/Other_Groups/IFMD/2620/tutorialPP.html>, 2001.

CAMPELLO R. e N. MACULAN. *Algoritmos e Heurísticas* - Desenvolvimento e Avaliação de Performance. Niterói, EDUFF, 1994.

CERNY, V. *Thermodynamical Approach to the Traveling Salesman Problem: An Efficient Simulation Algorithm, Journal of Optimisation Theory and Applications*, n. 45, 1985, p. 41-51.

COOK, S. *The complexity of Theorem Proving Procedures,* Proc. 3rd ACM Symp. On the Theory of Computing, 1971, p. 151-158.

DANTZIG, G., FULKERSON, R. AND JOHNSON, S. *Solution of a large-scale traveling-salesman problem,* Operations Research, vol. 36, 1954, p. 393-410.

DONGARRA, Jack et al. *MPI: The Complete Reference*. Massachusetts Institute of Tecnology. Disponível em: <http://www.netlib.org/utk/papers/mpi-book/mpi-book.html>, 2001.

DUNCAN, R. *A Survey of Parallel Computer Architectures.* IEEE Computer, fev. de 1990.

FLYNN, M. J. *Some Computer Organizations and Their Effectiveness.* IEEE Transactions on Computers, v. C-21, pp.948-960, 1972.

GAREY, M. R. and JOHSON, D. S. *Computers and Intractability: A Guide to the Theory of NP-Completeness.* San Francisco: W.H.Freeman, 1979.

GU, J. and HUANG, X. *Efficient local search with search space smoothing: a case study of the traveling salesman problem (TSP)* IEEE Trans. Systems, Man, and Cybernetics, vol. 24, 1994, p. 728-735.

GUTIN, G. AND YEO, A. *Small diameter neighbourhood graphs for the traveling salesman problem.* Technical Report DMS of Brunel University, 1998.

HELSGAUN, K. *An effective implementation of the Lin-Kernighan traveling salesman heuristic. European Journal of Operational Research*, v. 126, 2000, p. 106-130.

JOHNSON, D. S. and MCGEOCH, L. A. *The traveling salesman problem: A case study in local optimization,* in (E.H.L. Aarts and J. K. Lenstra,eds), *Local Search in Combinatorial Optimization*, Wiley & Sons, New York, 1997.

KIRKPATRICK, S., GELATT, Jr. C. D. and VECCHI, M. P. *Optimization by Simulated Annealing. Science,* n. 220, 1983, p. 671-680.

KIRNER, C. *Arquiteturas de sistemas avançados de computação,* Anais da Jornada EPUSP/IEEE em Sistemas de Computação de Alto Desempenho, pp. 307-353, 1991.

KITAJINA, J.P.F.W. *Programação paralela utilizando mensagens.* In: *XV Congresso da Sociedade Brasileira de Computação, XIV Jornada de Atualização em Informática*, 1995.

KORTE, B. *Applications of combinatorial optimization*, talk at the 13th International Mathematical Programming Symposium, Tokyo, 1988.

LAPORTE, G. *The traveling salesman problem: an overview of exact and approximate algorithms. European Journal of Operational Research*, v. 59, 1992, p. 231-247.

LAWLER, E. L. et al. *The Traveling Salesman Problem: a guided tour of combinatorial optimization.* Chichester: John Wiley &Sons, 1985.

LEWIS, H. R. and PAPADIMITRIOU, C. H. *Elementos de Teoria da Computação.* Trad. Edson Furmankiewicz. 2.ed. Porto Alegre: Bookman, 2000.

LIN, S. AND KERNICHAN, B W., *AnEffective Heuristic Algorithm for the Traveling Salesman Problem,* Operations Research, vol.21 1973, p. 498-516.

LUNDY, M. AND MEES, A. *Convergence of an annealing algorithm.* Mathematical Programming, v. 34, 1986, p. 111-124.

MARTIN O. C. and OTTO, S. W. *Combining simulated annealing with local search heuristics*, Annals of Operations Research, V. 60, Balzer Scientific Publishers, Amsterdam, 1995.

MAZZUCCO JUNIOR, J. *Uma Abordagem Híbrida do Problema da Programação da Produção através dos Algoritmos Simulated Annealing e Genético.* (Tese de Doutorado) PPGEP / UFSC, 1999.

MCBRYAN, O. A. *An overview of message passing environments.* Parallel Computing, v. 20, pp. 417-444, 1994.

METROPOLIS, W. et al. *Equation of State Calculations by Fast Computing Machines, Journal of Chemical Physics*. v. 21, 1953, p. 1087-1092.

PACHECO, P. S. *A User's Guide to MPI.* Department of Mathematics, University of San Francisco, 1998.

PAPADIMITRIOU, C. H. and STEIGLITZ, K. *Combinatorial optimization: algorithms and complexity.* Englewood Cliffs. N. J.: Prentice-Hall,1982.

PUNNEN, A. P. *The traveling salesman problem: new polynomial approximation algorithms and domination analysis.* Manuscript, Dec. 1996.

QUEALY, Angela, *An introduction to Message Passing*, slides, NASA Lewis Research Center, 1994. Disponível em: <http://www.lerc.nasa.gov/Other_Groups/IFMD/2620/tutorialPP.html>, 2001.

QUINN, M.J. *Designing Efficient Algorithms for Parallel Computers.* São Paulo: McGraw Hill, 1987.

REEVES, C. R. *An Improved Heuristic for the Quadratic Assignment Problem. Journal of the Operation Research Society*, v. 36, 1985, p. 163-167.

REINELT, G. *A traveling salesman problem library, ORSA journal of computing*, v. 3, 1991, p. 376-384.

ROUTO, Terada. *Desenvolvimento de algoritmos e estruturas de dados.* São Paulo: McGraw-Hill, Makron, 1991.

SNIR, M. et al. *MPI: The Complete Reference.* Massachusetts: The MIT Press, 1996.

SOUZA, Márcio Augusto de Avaliação das Rotinas de Comunicação Ponto a Ponto`do MPI. (Dissertação de Mestrado) ICMSC / USP, 1996.

TANENBAUM, A. S. *Modern Operating System.*, Prentice-Hall, 1992.

TANOMARU, J. *Motivação, Fundamentos e Aplicações de Algoritmos Genéticos,* In: *II Congresso Brasileiro de Redes Neurais*,Curitiba-Pr, 1995.

TREW, A., WILSON, G. *Past, Present, Parallel: a Survey of Available Parallel Computing Systems.* Springer-Verlay, 1991.

ZALUSKA, E. J. *Research lines in distributed computing systems and concurrent computation. Anais do Workshop em Programação Concorrente, Sistemas Distribuídos e Engenharia de Software*, pp. 132-155, 1991.