UNIVERSIDADE FEDERAL DE SANTA CATARINA
PROGRAMA DE PÓS-GRADUAÇÃO EM ENGENHARIA DA PRODUÇÃO

UMA ABORDAGEM HÍBRIDA DO PROBLEMA DA PROGRAMAÇÃO DA PRODUÇÃO ATRAVÉS DOS ALGORITMOS *SIMULATED ANNEALING* E GENÉTICO

Elaborado por **José Mazzucco Junior** como requisito parcial para obtenção do grau de Doutor em Engenharia de Produção

Banca Examinadora:

Prof. Ricardo Miranda Barcia, Ph.D. - orientador (UFSC)
Prof. Marco Antônio Barbosa Cândido, Dr. - (PUC - PR)
Prof. Rômulo Silva de Oliveira, Dr. - (UFRGS)
Prof. Rogério Cid Bastos, Dr. - (UFSC)
Prof. Fernando Álvaro Ostuni Gauthier, Dr - (UFSC)
Prof. Pedro Alberto Barbetta, Dr. - moderador (UFSC)

0.300.328-2

Florianópolis, 1999.

BU

UFSC-BU

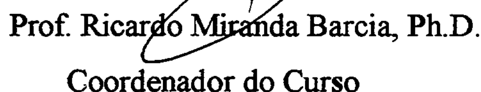
UNIVERSIDADE FEDERAL DE SANTA CATARINA

PROGRAMA DE PÓS-GRADUAÇÃO EM ENGENHARIA DA PRODUÇÃO

UMA ABORDAGEM HÍBRIDA DO PROBLEMA DA PROGRAMAÇÃO DA PRODUÇÃO ATRAVÉS DOS ALGORITMOS *SIMULATED ANNEALING* E GENÉTICO

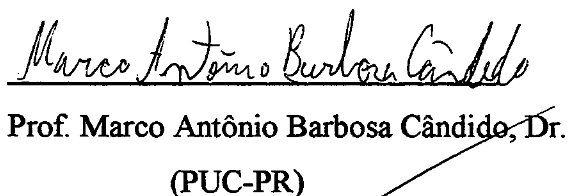
Elaborado por **José Mazzucco Junior.**

Esta tese foi julgada adequada para a obtenção do título de Doutor em Engenharia de Produção, em sua forma final, pelo Programa de Pós-Graduação em Engenharia de Produção.

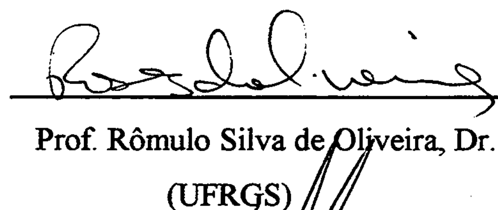


Prof. Ricardo Miranda Barcia, Ph.D.
Coordenador do Curso

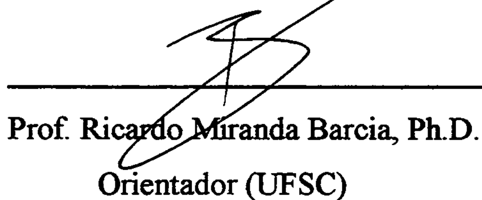
Banca Examinadora:



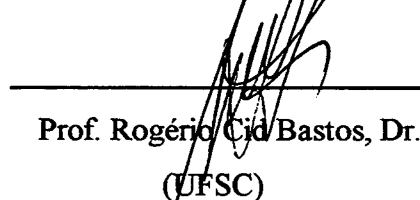
Prof. Marco Antônio Barbosa Cândido, Dr.
(PUC-PR)



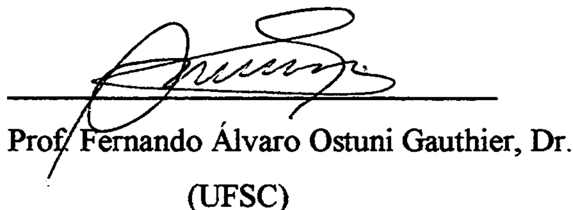
Prof. Rômulo Silva de Oliveira, Dr.
(UFRGS)



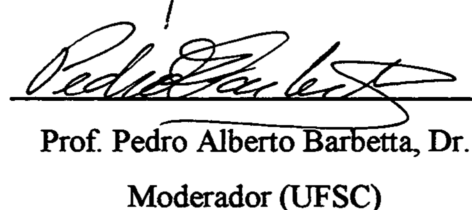
Prof. Ricardo Miranda Barcia, Ph.D.
Orientador (UFSC)



Prof. Rogério Cid Bastos, Dr.
(UFSC)



Prof. Fernando Álvaro Ostuni Gauthier, Dr.
(UFSC)



Prof. Pedro Alberto Barbetta, Dr.
Moderador (UFSC)

Aos meus pais José e Margarida,
à minha esposa Marise e
à minha filha Maiara.

AGRADECIMENTOS

Quero registrar os meus agradecimentos ao Prof. Ricardo Miranda Barcia, orientador e amigo, pelos incentivos e contribuições recebidas no decorrer do trabalho.

Também não poderia deixar de agradecer aos professores integrantes da banca examinadora pelas críticas construtivas e elogios recebidos.

Sou grato também aos professores e funcionários do meu departamento INE e aos funcionários da secretaria do PPGEP.

RESUMO

Dentre os principais fatores que compõem o controle e o gerenciamento da produção industrial encontra-se a programação da produção. Este fator, especialmente em se tratando de programação da produção do tipo *job shop*, se traduz em uma tarefa árdua devido a sua grande complexidade. Trata-se de um dos problemas mais difíceis de otimização combinatorial conhecidos. Informalmente, o problema da programação da produção do tipo *job shop* pode ser descrito da seguinte forma. São dados um conjunto de *jobs* e um conjunto de máquinas. Cada *job* consiste de uma cadeia de operações em que cada uma das quais deve ser processada durante um período de tempo ininterrupto, de um dado tamanho, em um dada máquina. Cada máquina pode processar no máximo uma operação por vez. Um programa consiste da alocação das operações em cada máquina. O problema é determinar um programa que realize todas as operações, no menor tempo possível.

Atualmente observa-se 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, em um tempo de processamento aceitável.

Neste trabalho, foi proposto e investigado a potencialidade de mais um método geral, baseado na combinação dos algoritmos *simulated annealing* (SA) e genético (AG). O termo "geral" aqui empregado denota a independência do método com relação ao problema que está sendo tratado. A grande maioria dos métodos de aproximação existente utiliza regras heurísticas de prioridade durante o processo de escalonamento. Esta característica, que obriga a incorporação de conhecimentos particulares do problema ao método de resolução, torna-o fortemente dependente do problema.

ABSTRACT

One of the main issues in production planning and control is production scheduling. Especially in job shop environments, the scheduling problem is very difficult. The job shop scheduling problem is known as one of the hardest problems in combinatorial optimization. Informally, the problem can be described as follows. A set of jobs needs to be processed on a set of machines. Each job consists of a predefined sequence of operations. Each operation needs to be processed during an uninterrupted time period on a specific machine. Each machine can process at most one operation at a time. A *schedule* is a feasible allocation of operations to time intervals on the machines. The problem is to find a schedule that minimizes the *makespan*.

Nowadays, there is a strong tendency to use approximate methods to solve this class of problems. Such approaches are able to generate good solutions in a reasonable amount of time. However, they do not guarantee an optimal solution for the problem.

In this work, a new general method is proposed and investigated. It is based on a hybridization of simulated annealing and genetic algorithms. The term *general* is used in order to denote the independence of the method on the problem instance. Most of the approximate methods use priority rules, i.e., rules for choosing an operation from a subset of operations that have not yet been scheduled. However, to use heuristic rules one must incorporate problem specific knowledge in the solution procedure. Therefore these methods become strongly dependent on the problem instance.

SUMÁRIO

1 - INTRODUÇÃO 1

1.1 - Objetivo do trabalho 3

1.2 - Organização 3

2 - PROGRAMAÇÃO DA PRODUÇÃO 5

2.1 - Conceituação básica 5

2.2 - Classificação dos problemas da programação da produção 7

2.2.1 - Geração de pedidos 7

2.2.2 - Complexidade do processamento 8

2.2.3 - Critério de escalonamento 9

2.3 - A classe do problema tratado 10

2.3.1 - Definições e premissas 10

2.4 - O método de resolução utilizado 12

3 - O ALGORITMO GENÉTICO E *SIMULATED ANNEALING* 15

3.1 - Introdução ao algoritmo genético 15

3.1.1 - Descrição do algoritmo genético 19

3.1.1.1 - O processo de avaliação 20

3.1.1.2 - O processo de seleção 21

3.1.1.3 - O processo de reprodução 25

3.1.2 - O teorema fundamental do algoritmo genético 27

3.2 - Introdução ao algoritmo *simulated annealing* 32

3.2.1 - *Simulated annealing* como um método de otimização 33

3.2.2 - Considerações teóricas 35

4 - COMPUTAÇÃO PARALELA 36

4.1 - Motivação 36

4.2 - Sistemas computacionais distribuídos 38

4.3 - Arquiteturas de multicomputadores 40

4.3.1 - Redes de interconexão de barramento 41

4.3.2 - Redes de interconexão bipontuais 42
4.3.2.1 - Redes bipontuais estáticas 42
4.3.2.2 - Redes bipontuais dinâmicas 44
4.4 - Classes de aplicação de sistemas distribuídos 46
4.5 - Suportes para programação distribuída 48
4.6 - Modelos de linguagens de programação distribuída 50
4.6.1 - Paralelismo 53
4.6.1.1- Expressões de paralelismo 54
4.6.2 - Comunicação e sincronismo entre processos 58
4.6.2.1 - Passagem de mensagem 59
4.6.2.1.1 - Envio de mensagem ponto a ponto 61
4.6.2.1.2 - Envio de mensagens um para vários 61
4.6.2.1.3 - Chamada remota de procedimento 62
4.6.2.2 - Dados Compartilhados 63
4.6.2.2.1 - Memória compartilhada em ambiente pseudo paralelo 64
4.6.2.2.2 - Memória compartilhada em ambiente paralelo real 71
5 - O MÉTODO DESENVOLVIDO 75
5.1 - Definição formal do problema 75
5.2 - Modelagem do problema 76
5.3 - Abordagens Existentes 80
5.4 - Uma nova abordagem 82
5.4.1 - Introdução 82
5.4.2 - Descrição do método 83
5.4.3 - Estrutura de vizinhança 85
5.4.4 - O operador genético 86
5.4.5 - O controle de temperatura 88
5.5 - A versão distribuída do método proposto 90
5.5.1 - Introdução 90

5.5.2 - O modelo de paralelização 91
5.6 - Análise dos resultados computacionais 94
6 - CONSIDERAÇÕES FINAIS 99
6.1 - Conclusões 99
6.2 - Propostas de novas pesquisas 101
REFERÊNCIAS BIBLIOGRÁFICAS 103

LISTA DE TABELAS

Tabela 2.1 - Tempos de processamento e roteamentos para um problema de programação de 3 *jobs* e 3 máquinas 11
Tabela 5.1 - Resultados dos problemas *benchmarks* de Fisher e Thompson (1963) 95
Tabela 5.2 - Resultados de problemas *benchmarks* propostos por Taillard (1993) 96
Tabela 5.3 - Resultados dos problemas *benchmarks* de Fisher e Thompson (1963) 98

LISTA DE FIGURAS

Figura 2.1 - Um programa de produção "ótimo" para o problema definido na tabela 2.1 12
Figura 3.1 - Conversão do valor da aptidão individual para o valor de expectativa 23
Figura 3.2 - Dois exemplos de *crossover* de um ponto 25
Figura 3.3 - Um exemplo da operação de mutação 26
Figura 4.1 - Exemplo de multicomputador baseado em barramento 41
Figura 4.2 - Grafo completo 42
Figura 4.3 - Topologia do tipo grelha 43
Figura 4.4 - Exemplos de hipercubos 44
Figura 4.5 - Exemplo de *crossbar* 45
Figura 4.6 - Trechos de dois processos paralelos 64
Figura 4.7 - Exemplo de exclusão mútua 65
Figura 4.8 - Exemplo de utilização de semáforos 67
Figura 4.9 - Exemplo de monitor 70
Figura 4.10 - Exemplo da utilização de um monitor 71
Figura 5.1 - Exemplo de representação de um problema *job shop*, com três *jobs* e quatro máquinas, através de um grafo disjuntivo 77
Figura 5.2 - Um esboço do algoritmo SA 84
Figura 5.3 - Ilustração de um processo de reversão do arco (v, w) 86
Figura 5.4 - Um esboço do algoritmo proposto 90
Figura 5.5 - Um exemplo de um programa paralelo 93

Capítulo 1

Introdução

Em todo ambiente industrial a produtividade está diretamente relacionada com o controle e o gerenciamento da produção. A competitividade e a dinamicidade do mercado obrigam as empresas a se fazerem cada vez mais ágeis em suas adaptações às constantes mudanças das condições mercadológicas impostas. Dentre os principais fatores que compõem o controle e o gerenciamento da produção encontra-se a programação da produção. Segundo Smith et al. (1986), a complexidade computacional relacionada com a construção e a manutenção da programação da produção de uma empresa, constitui o maior obstáculo na busca de um bom desempenho.

A programação da produção trata com a alocação eficiente de recursos na fabricação de bens. Os problemas da programação surgem sempre que um conjunto comum de recursos (mão de obra, matéria prima, equipamentos, etc) deve ser compartilhado, na fabricação de uma variedade de produtos, durante um mesmo período de tempo. A tarefa da programação da produção pode então ser definida como sendo o planejamento temporal da execução de um dado conjunto de ordens, onde o processamento de cada ordem corresponde à fabricação de um determinado produto, realizada através de uma série de operações, em um conjunto de recursos disponíveis, em uma seqüência predefinida e sob várias restrições adicionais. Como resultado tem-se um programa (ou uma escala) de produção que especifica as atribuições temporais das operações das ordens nos recursos a serem utilizados, de forma que restrições de produção sejam satisfeitas e que os custos de produção sejam minimizados.

O problema geral da programação da produção encontra-se entre os mais difíceis problemas combinatoriais conhecidos, pertencendo à classe dos problemas NP-*hard* Goyal et al. (1988) e King (1980). Como um agravante, no tratamento de problemas de escalonamento em um ambiente de produção real, os atributos que devem ser considerados e os requisitos impostos pelos numerosos detalhes do domínio da particular aplicação, isto é, máquinas alternativas, prazos de entrega, restrições de custos, disponibilidade de recursos, custos de *setup*, etc, tornam a programação da produção uma tarefa ainda mais árdua. Dessa forma, atualmente, a tarefa crucial da programação da produção tem sido a busca de programas que forneçam soluções tão próximas quanto possível das soluções ótimas.

Tradicionalmente, os problemas de programação industrial eram investigados, principalmente, em engenharia de produção, na área de pesquisa operacional. Hoje o interesse por esse assunto, conduzido por uma variedade de considerações teóricas e pragmáticas, atinge diversas áreas. A crescente competitividade do mercado internacional de bens manufaturados é um dos primeiros fatores responsáveis pelo fomento à pesquisa e desenvolvimento da programação da produção industrial. Essa competitividade é também responsável pelo surgimento de novos e sofisticados sistemas de manufatura, motivados pelo acentuado e sustentável declínio dos custos dos computadores industriais e robôs. Esses novos e notáveis sistemas de manufatura automatizados, ou seja, sistemas de manufatura integrados por computadores (CIM) vêm gerando, por sua vez, uma gama imensa de novos problemas operacionais, aumentando o volume e a diversidade da pesquisa na programação da produção. Como conseqüência, esse assunto se alastrou, passando a ser de grande interesse também em áreas dedicadas a teoria de controle, inteligência artificial, simulação de sistemas, sistemas de larga escala e outros ramos da engenharia e da ciência da computação Groover (1987) e Ranky (1986).

Atualmente, observa-se um aumento considerado no interesse por abordagens baseadas em técnicas de otimização de busca local para o tratamento de problemas da programação da produção. São métodos aproximados que normalmente utilizam como base os algoritmos, *tabu search*, genético e *simulated annealing*.

Esses algoritmos têm sido intensivamente pesquisados em soluções de problemas gerais de otimização e, especialmente, nos combinatoriais. O extraordinário sucesso atualmente alcançado por esses algoritmos, principalmente com relação aos dois últimos, é conseqüência de diversos fatores, dentre eles: a utilização de mecanismos de otimização modelados diretamente da natureza, aplicabilidade geral, flexibilidade nas adaptações às restrições específicas em casos reais, a excelente relação entre qualidade e facilidade de implementação e tempo de processamento.

O notável avanço tecnológico nas áreas de sistemas de computação (*hardware* e *software*) vem propiciando a construção de máquinas com capacidades de processamento cada vez maiores, graças ao grande número de processadores e imensa capacidade de memória com que essas máquinas paralelas vem sendo dotadas.

Como não poderia deixar de ser, o paralelismo tem potencialmente importante conseqüência para a área de otimização combinatorial. A utilização de máquinas paralelas torna possível determinar soluções em menor tempo e aumentar o tamanho e a complexidade dos problemas dessa classe que podem ser tratados.

1.1 Objetivo do Trabalho

Nesse trabalho, o interesse fundamental está centrado no desenvolvimento de uma nova abordagem, com implementações seqüencial e paralela, que possa servir como plataforma na resolução de problemas da programação industrial, tendo como base os algoritmos genético e *simulated annealing*.

1.2 Organização

Este trabalho é composto de seis capítulos organizados da forma seguinte. O capítulo 2 caracteriza o contexto geral no qual o trabalho desenvolvido está inserido, apresentando uma abordagem teórica e conceitual do problema da programação da produção, bem como são formuladas as premissas que fundamentam o trabalho.

No capítulo 3 é introduzido uma fundamentação básica conceitual de algoritmo genético e *simulated annealing*, onde os elementos básicos desses dois métodos são revistos.

O capítulo 4 está dedicado ao fornecimento de uma visão geral do que se constitui um sistema distribuído, bem como em descrever e comparar os métodos e linguagens que podem ser utilizados para a programação de sistemas distribuídos. Também discute os diferentes tipos de sistemas computacionais distribuídos existentes, os tipos de aplicações para os quais são indicados, e os suportes de programação requeridos para a implementação dessas aplicações.

No capítulo 5 é apresentado o modelo matemático que é utilizado no tratamento do problema e a forma como os dois métodos combinados constituíram o modelo proposto. Também faz parte desse capítulo uma análise comparativa entre os resultados computacionais obtidos com o método proposto e outros encontrados na literatura.

O trabalho completa-se com o capítulo 6 onde são apresentadas as considerações finais e propostas de novas investigações no assunto.

CAPÍTULO 2

Programação da Produção

Neste capítulo é apresentada uma visão geral do contexto no qual o presente trabalho se encontra inserido. Inicialmente a problemática geral, alguns conceitos fundamentais e uma visão geral de produtividade industrial são apresentados. Em seguida é apontada, dentre as diversas abordagens dadas ao problema da programação da produção industrial, aquela que será utilizada nesse trabalho.

2.1 Conceituação Básica

A programação da produção compõe o nível mais baixo na hierarquia de um sistema de planejamento da produção Gelders et al.(1981). No primeiro nível dessa hierarquia é determinado, com base nas decisões agregadas sobre produção e capacidade, o programa de produção. O resultado é chamado de programa mestre. Uma vez fixado o programa mestre, atinge-se o segundo nível da hierarquia, onde são determinadas as quantidades a serem produzidas (ou compradas) dos diferentes componentes. Definidas as quantidades bem como as datas de entrega dos diferentes componentes, alcança-se o terceiro e último nível, onde os programas de produção e as alocações dos recursos necessários são elaborados. O objetivo de uma programação é, portanto, determinar uma maneira adequada de se atribuir e seqüenciar a utilização desses recursos compartilhados de forma que restrições de produção possam ser satisfeitas e os custos de produção sejam minimizados.

Na sua forma geral, pode se estabelecer que o problema da programação da produção envolve um conjunto de *jobs* a ser processado, onde cada *job* compreende um conjunto de operações a ser executado. As operações reqüerem máquinas e recursos humanos e materiais e devem ser executadas de acordo com alguma sequência tecnológica viável. Os programas são severamente influenciados por uma série de fatores tais como prioridade de *jobs*, prazos de entrega, restrições de custos, níveis de produção, restrições quanto aos tamanhos dos lotes, disponibilidade e capacidade de máquinas, precedências de operações, requerimentos de recursos e disponibilidade de recursos. O programa gerado seleciona uma seqüência adequada de operações - o roteamento do processo de produção - a qual resultará na conclusão de todos os *jobs* do conjunto, no menor tempo possível.

As decisões tomadas no nível da programação de produção são caracterizadas pelo fato de que individualmente podem não apresentar importância considerável. O conjunto de decisões, tomado ao longo de um certo período de tempo nesse nível, entretanto, tem uma enorme influência nos planos elaborados nos níveis superiores da hierarquia, podendo comprometer severamente o desempenho de um dado ambiente de programação. Uma programação de produção ruim pode conduzir facilmente à utilização não otimizada de recursos e materiais, atrasos nas datas de entregas, baixa qualidade de produtos, baixa lucratividade, etc.

Pelo menos três razões aparentes têm levado o problema da programação da produção à uma intensa e contínua exploração. A primeira deve-se à importância prática que as regras de uma programação apresenta. Muito embora a terminologia conduza a ambientes de produção industrial, o problema da programação surge em diversas áreas. Por exemplo, no escalonamento de processadores em um sistema de computação distribuído, na determinação da melhor rota entre dois nós em uma rede de computadores, etc. A segunda, por se tratar de um problema combinatorial de extrema dificuldade, considerado NP-completo segundo a nomenclatura de análise de algoritmos, se torna assim um desafio permanente aos pesquisadores, Goyal et al. (1988) e King (1980). A terceira razão é indireta, as estratégias propostas para os problemas da

programação são relativamente gerais e assim têm inspirado importantes métodos para a resolução de outros problemas combinatoriais.

2.2 Classificação dos Problemas de Programação da Produção

O esquema de classificação aqui utilizado foi escolhido dentre vários esquemas propostos para a categorização dos problemas da produção. A escolha deve-se ao fato de que com a vasta abrangência por ele apresentada, se fez possível englobar as características gerais dos aspectos teóricos e práticos dos problemas. Esse esquema propõe três dimensões para a classificação dos problemas de programação da produção Graves (1981):

1) Geração de pedidos;
2) Complexidade do processo produtivo e
3) Critério de programação.

2.2.1 Geração de Pedidos

A primeira dimensão, geração de pedidos, refere-se à origem dos pedidos que podem ser gerados diretamente pelas ordens de compra dos produtos pelos clientes, ou, indiretamente, por decisões de reposição de estoques. Essa distinção, que é referida como *open shop* e *closed shop* respectivamente, é bastante importante para o problema da programação da produção, a qual sofre forte influência, dependendo dessa origem. No caso de *open shop*, todas as ordens de produção são geradas pelos pedidos de compras e nenhum estoque é mantido (produz diretamente para vendas). No caso de *closed shop* todos os pedidos de compras são atendidos pelo estoque e as tarefas de produção são geralmente conseqüências de decisões tomadas na reposição de estoques (produz para estoques e não diretamente para vendas). A programação da produção, na sua forma mais simples, para o caso de *open shop*, constitui-se em um problema de seqüenciamento, no qual as ordens são seqüenciadas em cada recurso. No caso de *closed shop*, a

programação da produção se envolve não somente com decisões de seqüenciamento, mas também com questões relacionadas com o dimensionamento de lotes e com as políticas de reposição de estoques. Um ambiente real de produção industrial, raramente, se caracteriza como puramente *open shop* ou *closed shop*. Ele pode, fundamentalmente, apresentar uma dessas características.

2.2.2 Complexidade do Processamento

A segunda dimensão, complexidade do processamento, está relacionada basicamente com o número de etapas de processamento associadas à produção de cada tarefa ou item. Uma subdivisão comum dessa dimensão é:

- um estágio, um processador;
- um estágio, processadores paralelos;
- múltiplos estágios, *flow shop* e
- múltiplos estágios, *job shop*

Um estágio, um processador é a forma mais simples do problema. Todas as tarefas requerem um único passo no processamento, que deve ser executado em único processador (máquina ou operário).

Um estágio, processadores paralelos é similar ao problema anterior exceto que cada tarefa requer um único passo de processamento, que pode ser executado em qualquer um dos processadores paralelos existentes.

Nos casos de *múltiplos estágios* cada tarefa requer processamento em um conjunto de processadores distintos, onde tipicamente existe uma severa ordem de precedência operacional a ser obedecida nas etapas de processamento de cada tarefa. As soluções viáveis são aquelas que satisfazem as relações de precedência. Um ambiente de produção é definido como *flow shop* quando todas as tarefas devem ser processadas em um mesmo conjunto de processadores e na mesma seqüência de processamento, existindo portanto, um único sentido no fluxo do processamento. Define-se um ambiente de

produção como sendo do tipo *job shop* quando não há restrições nos passos de processamento para cada tarefa, não existindo, nesse caso, um único fluxo. Pode existir, assim, rotas alternativas de processamento de uma tarefa. Este último tipo corresponde a um problema da programação da produção na sua forma mais geral, dentro dessa classificação.

2.2.3 Critério de Escalonamento

A terceira e última dimensão dentro da classificação adotada, critério de escalonamento, trata com as medidas através das quais a programação da produção é avaliada. Duas classes abrangentes de critérios de avaliação são consideradas: a classe referente ao custo de programação e a classe referente ao desempenho da programação. Os custos de programação de um programa de produção particular compreendem os custos fixos associados com *setup*, estocagem de produtos e materiais, custos referentes a atrasos de entregas, etc. Também são incluídos os custos referentes à geração do sistema do próprio programa, bem como a manutenção de tal sistema. Já o desempenho de um programa de produção pode ser avaliado por várias maneiras. Medidas comuns são os níveis de utilização dos recursos produtivos, percentagens de tarefas concluídas com atraso, tempo médio de processamento de tarefas ou itens, tempo total de processamento das tarefas, tempo médio ou máximo de atraso, etc. Em grande parte dos ambientes de produção, a avaliação da programação da produção se faz através da combinação dessas duas classes. Entretanto, a maioria das referências bibliográficas teóricas no assunto tratam o problema com um único critério.

Duas outras dimensões, que poderiam ser aqui incluídas, são a natureza da especificação dos pedidos e o ambiente de programação. Dependendo da maneira como os pedidos são gerados, suas especificações podem ser ditas determinísticas ou estocásticas. Por exemplo, para um ambiente *open shop*, o tempo de processamento para cada passo da execução de cada tarefa pode ser conhecido ou ser uma variável aleatória com uma distribuição de probabilidade específica. Da mesma forma, para um ambiente

closed shop, o processo de demanda dos clientes, o qual dirige a decisão da reposição de estoque, pode ser assumido como sendo estocástico ou determinístico. Quanto ao ambiente de programação, uma distinção comum é feita entre ambiente estático e dinâmico. Em um ambiente estático, o problema da programação da produção é definido considerando um conjunto finito de pedidos completamente especificados; nenhum pedido será adicionado a esse conjunto e nenhuma alteração nas especificações desses pedidos será realizada. Em contraste, em um ambiente dinâmico, define-se o problema da programação da produção em relação não somente ao conjunto dos pedidos conhecidos, mas também considerando pedidos adicionais e especificações geradas em tempos futuros. Para a maioria dos ambientes reais de produção industrial, o problema de programação é estocástico e dinâmico. A grande maioria dos modelos construídos para os problemas da programação, no entanto, é inerentemente determinístico e estático.

2.3 A Classe do Problema Tratado

Uma vez classificados os problemas da programação da produção nas suas formas genéricas, será então apresentado, especificamente, a classe do problema, conhecida como *job shop*, que se pretende tratar nesse trabalho. Para tanto, recorreu-se às definições e premissas existentes na teoria clássica da programação.

2.3.1 Definições e Premissas

Descreve-se, a seguir, um problema da programação da produção do tipo *job shop*, em sua forma básica.

Um conjunto de n *jobs* $J = \{ J_1, J_2, \ldots, J_n \}$ deve ser processado em um conjunto de m máquinas $M = \{ M_1, M_2, \ldots, M_m \}$ disponíveis. Cada *job* possui uma ordem de execução específica entre as máquinas, ou seja, um *job* é composto de uma lista ordenada de operações, cada uma das quais definida pela máquina requerida e pelo tempo de processamento na mesma. As restrições que devem ser respeitadas são:

•operações não podem ser interrompidas e cada máquina pode processar apenas um *job* de cada vez;

•cada *job* pode estar sendo processado em uma única máquina de cada vez e

•não existe restrição de precedência entre operações de diferentes *jobs*.

Uma vez que as seqüências de máquinas de cada *job* são fixas, o problema a ser resolvido consiste em determinar as seqüências dos *jobs* em cada máquina, de forma que o tempo de execução transcorrido, desde o início do primeiro *job* até o término do último, seja mínimo. Essa medida de qualidade de programa, conhecida por *makespan*, não é a única existente, porém é o critério mais simples e o mais largamente utilizado.

Será apresentado, a seguir, um exemplo ilustrativo de um problema de programação do tipo *job-shop*. A tabela 2.1 fornece os tempos de processamento e os roteamentos das operações nas máquinas para três *jobs* que deverão ser processados em três máquinas Rodammer, White (1988).

Tabela 2.1 Tempos de processamento e roteamentos para um problema de programação de 3 *jobs* e 3 máquinas

Job	1ª maq	2ª maq	3ª maq
1	1 / M1	8 / M2	4 / M3
2	6 / M2	5 / M1	3 / M3
3	4 / M1	7 / M3	9 / M2

A figura 2.1 é o gráfico de Gantt que representa uma das soluções para o problema expresso na tabela 2.1. É importante observar que a ordem requerida das operações dentro de cada *job* (a seqüência tecnológica) foi preservada na solução apresentada. Por exemplo, o *job* 2 foi processado nas máquinas 2, 1 e 3, respectivamente, como especificado na tabela 2.1. A ordenação das operações em cada máquina foi selecionada de forma que o objetivo desejado fosse alcançado, por exemplo, a máquina 3 executou os *jobs* 3, 2 e 1, respectivamente.

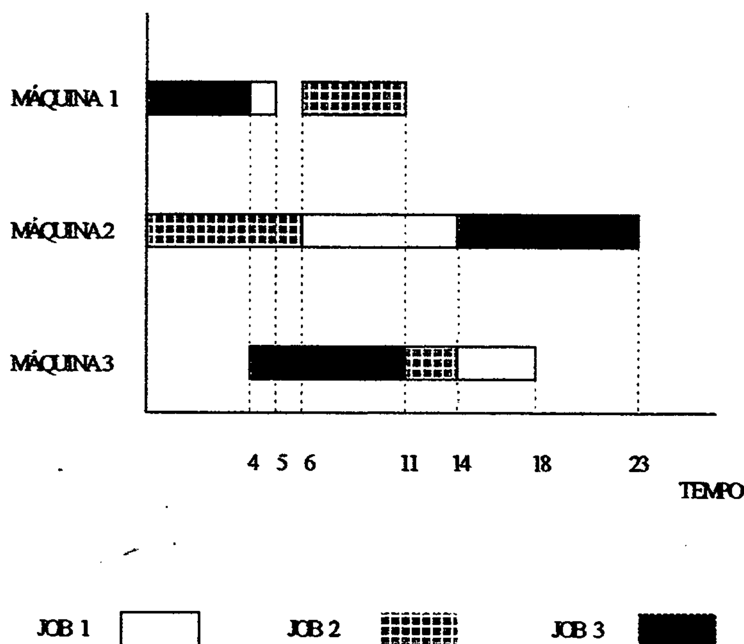


Figura 2.1 Um programa de produção " ótimo" para o problema definido na na tabela 2.1.

Sob as suposições, usualmente consideradas, de que todos os *jobs* estão prontos para serem processados no início do processamento e de que todas as operações requerem o uso exclusivo de uma máquina que não pode ser compartilhada, o programa apresentado na figura 2.1, representa uma seqüência de operações que minimiza o tempo total requerido para completar o processamento de todos os 3 *jobs*.

2.4 O método de Resolução Utilizado

Um grande número de abordagens diferentes empregadas nos problemas da programação, nos mais diversos ambientes reais de produção industrial, está disponível e muitos outros métodos de resolução certamente ainda deverão ser propostos. Em Rodammer e White (1988) são apresentados sete paradigmas de programação da produção, cujas concepções básicas se diferem radicalmente em função, especialmente, das diferentes maneiras como são considerados e manejados os principais atributos

constantemente presentes nos ambientes reais de programação. É muito improvável que uma única técnica de programação venha contemplar, em toda sua riqueza de detalhes, a todos esses complexos atributos, nos diferentes problemas de programação industrial. Provavelmente uma ferramenta com tal pretensão se inviabilizaria pela sua própria complexidade operacional. Os métodos de programação que efetivamente são utilizados tendem a capturar somente as características mais importantes do particular ambiente em que está sendo empregado, ignorando os menos importantes e tratando aqueles atributos considerados de relevância intermediárias de forma agregada. A grande maioria dos paradigmas de programação conhecidos considera apenas um subconjunto desses atributos, tornando a sua aplicabilidade inteiramente condicionada à forma como um problema de produção possa ser estruturado.

O conjunto de paradigmas de programação da produção estende-se desde o primitivo método manual, onde a experiência humana adquirida pela prática constitui o seu principal componente, até aos sofisticados e complexos sistemas especialistas, que tipicamente tratam o problema da programação da produção como a determinação e a contemplação do grande e variado número de restrições encontrado no domínio da programação, fazendo uso de técnicas avançadas de representação de conhecimento e de regras heurísticas na busca de boas soluções Bruno (1986), Han e Dejax (1991), He et. al. (1993) e Smith (1986). Na extensão desse conjunto de paradigmas são encontrados métodos tais como: otimização estocástica, simulação discreta e teoria de controle. A otimização estocástica utiliza a teoria de filas, teoria da confiabilidade, técnicas de dimensionamento de lotes, teoria de controle de estoque, etc, Jackson (1963). Nos método de simulação de eventos discretos são criados modelos que avaliam numericamente um certo programa de produção, dentro de um período de tempo determinado, constituindo um ambiente de testes, principalmente, para a programação heurística e regras de despachos Law (1986). Métodos que utilizam a teoria de controle consideram que um sistema tem seu estado corrente definido por parâmetros como nível de estoque, estados das máquinas, recursos disponíveis, etc e como possível seqüência de entrada, a qual atua no sistema produzindo seqüência de saída e mudança de estado, parâmetros como ordens de produção, chegada de matéria prima, distúrbios

programados, etc, ver Gershwin et al. (1986). A abordagem que será utilizada nesse trabalho, referenciada por teoria da programação da produção, trata o problema da programação da produção industrial sob o enfoque estritamente teórico. Ela vem sendo pesquisada extensivamente há décadas e pode ser, fundamentalmente, dividida em dois grandes grupos: métodos ótimos e métodos aproximados. O primeiro grupo, desenvolvidos principalmente em Pesquisa Operacional, também conhecido como grupo dos métodos exatos ou analíticos, visto que, normalmente recorrem à formulação matemática ou procedimentos analíticos rigorosos, trabalha na busca de soluções ótimas para o problema. No segundo grupo, onde se inclui o método desenvolvido neste trabalho, encontram-se os métodos que buscam soluções próximas às ótimas em tempos computacionais aceitáveis (são, na realidade, métodos aproximados que se traduzem em formas rápidas de se conseguir soluções boas e viáveis). Foram propostos como forma alternativa de resolução dos problemas de programação uma vez que, devido a grande complexidade computacional normalmente apresentada pelos métodos ótimos, suas utilizações práticas acabam inviabilizadas. Atualmente, em ambientes industriais reais, trabalhos em problemas de programação de máquinas demostram claramente a necessidade de métodos aproximados. A abordagem teórica da programação da produção, apesar de ter seus métodos com base em um formalismo simplificado do problema da programação da produção encontrado na prática, possui sua importância particular, uma vez que, captando a complexidade computacional fundamental do problema central do seqüenciamento de *jobs* em máquinas, de maneira genérica, pode definitivamente constituir a base de muitas ferramentas comercialmente viáveis. Um importante trabalho cujo objetivo foi aproximar as abordagens clássica (teórica) e a real (prática) do problema da programação da produção do tipo *job shop*, é encontrado em Cândido (1997).

No capítulo 5 é detalhada a maneira como foram combinados os algoritmos genético e *simulated annealing*, constituindo uma nova abordagem, na resolução de problemas de programação da produção, tendo como base a abordagem teórica aqui aludida.

CAPÍTULO 3

Os Algoritmos Genético e *Simulated Annealing*

Este capítulo encontra-se dividido em duas partes. A primeira aborda, inicialmente, o processo da evolução biológica natural que constituiu a inspiração inicial do algoritmo genético. Introduz, então, o algoritmo através da apresentação de sua estrutura básica e da discussão de seus principais elementos. Finaliza com uma apresentação formal de sua potencialidade, através de uma análise do chamado teorema fundamental dos algoritmos genéticos.

A segunda parte, seguindo a mesma ordem de apresentação do algoritmo genético, aborda o método conhecido por *simulated annealing*, apresentando inicialmente o modelo original da física, 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 ao Algoritmo Genético

O professor John Holland da Universidade de Michigan (U.S.A), em suas explorações dos processos adaptativos de sistemas naturais e suas possíveis aplicabilidades em projetos de *softwares* de sistemas artificiais, no final da década de 1970, conseguiu incorporar importantíssimas características da evolução natural a um algoritmo, Holland (1993). Para que se consiga entender, em sua plenitude, esse extraordinário algoritmo que o autor resolveu chamar de algoritmo genético, é

conveniente que se conheça um pouco desses processos biológicos observados na evolução natural.

O cientista inglês Charles Darwin estudando as espécies e suas evoluções, coletou durante anos um volumoso material que demonstrou, principalmente, a existência de inúmeras variações em cada espécie, Darwin (1953). Seus estudos, aliados às pesquisas de outros cientistas no assunto, tornaram evidente que as espécies realmente se modificam. Um dos principais pontos dos estudos de Darwin (1953) foi sem dúvida o aspecto das variações apresentadas entre indivíduos da mesma espécie. Através de estudos de pombos, por exemplo, ele verificou a enorme variedade que se obtém cruzando uma mesma espécie. Segundo Darwin (1953), novas variedades são produzidas por meio da seleção natural. As variações mais favoráveis de cada espécie conseguem sobreviver com mais facilidade, uma vez que têm mais chance de se reproduzirem.

Muitas experiências comprovam que a seleção natural é um fato incontestável, podendo não ser o único mecanismo evolutivo, porém sem dúvida um dos fatores principais do processo. Uma das principais constatações dessa evolução foi alcançada através de uma experiência realizada em laboratório, onde uma placa contendo cem milhões de bactérias foi posta em contato com uma dose de penicilina insuficiente para o combate total das mesmas. Observou-se que as dez bactérias que haviam sobrevivido se reproduziram na própria placa, e que essa nova geração sobreviveram ao meio, mesmo com a presença da penicilina. Dobrou-se então a quantidade do antibiótico e observou-se que quase todas as bactérias morreram. Novamente as poucas bactérias que sobreviveram a nova dosagem passaram a se multiplicar, sendo novamente expostas à dosagens cada vez mais altas. Esse processo se repetiu por cinco gerações e ao final da experiência, apresentou-se uma linhagem de bactérias resistentes a uma dose de penicilina duas mil e quinhentas vezes maior do que a inicial, utilizada na primeira cultura. É importante observar que as bactérias não desenvolveram individualmente resistência à penicilina, eram descendentes das poucas bactérias que herdaram uma característica favorável à sua sobrevivência naquele meio, havendo, portanto, um processo biológico de adaptação ao meio.

Embora os mecanismos que conduzem a evolução natural não estejam ainda plenamente compreendidos, algumas de suas importantes características são conhecidas. A evolução dos seres vivos se processa nos cromossomos (dispositivos orgânicos onde as estruturas desses seres são codificadas). Parte da criação de um ser vivo é realizada através de um processo de decodificação de cromossomos. Muito embora, os processos específicos de codificação e decodificação de cromossomos também não estejam ainda totalmente esclarecidos, existem algumas características gerais na teoria desse assunto que estão plenamente consolidadas:

- a evolução é um processo que se realiza nos cromossomos e não nos seres que os mesmos codificam;
- a seleção natural é a ligação entre os cromossomos e o desempenho de suas estruturas decodificadas. Os processos da seleção natural determinam que aqueles cromossomos bem sucedidos devem se reproduzir mais freqüentemente do que os mal sucedidos;
- é no processo de reprodução que a evolução se realiza. Através de mutação (alterações aleatórias de genes) o cromossomo de um ser descendente podem ser diferente do cromossomo de seu gerador. Através do processo de recombinações (*crossover*) dos cromossomos de dois seres geradores, é também possível que os cromossomos do ser descendente se tornem muito diferentes daqueles dos seus geradores e
- a evolução biológica não possui memória. A produção de um novo indivíduo depende apenas de uma combinação de genes da geração que o produz.

Holland (1993), convenceu-se de que essas características da evolução biológica poderiam ser incorporadas a um algoritmo para constituir um método extremamente simples de resolução de problemas complexos, imitando o processo natural, ou seja, através da evolução. Dessa forma, ele começou a trabalhar em algoritmos que manipulavam cadeias de dígitos binários, as quais chamou de cromossomos. Seus algoritmos realizavam evoluções simuladas em populações de tais cromossomos. Tal como nos processos biológicos, os algoritmos nada conheciam a respeito do problema que estava sendo resolvido. A única informação que lhes eram fornecida consistia da avaliação de cada cromossomo que os mesmos produziam. A utilidade dessa informação

era somente com respeito à condução do processo de seleção dos cromossomos que reproduziriam. Os cromossomos que melhor evoluíam apresentavam uma tendência a se reproduzirem mais freqüentemente do que aqueles cuja evolução era ruim.

Esse tipo de algoritmo que, através de um simples mecanismo de reprodução sobre um conjunto de soluções codificadas de um determinado problema, consegue produzir solução de boa qualidade, constitui hoje uma importante técnica de busca, empregada na resolução de uma grande variedade de problemas, nas mais diversificadas áreas. O algoritmo genético é bem mais robusto quando comparado aos principais métodos de busca tradicionais existentes. Sabe-se que esta característica para sistemas computacionais é um fator extremamente importante. Quanto mais robusto for um sistema computacional, maior será o seu período de vida útil e mais reduzido será o seu custo para reprojetar.

Para efeito comparativo, pode-se citar o método tradicional de busca baseado em cálculos. Este método, que tem sido intensamente utilizado e pesquisado, não é considerado um método robusto, uma vez que depende fortemente da existência de derivadas (valores de inclinação bem definidos). Mesmo que se permita aproximações numéricas para o cálculo dos valores das derivadas, ainda acaba sendo uma severa restrição. Por outro lado, esse método realiza a busca sempre na vizinhança do ponto corrente, constituindo, portanto, um método de busca local. Cabe salientar que surge aqui uma outra característica importantíssima dos algoritmos genéticos, a qual, aliás, foi um dos fatores que motivaram essa pesquisa: o seu paralelismo inerente ou intrínseco. No método baseado em cálculo, assim como em outros existentes, a busca pela melhor solução se processa sempre de um único ponto para outro, no espaço de decisão, através da aplicação de alguma regra de transição. Essa característica dos métodos ponto a ponto constitui um fator de risco, uma vez que, em um espaço com vários picos, é grande a probabilidade de um falso pico ser retornado como solução. Em contraste, o algoritmo genético trabalha simultaneamente sobre um rico banco de pontos (uma população de cromossomos), subindo vários picos em paralelo e reduzindo, dessa forma, a probabilidade de ser encontrado um falso pico.

A utilização do algoritmo genético na resolução de um determinado problema depende fortemente da realização de dois importantes passos iniciais:

- encontrar uma forma adequada de se representar soluções possíveis do problema em forma de cromossomo e
- determinar uma função de avaliação que forneça uma medida do valor (da importância) de cada cromossomo gerado, no contexto do problema.

A forma de representação das soluções possíveis em cromossomos varia de acordo com o problema. Nos trabalhos originais de Holland, essas representações eram feitas somente através de codificação utilizando cadeias de *bits,* o que é considerado por muitos como algoritmos genéticos puros. Hoje as representações são elaboradas em diversos tipos. A função de avaliação constitui-se no único elo de ligação entre o algoritmo genético e o problema a ser resolvido. Recebendo um cromossomo como entrada, essa função retorna um número ou uma lista de números que exprime uma medida do desempenho daquele cromossomo, no contexto do problema a ser resolvido. A função de avaliação, no algoritmo genético, desempenha o mesmo papel que o ambiente no processo de evolução natural. Assim como a interação de um indivíduo com seu meio ambiente fornece uma medida de sua aptidão, a interação de um cromossomo com uma função de avaliação também resulta em uma medida de aptidão, medida essa que o algoritmo genético utiliza na realização do processo de reprodução.

3.1.1 Descrição do Algoritmo Genético

O algoritmo genético pertence à uma das classes dos chamados algoritmos evolucionários. Ele se constitui em um método de busca de propósito geral, baseado em um modelo de evolução biológica natural. O seu objetivo é explorar o espaço de busca na determinação de melhores soluções, permitindo que os indivíduos na população evoluam com o tempo. Cada passo do algoritmo no tempo, na escala evolutiva, é chamado de geração. Partindo de uma geração inicial, normalmente criada aleatoriamente, o algoritmo

executa continuamente o seguinte laço principal, Davis (1991), Gauthier (1993), Goldberg (1989) e Tanese (1989):

1) avalia cada cromossomo da população através do cálculo de sua aptidão, utilizando a função de avaliação (etapa de avaliação);

2) seleciona os indivíduos que produzirão descendentes para a próxima geração, em função dos seus desempenhos apurados no passo anterior (etapa de seleção);

3) gera descendentes através da aplicação das operações de cruzamento, *crossover* e mutação sobre os cromossomos selecionados no passo anterior, criando a próxima geração (etapa de reprodução) e

4) se o critério de parada for satisfeito, termina e retorna o melhor cromossomo até então gerado, senão volta ao passo 1.

A seguir são descritos os passos acima apresentados, considerando que os parâmetros necessários na utilização do algoritmo são:

- tamanho da população;
- número máximo de gerações;
- probabilidade de ocorrência de *crossover* por par de geradores e
- Probabilidade de ocorrência de mutação por posição de gene.

3.1.1.1 O processo de Avaliação

O primeiro passo (avaliação) consiste simplesmente na aplicação de uma função de avaliação em cada um dos indivíduos da população corrente. Como visto anteriormente, essa função deve expressar a qualidade de cada indivíduo daquela população, no contexto do problema considerado. Essa função, extremamente importante, uma vez que constitui o único elo entre o algoritmo genético e aquele problema particular que está sendo resolvido, depende fortemente da forma como as soluções foram representadas.

A grande maioria das aplicações de algoritmos genéticos utilizam representações indiretas das soluções, ou seja, o algoritmo trabalha sobre uma população de soluções

codificadas. Desta forma, antes da avaliação de um cromossomo (antes da aplicação da função de avaliação), uma transição da codificação da solução para a solução real, necessita ser feita. Por outro lado, uma aplicação que empregue representação direta de solução, o valor que é passado para a função de avaliação é o próprio cromossomo. Neste caso, toda informação relevante ao particular problema que está sendo tratado, deve estar incluída na representação da solução.

3.1.1.2 O processo de Seleção

O segundo passo (seleção) efetua, com base nas aptidões individuais, a seleção dos indivíduos que procriarão para formar a próxima geração. Esta seleção é probabilística, pois um indivíduo com aptidão alta tem maior probabilidade de se tornar um gerador do que um indivíduo com aptidão baixa. São escolhidos, neste passo, tantos geradores quanto for o tamanho da população.

O processo inicia-se com a conversão de cada aptidão individual em uma expectativa que é o número esperado de descendentes que cada indivíduo poderá gerar.

A seguir, serão apresentadas duas maneiras para o cálculo dos valores das expectativas, em seguida, como estes valores são utilizados no processo efetivo da escolha dos geradores, com base nos valores das expectativas.

No algoritmo original de Holland (1993), essa expectativa individual era calculada através da divisão da aptidão individual pela média das aptidões de toda a população da geração corrente. Desta forma, a expectativa *e*, de um indivíduo *i*, é calculada por:

$$e_i = apt_i / mapt_t ,$$

onde apt_i é a aptidão do indivíduo *i*, e $mapt_t$ é a aptidão média da população no tempo *t*.

As expectativas assim produzidas têm as seguintes características: um indivíduo com aptidão acima da média terá expectativa maior do que *1*, enquanto que um indivíduo com aptidão abaixo da média terá uma expectativa menor do que *1*; a soma dos valores de todas as expectativas individuais será igual ao tamanho da população.

No entanto, pelo menos dois problemas podem ser apontados na utilização desse método simples do cálculo das expectativas. Primeiro, o método não deve manipular valores de aptidão negativos, uma vez que poderiam ser convertidos em valores de expectativa negativos, os quais não teriam qualquer sentido. Segundo, esse método pode apresentar efeitos indesejáveis em duas ocasiões durante o seu processamento. Logo no início, quando a população é aleatoriamente calculada e a média das aptidões é baixa, indivíduos aptos podem receber um número desordenado de descendentes, conduzindo a uma convergência prematura. Também no decorrer do processamento, esse método tem uma forte tendência em alocar a todos os indivíduos um descendente, mesmo para aqueles indivíduos com expectativa muito baixa, resultando em uma exploração ineficiente do espaço de busca e consequentemente em um retardo acentuado na convergência do método.

Um segundo método de conversão de valores de aptidão em números esperados de descendentes, referenciado em Goldberg (1989) como o método do truncamento sigma, é baseado na aptidão média de toda a população e no desvio padrão das aptidões sobre a população. Sua idéia básica é também muito simples: um indivíduo, cuja aptidão seja igual à média das aptidões da população, terá uma expectativa de produzir um descendente; um indivíduo, cuja aptidão seja um desvio padrão acima da média, terá uma expectativa de produzir um descendente e meio; um indivíduo, cuja aptidão seja dois desvios padrões acima da média, terá uma expectativa de produzir dois descendentes, e assim por diante. A figura 3.1 ilustra este método.

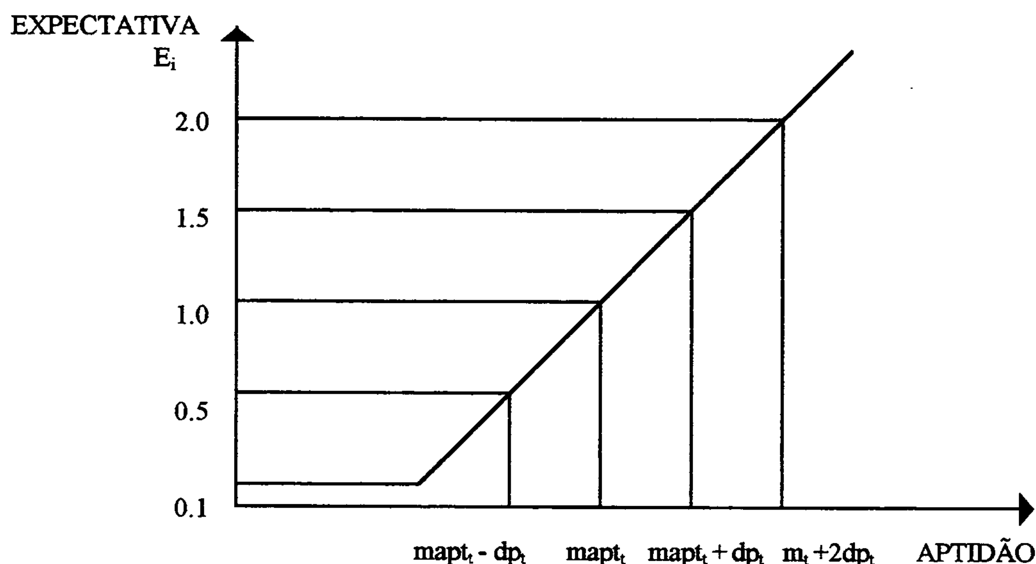


Figura 3.1 Conversão do valor da aptidão individual para o valor da expectativa.

Dessa forma, a expectativa *e* de um indivíduo *i* é calculada por:

$$e_i = (apt_i - mapt_t) / 2dp_t + 1 \qquad se \quad dp_t \neq 0$$

ou

$$e_i = 1 \qquad se \quad dp_t = 0$$

O apt_i é a aptidão do indivíduo *i* e $mapt_t$, e dp_t são, respectivamente, a média e o desvio padrão das aptidões no tempo (ou geração) *t*.

Caso o desvio padrão seja zero, então qualquer indivíduo na população terá o mesmo valor de aptidão. Neste caso, automaticamente esse método irá atribuir um valor de expectativa igual a *1* para todos os indivíduos da população. Se o valor da expectativa tornar-se negativo, ele será alterado para um valor pequeno, como por, exemplo, *0.1*, de forma que aqueles indivíduos com aptidões muito baixas, terão pequenas probabilidades de se reproduzirem. Nota-se, que com esse segundo método, a soma dos números esperados de descendentes para todos os indivíduos na população não necessariamente será igual ao tamanho da população, como ocorria no primeiro.

Após a apresentação desses dois métodos de conversão dos valores de aptidão para os valores de expectativa, será descrito, a seguir, uma maneira de se efetuar uma

última conversão, desta feita, dos valores esperados para os números definitivos de descendentes que cada indivíduo produzirá. O número de descendentes de um indivíduo não poderia ser o seu valor de expectativa, uma vez que o mesmo é um número real e, por exemplo, para um indivíduo que recebesse o valor de expectativa igual a *1,2* não faria sentido afirmar que o mesmo geraria um descendente e um quinto.

O número inteiro e definitivo de descendentes que cada indivíduo receberá, chamado de valor de descendente individual e calculado a partir do valor esperado, pode ser obtido através de uma técnica utilizada em diversos métodos, a qual será descrita a seguir.

Supondo que o valor esperado de cada indivíduo de uma geração seja representado em um círculo, ocupando uma fatia proporcional a seu número (a área total do círculo é igual a soma de todos os valores esperados) e imaginando, ainda, que o círculo, assim obtido, represente um alvo sobre o qual serão arremessados dardos, um descendente será, então, alocado ao indivíduo cuja fatia for atingida por um dardo arremessado sobre o círculo alvo. Quanto maior o tamanho de uma fatia (quanto maior o valor esperado), maior a probabilidade do dardo atingir esta fatia e, portanto, do indivíduo correspondente procriar. O número de dardos arremessados será igual ao tamanho da população.

O método de conversão do valor esperado em valor de descendente propriamente dito, conhecido como método de amostragem universal estocástico, proposto por Bark (1987), apresenta uma pequena variação do originalmente proposto. A diferença é que ao invés de um dardo ser arremessado para produzir um descendente e, então, selecionar o seu gerador individualmente, esse método seleciona todos os geradores simultaneamente através do lançamento conjunto de todos os dardos. O lançamento conjunto dos dardos se processa em intervalo de tamanho constante e igual ao quociente da área do círculo alvo pelo número de dardos a ser lançado (tamanho da população). A vantagem desse método, segundo Tanese (1989), é que os erros decorrentes do processo de amostragem são minimizados, especialmente para pequenas populações.

Encerrando o segundo passo principal do algoritmo genético, o qual descreve o processo de seleção dos cromossomos que reproduzirão a próxima geração, é importante

salientar que, apesar de sua aparência confusa (converte a aptidão de cada indivíduo para um valor esperado e, depois, converte este valor esperado para o valor de descendentes), este processo, de fácil implementação, vem tornar o algoritmo genético ainda mais robusto, uma vez que generaliza a função de avaliação, permitindo que a mesma retorne valores bem diversificados.

3.1.1.3 O processo de Reprodução

O terceiro passo (reprodução) é efetuado em duas etapas. Na primeira são feitas cópias dos indivíduos selecionados como geradores da próxima população. O número de cópias feitas de cada indivíduo gerador será igual ao seu número de descendentes individual, calculado no passo anterior. Na segunda etapa, essas cópias serão agrupadas, duas a duas, de forma aleatória e sem reposição, constituindo os pares de geradores. Cada par de indivíduos, através da aplicação dos operadores genéticos, produzirá dois descendentes para a geração seguinte. Os operadores genéticos básicos, neste passo, são reprodução, cruzamento (*crossover*) e mutação. Suas descrições serão vistas a seguir.

Através da operação de reprodução, cada par de geradores, escolhido no passo anterior, fará parte da próxima população automaticamente.

O operador *crossover*, atuando sobre um par de geradores (presumidamente indivíduos aptos), permuta alguns segmentos de suas estruturas, para formar dois novos indivíduos descendentes, conforme mostra a figura 3.2.

par 1
1 1 1 0 0 1 | 0 0 1 1 ⇒ filho 1: 1 1 1 0 0 1 1 0 1 0
0 0 0 1 1 0 | 1 0 1 0 ⇒ filho 2: 0 0 0 1 1 0 0 0 1 1

par 2
0 0 1 | 0 0 1 1 0 0 1 ⇒ filho 1: 0 0 1 1 1 0 1 1 0 0
0 1 1 | 1 1 0 1 1 0 0 ⇒ filho 2: 0 1 1 0 0 1 1 0 0 1

Figura 3.2 Dois exemplos de *crossover* de um ponto.

Nos exemplos apresentados na figura 3.2 as operações de *crossover* foram realizadas em um único ponto. No par1, após o sexto gene, contando da esquerda para a direita. No par2, após o terceiro gene. Esse tipo de operação, chamado de *crossover* de um ponto, utilizado originalmente por Holland (1993), ocorre quando partes dos geradores tomadas a partir de um ponto de corte, escolhido aleatoriamente, são permutadas. Existem outras formas de *crossover,* como por exemplo, o de dois pontos, no qual duas posições são escolhidas aleatoriamente e os materiais genéticos permutados entre os geradores serão os segmentos contidos entre essas posições.

Tradicionalmente, o número de operações de *crossover* sofridas por um par de geradores, é determinado por uma variável aleatória, cuja distribuição é a de Poisson, com média igual ao parâmetro fornecido (probabilidade de ocorrência de *crossover* por par de indivíduos). Portanto, se a taxa de *crossover* fornecida for igual a *1*, o número médio de operações *crossover* por par de geradores será *1*. Entretanto, é possível que alguns pares de geradores sofram a operação de *crossover* mais de uma vez. Para cada operação de *crossover,* a posição de corte é selecionada de maneira aleatória e uniformemente distribuída em relação a todas as posições possíveis.

Crossover é um componente extremamente importante do algoritmo genético, sendo considerado por muitos pesquisadores como parte vital do algoritmo. Seus efeitos são discutidos formalmente logo mais adiante, ainda neste capítulo.

Por outro lado, a operação de mutação é considerada de retaguarda. Tomando normalmente um descendente recém-criado como operando, essa operação altera uma pequena porção de seus genes, em alguma posição escolhida aleatoriamente. A taxa, através da qual, a operação de mutação é aplicada à uma posição de um indivíduo, é determinada pelo parâmetro fornecido: probabilidade de ocorrência de mutação por posição de genes. A figura 3.3 mostra um exemplo dessa operação.

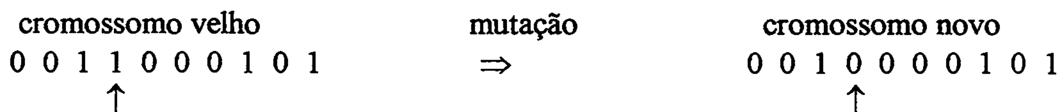


Figura 3.3 Um exemplo da operação de mutação

No exemplo anterior, figura 3.3, o gene que ocupa a quarta posição no cromossomo velho sofre uma mutação, passando de 1 para 0 no cromossomo novo.

Caso vários indivíduos em uma população contiverem os mesmos valores em uma dada posição, como examinado, a operação de *crossover* jamais conseguirá introduzir algum valor diferente naquela posição. Em tais situações, somente a mutação poderá conseguir introduzir novos valores.

3.1.2 O Teorema Fundamental do Algoritmo Genético

A primeira parte deste capítulo se encerra com a discussão do aspecto formal sobre o qual se encontra embasado o algoritmo genético. Será mostrado formalmente, através de uma análise do teorema de Holland (1993), que o efeito da ação das operações *crossover* e mutação, durante o processamento de um algoritmo genético, consiste na combinação contínua de segmentos de genes de boas soluções, provenientes de diversos cromossomos. Esses segmentos são conhecidos como blocos de construção ou simplesmente esquemas Holland (1993) e Tanese (1989). A hipótese fundamental feita pelo algoritmo genético é que indivíduos com alto grau de aptidão são portadores de bons esquemas e que esses bons esquemas podem ser utilizados na construção de indivíduos com aptidões ainda mais altas. O algoritmo genético constrói, portanto, uma nova geração através da exploração das informações contidas nos blocos de construção dos indivíduos da geração corrente.

Formalmente, assume-se que os pontos em um espaço de busca do algoritmo genético são representados por cadeias binárias de comprimento l, existindo assim, 2^l pontos no espaço. Seja A o conjunto formado por todos esses pontos. Sejam $B(t) \subseteq A$, a população na geração t e m o número de indivíduos em $B(t)$. Um indivíduo em $B(t)$ pode ser denotado por x_i, onde $1 \leq i \leq m$. Cada x_i pode ser considerado como um ponto amostral retirado do conjunto A. Na terminologia genética, x_i é um cromossomo composto de l genes. Cada gene possui uma posição, ou *locus*, no cromossomo e pode

tomar diferentes estados chamados *alleles*. Nos trabalhos originais de Holland, os possíveis *alleles* para cada *locus* eram "0" ou "1".

Um esquema é uma cadeia de *l bits*, tomada sobre o alfabeto {0, 1, #}, onde o símbolo "#" representa qualquer *bit* : 0 ou 1. Por exemplo, o esquema: 0 1 1 # # # # # # # , denota o conjunto de pontos cujos primeiros três *bits* são 0 1 1 e cujos 7 restantes são todas as combinações possíveis de 0's e 1's. Já o esquema: # 1 0 1 0 0 0 1 0 #, denota os seguintes pontos: 1 1 0 1 0 0 0 1 0 1, 1 1 0 1 0 0 0 1 0 0, 0 1 0 1 0 0 0 1 0 0 e 0 1 0 1 0 0 0 1 0 1. Uma instância de um esquema *s* é um elemento representado por *s*. Por exemplo, os 4 pontos acima mostrados são todas as possíveis instâncias do esquema: # 1 0 1 0 0 0 1 0 #.

Na primeira fase do processo de reprodução, como visto, são efetuadas as alocações dos descendentes e as cópias dos geradores. O algoritmo incrementa o número de cópias de cada indivíduo apto na população corrente, tantas cópias quanto for o número de seus descendentes. Será examinado o que acontece com um esquema *s* qualquer não considerando, inicialmente, os efeitos das operações de *crossover* e mutação. Seja $N(s, t)$ o número de instâncias de *s* na geração $B(t)$ e seja S o conjunto dos elementos x_i que apresentam uma instância de s, $i = 1, \dots, N(s, t)$. Na geração seguinte, $t+1$, após realizadas as devidas alocações de descendentes e feitos os números apropriados de cópias de cada gerador, o número de instâncias de *s* será a soma do número de cópias realizadas de todos os indivíduos x_i, que possuem instâncias de *s* em $B(t)$. Este número é igual à soma dos números de descendentes alocados à cada um desses indivíduos. Para cada x_i em $B(t)$, seja $apt(x_i)$ o seu valor de aptidão. O número esperado de descendentes para x_i, para a geração $t+1$, $E(x_i, t+1)$, é baseado na aptidão, $apt(x_i)$, relativa à média das aptidões da população, $mapt(t)$. Na forma básica do algoritmo,

$$E(x_i, t+1) = apt(x_i) / mapt(t) \qquad (3.1)$$

Assim, o número esperado de instâncias de *s* no tempo $t+1$ é:

$$N(s, t+1) = \sum_{x_i \in S} E(x_i, t+1) \qquad (3.2)$$

Substituindo (3.1) em (3.2), tem-se

$$N(s, t+1) = \sum_{x_i \in S} apt(x_i) / mapt(t) \quad (3.3)$$

Rearranjando (3.2), tem-se

$$N(s, t+1) = (1 / mapt(t)) \sum_{x_i \in S} apt(x_i) \quad (3.4)$$

Seja *mapt(s, t)* a média das aptidões observadas de *s* no período *t*, ou seja,

$$mapt(s, t) = \sum_{x_i \in S} apt(x_i) / N(s, t) \quad (3.5)$$

Substituindo a equação (3.5) em (3.4), obtém-se

$$N(s, t+1) = (mapt(s, t) / mapt(t)) N(s, t) \quad (3.6)$$

A equação (3.6) estabelece que o número esperado de instâncias do esquema *s* aumenta ou diminui (na fase de seleção e cópias) na razão direta da média das aptidões observadas com relação à média das aptidões da população inteira. Isso é verdade para todos os esquemas que possuam uma instância em *B(t)*. No entanto, deve-se notar que essa operação é baseada na aptidão observada do esquema, a qual pode não ter o mesmo valor de sua aptidão real, particularmente no caso de esquemas que possuam poucas instâncias em *B(t)*, pois neste caso, o valor observado da aptidão se torna maior devido ao pequeno valor de *N(s, t)*.

Será examinado, a seguir, o que acontece com os esquemas, considerando a aplicação das operações *crossover* e mutação. As questões de interesse são: onde essas operações quebrarão os esquemas e como essas operações afetarão os cálculos do número esperado das instâncias de esquemas de uma geração para outra.

Para tanto, outras definições fazem-se necessárias. As posições dos *bits* 0's ou 1's em um esquema *s* é chamado de locais de definição de *s* e o número de tais locais é chamado de ordem de *s*, denotado por *o(s)*. A distância entre o primeiro local de definição e o último, chamada tamanho de definição, é denotada por *d(s)*. Por exemplo

o esquema s = # # 0 # # # 1 0 # #, possui ordem 3 e tamanho de definição 5, ou seja, $o(s) = 3$ e $d(s) = 8 - 3 = 5$.

O operador *crossover*, pela recombinação de segmentos de indivíduos diferentes, empenha-se em criar descendentes que sejam instâncias de bons esquemas presentes nos dois geradores. No entanto, o *crossover* pode também, a partir de bons esquemas de geradores, produzir descendentes que não apresentem mais instâncias de quaisquer blocos de construção considerados bons. A chance dessa perturbação ocorrer, como será examinado a seguir, depende do número de ocorrências de *crossover* e do tamanho de definição $d(s)$ dos esquemas. Quanto maior um esquema, maior a probabilidade de ser perturbado.

Seja P_C a probabilidade de um *crossover* em cada par de geradores (equivalente ao parâmetro fornecido, isto é, taxa de *crossover*). Um esquema s, de tamanho l, será perturbado por uma operação *crossover*, somente se a posição de *crossover* selecionada recair entre o primeiro e o último local de definição de s. Como o número de locais possíveis é $(l-1)$ e o número de locais favoráveis é $d(s)$, a probabilidade de ocorrência de uma perturbação será igual a $P_C(d(s)/(l-1))$. Conseqüentemente, a probabilidade de uma operação *crossover* não perturbar o esquema s será igual a $(1 - P_C(d(s)/(l-1)))$.

Da mesma maneira, a probabilidade de uma operação de mutação vir a perturbar um esquema s pode ser calculada da seguinte forma. Seja P_M a probabilidade de mutação por local (equivalente ao parâmetro: taxa de mutação). Um esquema s será perturbado se uma mutação ocorrer em qualquer um dos locais de definição de s. Portanto, a probabilidade de que uma mutação não ocorra em local de definição de s é igual a $(1 - P_M)^{O(S)}$.

A equação (3.6), incluindo os efeitos causados pelas operações de *crossover* e mutação, pode ser finalmente escrita como:

$$N(s, t+1) \geq N(s, t)\ (mapt(s, t) / mapt(t))\ \{[(1-P_C)\ (d(s) / (l-1))]\ (1-P_M)^{O(S)}\} \quad (3.7)$$

O resultado, expresso pela inequação (3.7), é conhecido como teorema fundamental dos algoritmos genéticos e seu enunciado pode ser feito da seguinte forma:

o número esperado de instâncias de um esquema *s*, em uma geração *t*, aumentará na geração seguinte *(t+1)*, desde que satisfeitas as condições:

- o esquema *s* tenha um tamanho de definição, *d(s)*, pequeno;
- a média das aptidões observadas das instâncias de *s* seja maior do que a média das aptidões dos cromossomos da geração *t*.

Observa-se que a expressão (3.7) fornece um limite inferior do número esperado de instâncias de um esquema *s*, não considerando o fato de que *crossover* pode formar instâncias de *s* através de cruzamentos de indivíduos que não são instâncias de *s*. Também não leva em consideração que uma mutação pode produzir uma instância de *s* à partir de um indivíduo que não era instância de *s*. Finalmente, desconsidera o fato de que, se dois indivíduos são instâncias de um esquema *s*, então nunca algum *crossover* criará, à partir deles, descendentes que não sejam instâncias de *s*, independente da posição da operação.

O teorema mostra que a potencialidade do algoritmo genético consiste na sua habilidade de processar esquemas (muito mais esquemas do que o número de indivíduos em uma população). Considerando o fato de que cada indivíduo, x_i, é uma instância de 2^l esquemas diferentes: esquemas com posições de *bits* idênticas às posições de *bits* correspondentes de x_i ou tendo o valor "#". Em uma população de tamanho *M*, haverá, assim, instâncias de algum número de esquemas entre 2^l e $M2^l$.O número de esquemas com instâncias em uma população é, em geral, menor do que $M2^l$ devido à sobreposição de esquemas. Por exemplo, cada indivíduo é uma instância do esquema consistindo de todos os "#"s.

No processamento de *M* indivíduos na população, o algoritmo genético irá efetivamente processar um grande número de esquemas representados por esses indivíduos. Essa propriedade é conhecida como paralelismo implícito ou intrínseco Holland (1993).

3.2 Introdução ao Algoritmo *Simulated Annealing*

Esta seção inicia com uma breve apresentação do algoritmo *simulated annealing* na sua versão original, utilizada 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 à 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:

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

Biblioteca Universitária
UFSC 0.300.328-2

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 a 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 da mudança ocorrer.

O método computacional que imita esse processo de recozimento de um sólido é chamado de *simulated annealing*.

3.2.1 *Simulated Annealing* como um Método de Otimização

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), 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. Ou seja, 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, de um conjunto de soluções obtidas de execuções sucessivas, realizadas a partir de diferentes soluções iniciais.

Simulated annealing não utiliza essa estratégia. Esse método tenta evitar a convergência para um mínimo local, aceitando, as 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 δ 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

$$g(\delta, T) = e^{-\delta / T} \qquad (3.9)$$

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

$$g(\delta, T) > random(0, 1) \qquad (3.10)$$

A semelhança com o método original de simulação de recozimento na termodinâmica, como já aludido, é grande, pois o parâmetro δ, 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 $g(\delta, T)$ implica em:

- a probabilidade de aceite de uma nova solução é inversamente proporcional ao incremento de δ e
- quando T é alto, a maioria dos movimentos (de um estado para outro ou de uma solução para outra) é aceita, entretanto, a 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.

3.2.2 Considerações Teóricas

Matematicamente, o algoritmo *simulated annealing* pode ser modelado através da teoria de cadeias de Markov. Utilizando esse modelo, vários resultados importantes, tratando de condições suficientes para a convergência, têm surgido na literatura. A grande maioria desses trabalhos, entretanto, não levam em consideração o número de iterações necessárias para se atingir essa convergência. Uma vez que o tamanho do espaço de solução E cresce exponencialmente com o tamanho do problema, o tempo de execução de um algoritmo desse tipo pode alcançar níveis inviáveis. Um resultado muito importante é fornecido no trabalho publicado por Hajek (1988), no qual, não só as condições necessárias e suficientes para a convergência assintótica do algoritmo para um conjunto de soluções ótimas globais são fornecidas, mas também, o número de iterações necessária para que essa convergência possa ocorrer.

CAPÍTULO 4
Computação Paralela

Este capítulo procura abranger os fundamentos da computação paralela de forma prática e objetiva, norteado por dois aspectos fundamentais: fornecer o embasamento teórico básico para o capítulo cinco e preconizar a utilização desse tipo de computação em projetos de engenharia da produção, nas suas mais diversas áreas. Descreve, sob o ponto de vista do usuário, os principais e mais representativos modelos arquitetônicos dos sistemas distribuídos existentes e, dentro dessa mesma visão, estuda as características principais das linguagens de programação paralela disponíveis, suas classes de aplicações, os suportes por elas requeridos e como essas linguagens os atendem.

4.1 Motivação

A programação paralela pode ser considerada como sendo a atividade de se escrever programas computacionais compostos por múltiplos processos cooperantes, atuando no desempenho de uma determinada tarefa Andrews (1991). 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 na minimização do tempo de processamento, ou mesmo na busca de uma estruturação melhor e ou mais segura de um sistema.

No desenvolvimento de um sistema paralelo, após a especificação do problema a ser resolvido, deve-se decidir quantos processos podem compor o mesmo e como esses processos irão interagir. Essas decisões são diretamente afetadas pela natureza da aplicação e pelas características do *hardware* que se tem disponível.

Uma vez projetado o sistema, definidos os processos componentes e suas interações, surge um problema crucial: a garantia de que a comunicação entre os

processos seja adequadamente sincronizada. Os diversos ambientes disponíveis para o desenvolvimento de sistemas distribuídos e as diferentes maneiras como os processos de um programa paralelo podem ser criados e coordenados são os principais temas deste capítulo.

Tal como em outras áreas empíricas da ciência da computação, a história da computação paralela não poderia ser diferente, ela surgiu em decorrência de desenvolvimentos marcantes verificados em *hardware*. Dentre os principais precursores desse tipo de programação, destacam-se os sistemas operacionais. Com a importante independência dos controles de dispositivos, alcançada na década de 1960, o processador central de um computador deixou de interagir diretamente como os dispositivos de entrada e saída, mas através de interfaces especializadas. Desta forma, pôde-se constatar que a utilização do processador central poderia ser consideravelmente otimizada, utilizando-se os seus tempos ociosos, decorrentes da espera pelas realizações das operações independentes de entrada e saída. Esses intervalos de tempo ociosos poderiam ser, por exemplo, utilizados no processamento de outros programas. Alguns sistemas operacionais começaram, então, a ser projetados como uma coleção de processos, em um ambiente em que as tarefas de usuários conviveriam harmoniosamente com os processos gerentes de dispositivos e os demais processos do sistema operacional Silberschatz e Peterson (1994) e Tanenbaum (1992). Em sistemas dessa natureza, onde um único processador é capaz de processar mais de um programa simultaneamente, conhecidos como sistemas multiprogramados, os processos são executados em fatias de tempo, que o processador a eles dedica de forma intercalada.

Assim, o despertar da idéia de se construir sistemas computacionais como coleção de processos cooperantes e a evolução tecnológica fortemente verificada na área da microeletrônica foram os principais fatores responsáveis pelo surgimento de uma grande variedade de arquiteturas compostas por múltiplos processadores, conhecidos por sistemas multiprocessados.

4.2 Sistemas Computacionais Distribuídos

Não há um consenso absoluto na literatura com relação a definição de sistema computacional distribuído havendo, no entanto, dentre as muitas definições encontradas, pelo menos um ponto em comum: todas requerem a presença de múltiplos processadores. A considerável controvérsia reinante pode, portanto, estar relacionada com o grande número de diferentes modelos arquitetônicos de sistemas, com múltiplos processadores propostos e construídos, abrangendo um grande leque de opções em termos de objetivo do projeto a ser executado, do seu tamanho e desempenho. Em um sistema multiprocessado com memória compartilhada múltiplos processadores compartilham uma memória comum. Já em um sistema multicomputador, vários processadores, com memórias privadas, conhecidos como nós, são conectados através de um *hardware* especializado que envia mensagens entre eles, em alta velocidade. Em um sistema de rede de computadores, processadores são conectados através de uma rede de comunicação que pode ser local ou de longa distância. Muitos sistemas de comunicações híbridos também podem ser encontrados, como, por exemplo, uma rede de *workstations* multiprogramadas. Deve-se, também, incluir nesse conjunto de sistemas, os computadores construídos especialmente para aplicações numéricas de computação intensiva: os computadores vetorizados, que utilizam vários processadores em aplicações simultâneas de operações aritméticas idênticas, sobre diferentes dados. Outros sistemas desse tipo são os chamados *dataflow* e os *reduction machines*, que aplicam simultaneamente diferentes operações à diferentes dados.

É importante salientar que os sistemas computacionais distribuídos existentes não diferem somente nas suas concepções arquitetônicas, eles também divergem, consideravelmente, nas formas como são programados. Alguns fazem uso de linguagens de programação convencionais, suplementando-as, na maioria das vezes, com rotinas de bibliotecas, que garantam a geração e a coordenação de processos. Outros sistemas já

são programados em linguagens completamente novas, especificamente projetadas para aplicações distribuídas.

Muitos estudiosos do assunto advogam que todas as configurações aqui mencionadas podem ser enquadradas na categoria de sistemas distribuídos, outros incluem somente os computadores geograficamente distribuídos, conectados por uma rede de longa distância.

Não obstante o mérito dessa questão fuja da alçada desse trabalho, é fundamental que sejam estabelecidas definições para sistema e, principalmente, programa distribuído, visto que esses conceitos serão utilizados no capítulo quatro, e o projeto e implementação de um programa desse tipo, são objetivos finais neste trabalho. Para tanto será adotada a definição de sistemas distribuídos como a encontrada em Bal et al. (1989), que considera um sistema de computação distribuído como consistindo de múltiplos processadores autônomos, que não compartilham memória primária, mas cooperam entre si, através de envio de mensagens sobre uma rede de comunicação. Cada processador em um sistema desse tipo executa seu(s) próprio(s) fluxo(s) de instrução e utiliza sua própria área de dados locais, ambos armazenados em sua memória local particular. Para programas distribuídos será adotada a definição proposta em Andrews (1991), que define um programa distribuído como sendo um programa concorrente (ou paralelo) no qual os processos componentes comunicam-se através de passagens de mensagens. Muito embora o nome leve o adjetivo "distribuído", decorrente do fato de que esses programas tipicamente são executados em ambientes distribuídos, um programa desse tipo, no entanto, também pode ser executado em um sistema de multiprocessadores com memória compartilhada, ou até mesmo em um monoprocessador.

Uma complementação da caracterização de sistemas distribuídos está fundamentada na comunicação. Em função do tipo de comunicação utilizada entre os processadores de um sistema distribuído, foram criadas duas categorias de sistemas. Uma arquitetura distribuída é dita ser fortemente acoplada se a comunicação é rápida e confiável e, também, se os seus processadores se encontram fisicamente próximos uns dos outros. Como exemplos podem ser citados as grelhas e os hipercubos, etc., que

utilizam uma rede de comunicação, consistindo de ligações ponto a ponto, confiável e rápida, a qual conecta cada processador a algum subconjunto de outros processadores. Já uma arquitetura distribuída é dita ser fracamente acoplada se a comunicação entre os seus processadores, fisicamente dispersos, apresente baixa velocidade e pouca confiabilidade. Exemplos de sistemas dessa classe são as redes de longa distância ou locais de estações de trabalho, que permitem comunicação direta entre dois processadores quaisquer, porém com baixa velocidade e não totalmente confiável. Ocasionalmente uma mensagem pode ser danificada, chegar fora de ordem ou, simplesmente, não alcançar o seu destino. Nesses casos, devem ser utilizados protocolos para implementar comunicações mais seguras.

Geralmente as aplicações que utilizam sistemas distribuídos fortemente acoplados, apresentam altas taxas de comunicação entre os seus processos, pois a assiduidade de intercomunicação entre as unidades de paralelismo é grande. Essas aplicações são ditas possuírem granularidades finas. Por outro lado, as aplicações com baixa granularidade, formadas por processos com baixo nível de interação, podem ser suportadas por sistemas distribuídos fracamente acoplados.

4.3 Arquiteturas de Multicomputadores

Uma vez que se pretende utilizar um sistema distribuído do tipo multicomputador para a implementação do sistema proposto nesse trabalho, será então abordada nesta seção, com mais detalhes, a categoria dos multicomputadores, ver Duncan (1990) e Reed (1987).

Como já visto anteriormente, em um multicomputador a interação entre os seus nós (processadores autônomos com memórias privadas, de onde são obtidas suas instruções e dados) é alcançada, única e exclusivamente, através do envio de mensagens que trafegam pelo sistema por meio de uma rede de interconexão de alta velocidade. Essas redes, que representam o principal elemento determinante do desempenho de um

sistema multicomputador, podem ser divididas em duas classes principais: as redes de interconexão de barramento e as redes de interconexão bipontuais.

4.3.1 Redes de Interconexão de Barramento

As redes de barramento (Figura 4.1) que compreendem o meio de conexão mais simples e de menor custo, apresentam uma grande flexibilidade no tocante ao número de conexões existentes, Feng (1992). Esse modelo permite adição e subtração de nós, sem a necessidade de alterações severas no sistema. Uma conseqüência imediata dessa propriedade está relacionada com o fator de tolerância à falhas, característica importante e desejável em muitos sistemas dessa natureza. Uma eventual falha de um ou até de mais nós não afetará diretamente as atividades dos demais elementos do sistema.

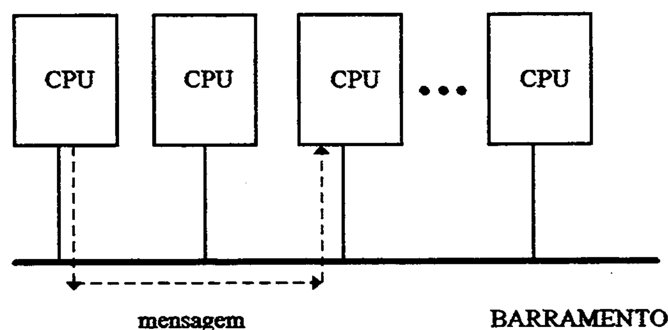


Figura 4.1 Exemplo de multicomputador baseado em barramento

Uma das principais desvantagens dos multicomputadores baseados em rede de barramento está relacionada com a extensibilidade, a qual fica limitada à capacidade do barramento utilizado e cuja sobrecarga pode conduzir à degradação do desempenho do sistema. Pode também ser apontada como uma outra desvantagem considerável, relacionada com o fator tolerância à falhas, a interrupção da comunicação entre todos os nós, decorrente de uma eventual falha no barramento.

4.3.2 Redes de Interconexão Bipontuais

As redes bipontuais são constituídas de um conjunto de canais de comunicação que interligam os nós dois a dois. As mais diversas variedades de topologias de redes de interconexão, que podem ser construídas com base nesse modelo, recaem em duas classes básicas que são as redes estáticas e as redes dinâmicas.

4.3.2.1 Redes Bipontuais Estáticas

Como o próprio nome indica, esse tipo de rede caracteriza-se por apresentar uma topologia fixa, as interconexões dos nós são estabelecidas na construção do sistema e se tornam imutáveis. A topologia ideal de um sistema baseado nesse modelo seria representada pelo grafo completo (Figura 4.2), no qual cada nó encontra-se conectado a todos os demais. Na prática, no entanto, a implementação de um sistema como esse é, muitas vezes, inviável, uma vez que o número de canais necessários para interligar n nós é igual a n^2.

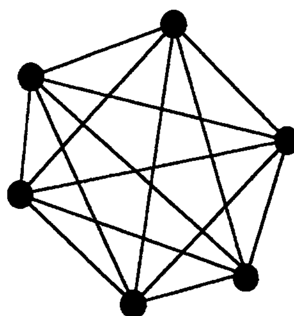


Figura 4.2 Grafo Completo

Redes baseadas no modelo bipontual estática são normalmente construídas com números menores de canais, obrigando que a comunicação entre os nós seja efetuada de maneira indireta e fazendo com que as mensagens percorram nós intermediários até alcançar o nó destino. Serão mostrados, a seguir, dois exemplos de topologias de redes

de interconexão, com número restrito de canais: a grelha e o hipercubo. A topologia conhecida como grelha é bidimensional e seu custo de implementação se torna muitas vezes viável uma vez que o número de canais cresce linearmente em relação ao tamanho da rede, Figura 4.3.

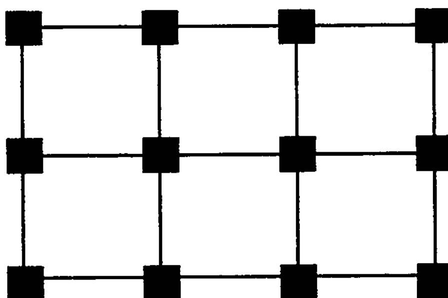


Figura 4.3 Topologia do tipo grelha

O principal problema apresentado por esse tipo de topologia, no entanto, reside no custo de transmissão de uma mensagem entre dois nós não vizinhos. Considerando o pior caso, em que uma mensagem deva ser enviada de um extremo a outro da rede, por exemplo, entre os nós situados no canto superior direito e inferior esquerdo, é necessário que a mensagem percorra $2\ n^{1/2}$ canais, onde n é o número total de nós da rede. Nos casos médios, o número de canais que devem ser percorridos é aproximadamente a metade do pior caso, tornando esta topologia proibitiva para grandes redes Tanenbaum (1992).

Dessa forma, constata-se esforços concentrados na pesquisa e no desenvolvimento de novas topologias, cujos custos médios de transmissão não cresçam proporcionalmente com o tamanho da rede; modelos como os hipercubos são produtos desses esforços. Um hipercubo ou n-cubo Reed (1987) é uma topologia caracterizada pelo parâmetro n, o qual representa sua dimensão e é igual para todos os nós de um sistema. Um hipercubo de dimensão zero é formado por apenas um nó processador. Um hipercubo de dimensão $n+1$ é formado a partir da composição de dois hipercubos de dimensão n. A figura 4.4 ilustra hipercubos de dimensões 1, 2, 3 e 4.

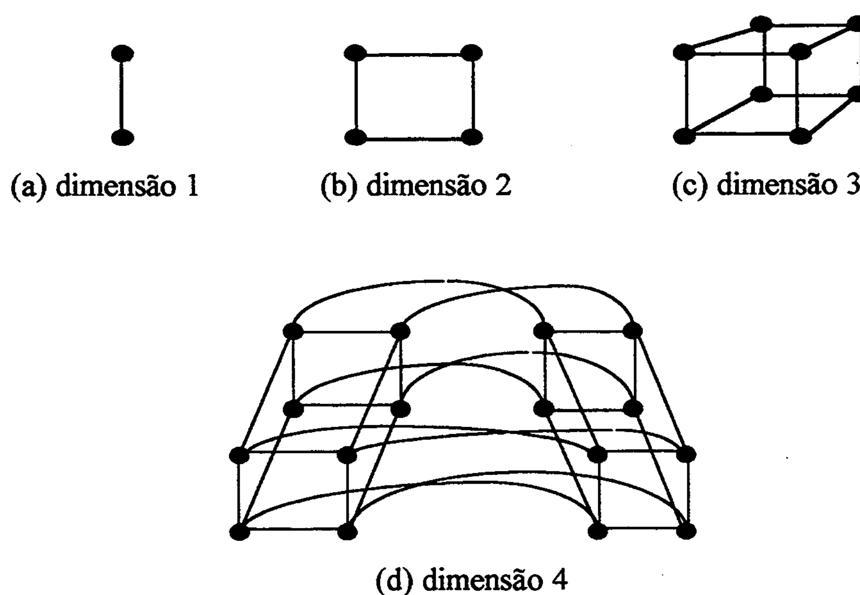


Figura 4.4 Exemplos de hipercubos

O número de nós percorridos por uma mensagem, no pior caso, para uma arquitetura do tipo hipercubo, composta por *n* nós, é igual a $log_2\ n$. Embora o hipercubo proporcione um ganho significativo no custo de transmissão quando comparado à grelha, ele não se constitui em uma arquitetura ideal, uma vez que o número de canais conectados à cada nó deve ser igual à sua dimensão.

4.3.2.2 Redes Bipontuais Dinâmicas

As redes dinâmicas não possuem uma topologia fixa, como as estáticas. As ligações entre os seus nós processadores são obtidas por meio de comutadores de conexões que lhes proporcionam uma maior flexibilidade. Uma rede desse tipo se amolda a uma gama maior de aplicações, uma vez que pode ser configurada de acordo com as características de comunicação de um determinado problema. A seguir, serão abordados, sem maiores detalhes, dois exemplos clássicos de redes desse modelo: as redes dinâmicas baseadas em *crossbar* e as redes dinâmicas multi-estágio.

Um *crossbar* é um dispositivo de *hardware* cujo objetivo é comutar conexões entre um conjunto de processadores, propiciando conexões entre quaisquer pares de processadores desse conjunto, Broomell (1983). Um *crossbar* com *m* entradas e *n* saídas permite a conexão de quaisquer entradas a quaisquer saídas, sendo possível, no entanto, para cada linha de entrada ou saída, a participação de uma única conexão de cada vez. A organização deste dispositivo é ilustrada na Figura 4.5, onde as entradas estão dispostas verticalmente e as saídas horizontalmente, constituindo uma grelha, na qual cada interseção de duas linhas representa um comutador, que controla a conexão entre a entrada e a saída correspondentes. Na mesma figura é destacada uma conexão entre a entrada *E4* e a saída *S3*, resultante do fechamento do comutador correspondente. Nessa situação, a realização de uma nova conexão, envolvendo a entrada *E4* e a saída *S3*, não seria possível.

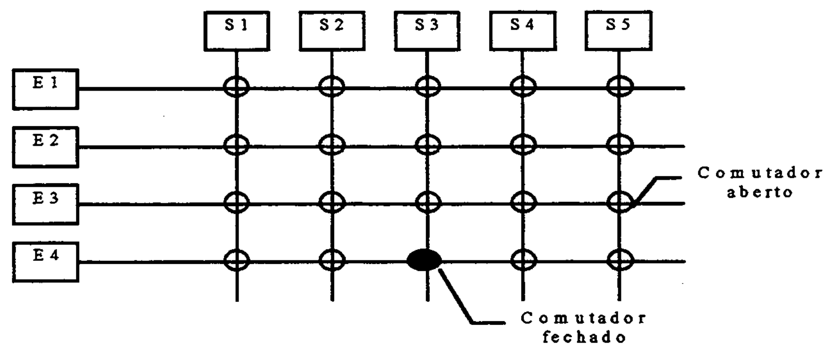


Figura 4.5 Exemplo de um *Crossbar*

O número de comutadores em um *crossbar* com *n* dispositivos a serem conectados é igual a n^2. Outras classes de redes dinâmicas estão sendo pesquisadas, buscando minimizar o número de comutadores utilizados. Por exemplo, as redes chamadas multi-estágio, é uma proposta nesse sentido, ver Hwang (1993) e Youn (1993).

As redes multi-estágio são, normalmente, formadas através da combinação de múltiplos elementos que possuem a capacidade de conectar duas entradas à duas saídas,

simultaneamente. A partir dessa combinação, pode-se construir redes dinâmicas com capacidade de conectar quaisquer elementos.

É importante salientar que dentre os sistemas distribuídos caracterizados como o mais fortemente acoplado e o mais fracamente acoplado, existe uma grande variedade de sistemas computacionais distribuídos. Muito embora a velocidade e a confiabilidade de comunicação desses sistemas diminuam de um extremo a outro, todos se ajustam a um modelo básico comum em que processadores autônomos, conectados por algum tipo de rede, comunicam-se por envio de mensagens. Devido a essa similaridade conceitual, as linguagens de programação utilizadas por esses sistemas não são fundamentalmente diferentes e, em princípio, qualquer uma dessas linguagens pode ser utilizada na programação de uma grande variedade de arquiteturas distribuídas. A escolha de uma linguagem mais adequada fica, portanto, muito mais em função da natureza da aplicação a ser implementada.

A seguir, serão vistas as principais classes de aplicações que têm sido desenvolvidas para sistemas distribuídos, quais os tipos de suportes requeridos e como esses suportes podem ser providos por uma linguagem de programação distribuída, Coulouris (1994).

4.4 Classes de Aplicação de Sistemas Distribuídos

A computação é uma ciência em constante e vigorosa evolução. Suas aplicações, que se alastram pelas mais diversificadas áreas de atuação humana, vêm incessantemente promovendo a busca por sistemas computacionais capazes de oferecer, cada vez mais, maior desempenho e alta confiabilidade. Um dos resultados efetivos alcançados, fruto dessas investigações, é o modelo arquitetônico composto por múltiplos elementos processadores, que veio propiciar os sistemas distribuídos. Sistemas dessa natureza têm-se revelado excelente alternativa, proporcionando um grande poder computacional através da exploração de eventos potencialmente paralelos, encontrados em grande parte das mais diversas aplicações. Esses sistemas, comparados aos sistemas convencionais,

podem apresentar uma melhor relação custo-desempenho e incrementos consideráveis de confiabilidade.

De maneira geral, as razões que justificam as aplicações de sistemas distribuídos podem ser classificadas em quatro principais categorias, ver Bal et al. (1989):

- redução do tempo de processamento;
- aumento da confiabilidade e da disponibilidade;
- melhoria na estruturação do sistema e
- aplicações inerentemente distribuídas.

Em muitos sistemas computacionais, a velocidade de processamento pode-se tornar um fator determinante. Através da exploração das partes de um programa que, potencialmente, possam ser executadas em processadores diferentes, simultaneamente, pode-se alcançar redução significativa no tempo de processamento. Um exemplo típico dessa categoria de aplicações é o sistema desenvolvido nesse trabalho, onde um espaço de soluções possíveis para um determinado problema da produção industrial, deve ser vasculhado por vários processos cooperantes, simultaneamente, almejando uma redução relevante no tempo gasto na determinação de uma boa solução .

Sistemas computacionais distribuídos são potencialmente mais confiáveis do que os tradicionais uma vez que possuem a, assim chamada, propriedade de falha parcial, ou seja, uma vez que os processadores são autônomos, uma falha em um deles não afetará o funcionamento normal dos demais processadores. Para aplicações envolvidas com situações críticas, onde a queda do sistema implica em danos irreparáveis, tais como controle de aeronaves, automatização industrial, etc, essa propriedade é altamente desejável. Dessa forma, confiabilidade pode ser incrementada pela replicação de funções ou dados críticos da aplicação, em vários processadores.

Muitas vezes a implementação de uma certa aplicação se torna mais natural se estruturada como uma coleção de serviços especializados. Cada serviço podendo utilizar um ou mais processadores dedicados, alcançando com isso maior desempenho e alta confiabilidade. Os serviços podem enviar requisições de tarefas a outros através de uma rede. Se uma nova função necessitar ser incorporada ao sistema, ou mesmo, se uma

função já existente necessitar de uma computação mais potente, torna-se relativamente fácil adicionar novos processadores. Um exemplo dessa categoria de aplicação de sistemas distribuídos é o Amoeba, um sistema operacional distribuído, que garante um excelente compartilhamento de recursos computacionais, através da implementação de um serviço de arquivos, um serviço de impressão, um serviço de processos, um serviço de terminais, etc, Mullender (1990).

Finalmente, há aplicações que se caracterizam como inerentemente distribuídas e, assim, a utilização de sistemas distribuídos é natural. Por exemplo, uma companhia com múltiplos escritórios e fábricas pode necessitar de um sistema distribuído de forma que pessoas e máquinas, em diferentes locais, possam se comunicar.

4.5 Suportes para Programação Distribuída

Após a descrição dos modelos arquitetônicos distribuídos existentes e os principais tipos de aplicações para esses sistemas, serão abordadas as diferentes formas como essas aplicações podem ser implementadas, Andrews (1991).

A programação distribuída, aqui referenciada como a atividade de implementação de uma aplicação em um sistema distribuído, requer pelo menos dois requisitos básicos essenciais, inexistentes na programação seqüencial:

- distribuição de diferentes partes de um programa entre os, possivelmente, diferentes processadores e
- coordenação dos processos componentes de um programa distribuído.

Em qualquer uma das categorias de aplicações já mencionadas, um programa distribuído conterá trechos de código que deverão ser processados em paralelo, em diferentes processadores. O primeiro e primordial requisito para que a programação distribuída possa existir é, portanto, a habilidade de se atribuir diferentes partes de um programa à diferentes processadores.

O segundo requisito, não menos importante, diz respeito a coordenação dos processos que compõem um programa distribuído. A comunicação e o sincronismo são

fatores indispensáveis para que os processos possam trocar informações e resultados intermediários e sincronizar suas ações.

Um terceiro requisito do suporte para programação distribuída, que deve ser apontado, particularmente importante para as aplicações tolerantes à falhas, é a habilidade de se detectar uma falha em um dos processadores e recuperá-la em tempo hábil suficiente para se evitar a queda geral do sistema.

O suporte para implementação de aplicações distribuídas, que idealmente deveria contemplar a todos os três requisitos acima aludidos, pode ser provido pelo sistema operacional ou pela linguagem especialmente projetada para programação distribuída.

No primeiro caso, as aplicações são programadas em uma linguagem seqüencial tradicional, estendida com rotinas de biblioteca que invocam primitivas do sistema operacional. Uma vez que as estruturas de controle e os tipos de dados de uma linguagem seqüencial não são adequadas à programação distribuída, esse enfoque muitas vezes pode gerar situações conflitantes, tais como nos dois exemplos considerados a seguir.

Ações triviais, como a criação de um processo ou mesmo o recebimento de uma mensagem de um remetente específico, podem ser perfeitamente expressas através de chamadas de rotinas de biblioteca, relativamente simples. Problemas surgem, no entanto, em situações mais complexas onde, por exemplo, um processo que deseja selecionar o recebimento de mensagens provenientes de alguns processos específicos, onde o critério de seleção dependa, por exemplo, do estado do receptor e do conteúdo da mensagem. Para que o sistema operacional consiga realizar um pedido como esse, provavelmente, um grande número de chamadas complicadas de biblioteca deverá ser necessário. Não deixando de considerar, também, o lado do programador que, com certeza, encontrará severas dificuldades para conseguir expressar sua lógica de programação, em uma linguagem que não apresente notações concisas para tais situações.

Problemas graves com tipos de dados podem surgir em circunstâncias nas quais, por exemplo, uma estrutura de dados complexa deva ser passada como parte de uma mensagem, para um processo remoto. O sistema operacional, desconhecendo a forma como os dados estão representados, é incapaz de converter essa estrutura em pacotes -

seqüências de *bytes* - para enviá-los pela rede. Novamente, o programador é quem deverá escrever o código explicitamente, arranjando a estrutura de dados no envio da mensagem, em uma seqüência de *bytes*, de tal forma que possa ser reconstituída no seu recebimento. Por outro lado, uma linguagem especificamente projetada para programação distribuída, faria essa conversão automaticamente, além de oferecer outras importantes vantagens na programação, tais como, melhor legibilidade, portabilidade e verificações em tempo de compilação.

Normalmente, os serviços de cooperação entre processos oferecidos pelos sistemas operacionais são extremamente simples, podendo ser sintetizados em duas primitivas, ver Ben (1985):

- SEND (mensagem): envia uma mensagem à um processo destinatário e
- RECEIVE (mensagem): recebe uma mensagem de um processo origem.

Dependendo do sistema operacional, estas primitivas podem ter efeito bloqueante ou não. Um bloqueio, normalmente, se estende até que a operação seja efetivamente realizada, caracterizando assim uma forma primitiva de sincronização.

A rigidez desses serviços oferecidos pelos sistemas operacionais constitui o aspecto mais relevante no contraste entre os suportes oferecidos pelos sistemas operacionais, e as linguagens de programação distribuída. Grande parte das linguagens construídas para esse fim apresentam modelos de programação de níveis mais elevados, possibilitando que o programador tenha um poder de abstração muito maior do que naqueles oferecidos pelo modelo simples de troca de mensagens, suportado pela maioria dos sistemas operacionais.

4.6 Modelos de Linguagens de Programação Distribuída

Um grande número de linguagens para programação distribuída surgiu na década 1980, dificultando até a tarefa da escolha da linguagem mais adequada para uma determinada aplicação, principalmente porque os seus modelos básicos de programação dessas linguagens diferenciavam consideravelmente.

O modelo de programação distribuída considerado básico é aquele no qual um conjunto de processos seqüenciais rodam em paralelo, comunicam-se por meio de passagem de mensagens e a detecção de falhas é feita explicitamente Bal et al. (1989). Este padrão reflete diretamente a concepção de sistemas distribuídos, ou seja, um conjunto de processadores autônomos conectados por meio de uma rede de comunicação. Para muitas aplicações, esse modelo de processos e passagem de mensagens é o suficiente e muitas vezes até vantajoso, pois ele pode ser eficientemente mapeado sobre a arquitetura distribuída, munindo o programador do controle total sobre os recursos de *hardware* (processadores e rede) disponíveis.

Entretanto, modelos alternativos de programação distribuída vêm sendo projetados pois, para muitas aplicações que exigem níveis de abstração maiores, o padrão básico se torna insuficiente.

A primitiva de comunicação, originalmente baseada em passagem de mensagens, também passou a ser questionada e novos padrões, que não necessariamente refletem diretamente o modelo de comunicação do *hardware*, começaram a surgir. Dentre os modelos alternativos que maiores impactos causaram ao padrão original de comunicação entre processos, destacam-se os modelos baseados em chamada remota de procedimento e em dados compartilhados, Finkel (1980). Este último, aparentemente paradoxal, já que em um sistema distribuído não existem memórias compartilhadas, merece algumas considerações. Assim como o *hardware* de um sistema pode ser fisicamente distribuído ou não distribuído, o *software* também possui uma distinção similar. No caso de sistemas de *software*, a distinção concerne na distribuição lógica dos dados utilizados, ao invés da distribuição física das memórias. Define-se um sistema de *software* logicamente distribuído como aquele consistindo de múltiplos processos que se comunicam através de passagem de mensagens explícitas. Um sistema de *software* logicamente não distribuído seria, portanto, aquele no qual os processos se comunicam por meio de dados compartilhados. Considerando isoladamente os sistemas de *hardware* e *software*, quatro combinações viáveis diferentes podem ocorrer, gerando as classes de sistemas abaixo, ver Bal et al. (1989):

- *software* logicamente distribuído, rodando em *hardware* fisicamente distribuído;

- *software* logicamente distribuído, rodando em *hardware* fisicamente não distribuído;
- *software* logicamente não distribuído, rodando em *hardware* fisicamente distribuído e
- *software* logicamente não distribuído, rodando em *hardware* fisicamente não distribuído.

A primeira classe é a normal, uma coleção de processos, cada um rodando em um processador diferente e se comunicando através das primitivas *SEND* e *RECEIVE*, que enviam mensagens, por exemplo, utilizando uma rede. A segunda com a mesma estrutura lógica de multiprocessos da anterior, simula a passagem fisica de mensagens através de passagem lógica de mensagens, utilizando memória compartilhada. A terceira classe esconde a distribuição fisica do sistema de *hardware*, simulando memória compartilhada. A última também utiliza comunicação por dados compartilhados, porém nesse caso a existência de memória fisicamente compartilhada facilita a implementação.

Finalmente, o terceiro aspecto importante que deve ser considerado nos modelos das linguagens de programação distribuída, além do paralelismo e da comunicação, é o tratamento de falhas de processadores. O problema crucial nessa questão é conseguir trazer, após a detecção de uma falha através de algum mecanismo utilizado, o sistema de volta a um estado consistente. Esta recuperação normalmente pode ser realizada se as quedas de processadores forem previstas e as devidas precauções forem tomadas durante a computação normal do sistema. Por exemplo, cada processador, em intervalos regulares, deve descarregar seu estado interno em uma memória secundária. Modelos alternativos estão sendo propostos para oferecer, ao programador, mecanismo de alto nível que permita expressar, especificamente, quais processos e dados são importantes e como eles devem ser recuperados após uma falha.

A seguir, serão inspecionadas as principais e as mais representativas primitivas de linguagens de programação distribuídas, que são propostas de soluções diferentes para o tratamento do paralelismo e da comunicação entre processos.

4.6.1 Paralelismo

Esta seção inicia fazendo uma importante distinção entre o paralelismo real e pseudo paralelismo, Trew e Wilson (1991). A execução de um programa em um ambiente paralelo real se caracteriza pela existência de processadores em número igual ou superior ao número de processos que compõem o programa. Na execução de um programa em um ambiente pseudo paralelo, por sua vez, o número de processadores pode ser menor que o número de processos existentes, podendo chegar inclusive a um único processador. Muitas vezes é vantajoso expressar um programa como uma coleção de processos rodando em paralelo, desconsiderando o fato de que esses processos podem rodar simultaneamente. Essa técnica, naturalmente, não tem utilidade nas aplicações em que a busca pela redução do tempo de processamento é o único objetivo.

Algumas linguagens escondem do programador essa distinção entre paralelismo real e pseudo paralelismo, fazendo com que o número de processos rodando efetivamente, em um dado momento, seja totalmente desconhecido. Outras linguagens não escondem essa distinção, permitindo, inclusive que o próprio programador atribua, explicitamente, determinadas partes do seu programa a determinadas unidades de processamento. Muitas vezes esse controle do mapeamento de processos em processadores é interessante, uma vez que aumenta a flexibilidade do sistema. Por exemplo, o programador pode utilizar variáveis compartilhadas na comunicação de processos que, sabidamente, estão rodando em um mesmo processador ou atribuir um determinado processador a um determinado processo por alguma razão especial.

Como será visto a seguir, cada linguagem de programação distribuída possui sua unidade de paralelismo que varia do processo à expressão Hansen (1995). A granularidade do paralelismo também varia de linguagem para linguagem e depende muito do custo de comunicação apresentado pelo sistema de *hardware* distribuído envolvido. Por exemplo, um sistema distribuído fortemente acoplado, tal como o hipercubo, pode suportar eficientemente um paralelismo de granularidade fina. Por outro

lado, um sistema de rede de longa distância inviabilizaria uma computação paralela com essa granularidade.

Finalizando este preâmbulo, é interessante considerar as linguagens que não suportam o controle explícito do paralelismo. Ao invés do programador efetuar a divisão do código do programa em segmentos paralelos, é o compilador que o faz. Algumas linguagens também não permitem controle explícito da comunicação entre processos. O envio de uma mensagem a um processo resulta na geração implícita de um outro processo que gerencia essa comunicação.

4.6.1.1 Expressões de Paralelismo

A maneira como o paralelismo é expresso varia de linguagem para linguagem e depende muito da unidade do paralelismo adotado, que pode ser um processo, um objeto, um comando, uma expressão ou, ainda, uma cláusula, Bal et al. (1989).

Para a grande maioria das linguagens imperativas, destinada à programação distribuída, o paralelismo encontra-se fundamentado na importante noção de processo. Existem várias considerações a esse respeito, mas, em geral, um processo pode ser considerado como um processador lógico que executa códigos seqüencialmente e possui seu próprio estado e dados.

A definição de um processo dentro de um programa distribuído é semelhante a de um procedimento em um programa seqüencial, havendo inclusive linguagens que permitem a definição do tipo "processo".

Uma vez definidos, os processos podem ser criados por duas maneiras bem distintas. Explicitamente, através de alguma construção do tipo *create*, que lembra uma chamada de procedimento em um programa seqüencial, com efeitos, todavia, totalmente diferentes. Enquanto que uma chamada de procedimento resulta apenas em um desvio do fluxo de execução do programa, um *create* inicia a execução de um novo processo, alocando-lhe um processador autônomo disponível. A criação implícita de um processo,

por outro lado, é feita através da declaração de uma variável como sendo de um tipo processo, previamente definido.

Em linguagens baseadas na criação implícita de processo, o número total de processos é fixado em tempo de compilação, devendo, portanto, ser previamente conhecido. Por um lado, esse aspecto é vantajoso, pois simplifica o mapeamento entre processos e processadores físicos. Por outro lado, entretanto, a criação estática de processo impõe restrição aos tipos de aplicações que possam vir a ser implementadas nessas linguagens.

A existência de uma construção do tipo *create* pode tornar a linguagem mais flexível, pois além de permitir criação dinâmica de processos, possibilita a passagem de parâmetros ao processo recém criado. Parâmetros, normalmente, são utilizados nessa situação para estabelecer canais de comunicação entre processos.

Um último aspecto considerado, relacionado com processo é a sua finalização. Um processo, normalmente, termina por si próprio, porém pode existir situações que exijam a existência de primitiva que venha abortar certos processos. Algumas providências podem também ser necessárias para se evitar que um processo tente se comunicar com um outro que já tenha terminado.

Outra unidade de paralelismo considerada é o objeto. Em geral, um objeto é uma entidade auto contida que encapsula dados e comportamento e que interage com o mundo externo (outros objetos) exclusivamente por meio de alguma forma de passagem de mensagem. Os dados contidos dentro de um objeto são invisíveis fora do mesmo. O comportamento de um objeto é definido pela sua classe, que compreende uma lista de operações que podem ser invocadas através do envio de mensagens a esse objeto. Através da herança, consegue-se definir uma classe como uma extensão de uma outra já definida.

As linguagens seqüenciais orientadas a objetos são baseadas no modelo de objetos passivos. Um objeto somente é ativado quando recebe uma mensagem de outro objeto, invocando alguma de suas operações. Enquanto o receptor da mensagem estiver ativo, o emissor estará esperando pelo resultado, tornando assim um objeto passivo. Após o retorno do resultado, o objeto receptor retornará ao estado passivo e o emissor para o

estado ativo. Em qualquer instante do processamento, num modelo como esse, no máximo um objeto poderá estar ativo.

Um ambiente paralelo pode ser obtido, a partir de um modelo seqüencial baseado em objeto, através de uma extensão que incorporasse qualquer uma das seguintes modificações:

- permitir a existência de objeto ativo sem que o mesmo tenha recebido mensagem;
- permitir que o objeto receptor continue executando após ter retornado o seu resultado;
- permitir que um objeto envie mensagens para vários objetos simultaneamente e
- permitir que o emissor de uma mensagem continue executando em paralelo com o receptor.

É interessante observar que o primeiro e o segundo métodos transformam o ambiente em um modelo baseado em objetos ativos, ou seja, é atribuído a cada objeto um processo.

A terceira unidade de paralelismo considerada é o comando. Algumas linguagens distribuídas expressam o paralelismo através de estruturas que agrupam comandos que devem ser processados em paralelo. Esse método, apesar de fácil utilização e entendimento, deixam muito a desejar com relação a estruturação de programa, e não apresentam construções tão gerais como outros mecanismos.

A quarta unidade de paralelismo a ser considerada é a expressão das linguagens funcionais. Em uma linguagem funcional pura, as funções têm o mesmo comportamento das funções matemáticas, ou seja, calculam um resultado que depende somente dos valores dos dados de entrada, não interferindo absolutamente em outras funções do programa. Fazendo uma comparação, as linguagens imperativas permitem que as funções interfiram uma nas outras por várias maneiras: através de variáveis globais, variáveis do tipo ponteiro, etc.

Desde que as funções, em uma linguagem funcional, não exerçam influência alguma umas sobre as outras, elas podem ser executadas em qualquer ordem, inclusive em paralelo. Por exemplo, na expressão abaixo:

$$h(f(10, 30), g(A))$$

é irrelevante se a função *f* for executada antes que a função *g*, ou não. Consequentemente as funções *f* e *g* podem, perfeitamente, ser executadas em paralelo. Em linguagens desse tipo, em princípio, todas as funções podem ser executadas em paralelo, com uma única restrição: a ordem de execução muitas vezes precisa ser considerada. Na expressão do exemplo acima, a função *h* necessita dos resultados das funções *f* e *g*, por conseguinte, *h* deve aguardar até que os resultados de *f* e *g* estejam disponíveis. Esse paralelismo implícito, verificado nas expressões desse tipo, apresenta uma granularidade fina, exigindo arquiteturas adequadas, tal como os computadores do tipo *data-flow*.

Para sistemas distribuídos, o enfoque funcional possui alguns problemas que ainda precisam ser resolvidos. Por exemplo, a execução em paralelo de todas as funções componentes de uma expressão sem restrição, pode não ser vantajosa, principalmente nas situações em que existam várias funções cujas tarefas sejam muito simples, fazendo com que o tempo gasto na comunicação dos resultados entre processos não justifique o paralelismo empregado.

A última unidade de paralelismo a ser considerada é a cláusula, tipicamente encontrada nos programas lógicos. A programação lógica também oferece, de forma natural, várias oportunidades de paralelismo. Por exemplo, os elementos figurantes localizados à esquerda de uma cláusula podem ser considerados como sendo sub teoremas a serem provados para se chegar à prova do teorema, representado pelo único elemento localizado à direita da mesma cláusula.

(1) *T :- T1, T2, T3, T4;*

(2) *T :- T5, T6;*

Nas duas cláusulas acima apresentadas, existem duas maneiras de se implementar o paralelismo. Uma delas é trabalhar as duas cláusulas para o predicado *T* paralelamente, tendo como condição de parada o sucesso de uma delas ou o fracasso de ambas (conhecido como paralelismo *OU*). A outra maneira seria trabalhar todos os sub teoremas, das duas cláusulas, paralelamente, tendo como condição de parada o sucesso de todos ou o fracasso de algum deles (conhecido como paralelismo *E*).

Outra interpretação que resulta na execução paralela de um programa lógico é descrever os sub teoremas em termos de processos. Por exemplo, associando um processo distinto a cada sub teorema a ser provado, na cláusula (1) do exemplo em questão, fica estabelecido que o processo para provar *T* seja substituído pelos quatro processos paralelos que tentam provar *T1, T2, T3 e T4.*

Nesta seção foram descritas várias maneiras diferentes através das quais as linguagens de programação distribuída podem prover suportes para a expressão do paralelismo. Propositadamente foi dada maior ênfase a expressão que tem como unidade de paralelismo o processo, procurando, na medida do possível, não se distanciar demasiadamente dessa noção, uma vez que se pretende trabalhar com uma linguagem imperativa, que tem o processo como base.

Vários padrões de interação entre processos existem na programação distribuída, Hansen (1995). Normalmente, cada um desses modelos está associado a uma técnica de programação que pode ser utilizada na resolução de um grupo de problemas semelhantes de programação distribuída. Por exemplo, no modelo conhecido por cliente servidor há dois tipos básicos de processos: clientes e servidores. Os processos clientes são assim chamados porque constantemente estão requisitando serviços aos servidores. Na maioria das vezes, um processo cliente ao requisitar um serviço ativa um servidor, o qual estava, provavelmente, bloqueado a espera de uma requisição de serviço. Em seguida, o cliente blocar-se-á, esperando a conclusão do serviço solicitado. Usualmente, um processo servidor é não terminante (possui estrutura geral em forma de um laço infinito) e é construído de forma que possa servir à vários clientes diferentes.

4.6.2 Comunicação e Sincronismo entre Processos

Outra questão de fundamental importância, que deve ser considerada no projeto e construção de uma linguagem de programação distribuída, é a maneira como os trechos de programas, que estão rodando em paralelo em diferentes processadores, vão cooperar. Dois tipos de interação são necessários nessa cooperação: a comunicação e o

sincronismo. Usualmente, os mecanismos destinados a comunicação e sincronização entre processos, por estarem fortemente relacionados, são tratados conjuntamente. A situação considerada a seguir ilustra este aspecto. Durante a execução de um programa distribuído contendo dois processos, um processo *P1* requer, em algum ponto de sua execução, algum dado proveniente do resultado de alguma computação efetuada pelo processo *P2*. Devendo existir, portanto, alguma maneira de *P1* e *P2* se comunicarem. Além disso, se *P1* atinge o ponto, na sua execução, que necessita dos dados *D* e se *P2*, por uma razão qualquer, ainda não comunicou a informação, *P1* deve estar preparado para bloquear a sua execução naquele ponto, esperando até que *P2* o faça. Devendo existir, desta forma, alguma maneira dos dois processos se sincronizarem.

4.6.2.1 Passagem de Mensagem

Tanto no envio de uma mensagem como no seu recebimento, vários fatores estão envolvidos e devem ser considerados, Andrews (1991). Quando se envia uma mensagem é fundamental que se tenha bem claro quem está enviando, para quem está sendo enviada e o que está sendo enviado. Pode-se ter garantia, ou não, de que a mensagem chegue ao seu destino, que a mesma seja aceita pelo receptor remoto, que haja alguma forma de confirmação, que haja alguma forma de tratamento de erro, etc. Já no recebimento de uma mensagem as considerações envolvidas são: no computador destino, para qual ou quais processos a mensagem que acàba de chegar é destinada, deve ou não ser criado um processo especial para a manipulação dessa mensagem; se a mensagem é enviada à um processo existente, porém ocupado, será ela enfileirada ou simplesmente descartada, etc.

Nesta seção, serão abordados quatro modelos de interação entre processos baseados em mensagens. Preliminarmente, serão considerados alguns conceitos fundamentais envolvidos com esses modelos, utilizando para tanto a forma mais simples de comunicação existente.

A primitiva básica de interação entre processos, baseada em mensagens, é a chamada envio de mensagem ponto a ponto Bal et al. (1989). Nesse modelo, estão

envolvidos dois processos, o emissor de um lado e o receptor do outro. O processo emissor inicia a interação sempre explicitamente, enviando uma mensagem ou realizando uma chamada remota de procedimento. Na outra ponta, o processo receptor pode fazer a recepção da mensagem explicitamente ou implicitamente. Na recepção explícita o receptor estará executando algum tipo de comando que possivelmente aceitará uma mensagem específica e determinará a ação a ser tomada na sua chegada. Na recepção implícita, por outro lado, a chegada de uma mensagem implica na chamada automática de um código de programa construído para esse fim, dentro do processo receptor. É usual criar, em uma situação como essa, um novo fluxo de execução de controle dentro do processo receptor.

Outro aspecto importante relacionado com a comunicação entre processos é a forma de endereçar, ou de nomear, os processos envolvidos em uma interação. Em outras palavras, para quem o processo emissor deseja enviar sua mensagem e, inversamente, de quem o receptor deseja receber mensagem. Com relação a esse aspecto, o endereçamento de uma mensagem pode ser direto ou indireto. No endereçamento direto, um único processo é especificado (receptor ou emissor). Um esquema de comunicação, baseado nesse modelo de endereçamento, se diz simétrico caso ambos os processos envolvidos (emissor e receptor) enderecem um ao outro. Em um esquema de comunicação desse tipo, porém assimétrico, apenas o emissor estará endereçando seu receptor, que por sua vez estará disposto a receber mensagens de qualquer processo. É importante observar que um esquema de recepção implícito de mensagem, como visto anteriormente, é sempre assimétrico.

Já com relação aos esquemas baseados em endereçamento indireto, um objeto intermediário deve sempre estar envolvido. Nesse objeto, chamado de *mailbox*, é para onde os processos emissores enviaram suas mensagens e de onde os receptores as retiraram. Na sua forma mais simples, um *mailbox* é considerado apenas como um endereço global compartilhado por todos os processos. Existem, no entanto, esquemas mais evoluídos que consideram os endereços dos *mailboxes* como valores numéricos, podendo ser, por exemplo, passados como parte de uma mensagem, permitindo expressar, assim, padrões de comunicação altamente flexíveis.

4.6.2.1.1 Envio de Mensagem Ponto a Ponto

Esse esquema, considerado o modelo mais primitivo de interação entre processos baseado em mensagem, pode ser implementado em duas formas bastante distintas que são a forma síncrona e a forma assíncrona. No modelo síncrono, o processo emissor é bloqueado até que o receptor tenha recebido efetivamente a mensagem (explicita ou implicitamente). Este esquema, portanto, além de permitir a interação entre os processos envolvidos, também os sincronizam. Já no modelo assíncrono, o processo emissor não espera pelo receptor, continuando o seu processamento imediatamente após o envio da mensagem.

No modelo assíncrono, como o emissor não espera pelo receptor, alguns problemas semânticos podem ser verificados, como por exemplo, o que fazer com as mensagens enviadas a um receptor e que ainda não foram processadas. Alguns sistemas alocam *buffers* para o armazenamento dessas mensagens pendentes, outros simplesmente as descartam. Já no modelo síncrono, sua principal desvantagem fica por conta de sua inflexibilidade: o processo emissor sempre deve esperar pelo receptor, mesmo que esse último não tenha que retornar mensagem alguma.

4.6.2.1.2 Envio de Mensagem um Para Vários

Um esquema em que um processo consiga enviar uma mensagem para vários processos simultaneamente pode ser muito útil em diversas circunstâncias. Várias redes, utilizadas em sistemas de computação distribuída, possuem essa facilidade e são capazes de enviar uma mensagem para todos os processadores conectados à rede (*broadcast*), ou a um grupo específico de processadores (*multicast*), gastando para tanto um tempo equivalente ao gasto no envio dessa mensagem a um único processador. O grande problema apresentado por esse modelo é a questão da confiabilidade. Infelizmente, não é possível garantir que as mensagens realmente cheguem a todos os seus destinos. O

hardware tenta enviar as mensagens para todos os processadores envolvidos, porém mensagens podem se perder devido a erros de comunicação ou porque alguns processadores receptores não estavam prontos para aceitá-las. Entretanto, há situações em que essa garantia não seja primordial, por exemplo, para localizar na rede um dos processadores que fornecem um determinado serviço, onde uma requisição pode ser lançada para todos os processadores componentes. Nesse caso, é desnecessário que todos os computadores da rede se manifestem, bastando encontrar apenas uma instância do serviço desejado. Algumas linguagens para programação distribuída fornecem uma primitiva para passagem de mensagem um para vários.

4.6.2.1.3 Chamada Remota de Procedimento

Em muitas aplicações envolvendo interações de processos através de passagem de mensagem, nem sempre o sentido da comunicação é único, ou seja, do emissor para o receptor. Muitas situações exigem que os dois processos se manifestem fazendo com que a comunicação passe a ter dois sentidos. Por exemplo, no modelo de programação cliente-servidor, já aludido anteriormente, um processo cliente, após solicitar um serviço, bloqueia-se a espera de uma mensagem com os resultados do mesmo, passando, assim, de emissor para receptor. O processo servidor correspondente, após o término da tarefa solicitada, deve enviar ao cliente uma mensagem com os resultados obtidos, passando assim de receptor para emissor. Muito embora esse modelo possa ser implementado com o esquema de envio de mensagem ponto a ponto, o qual sempre estabelece sentido único à comunicação entre dois processos, esse trabalho seria facilitado com o emprego de uma ferramenta mais apropriada.

Será apresentada a seguir, na sua concepção básica, uma construção conhecida como chamada remota de procedimento, que além de estabelecer sentido duplo à comunicação entre dois processos, se caracteriza por ser uma ferramenta de maior flexibilidade, capaz de propiciar níveis de abstração mais altos nas implementações, ver Birrel e Nelson (1984).

Uma primitiva para chamada remota de procedimento lembra uma chamada normal de procedimento em um programa seqüencial, exceto que o comando de chamada encontra-se no programa de um processo (chamador), e o código do procedimento em um programa de um outro processo remoto (chamado). Quando um processo chamador invoca um procedimento remoto *PROC* de um processo chamado, os parâmetros reais de *PROC*, fornecidos pelo processo chamador, são enviados ao processo chamado, juntamente com o pedido da chamada. O processo chamado executa então o seu código correspondente ao procedimento, com os valores dos parâmetros fornecidos pelo chamador. Ao término da execução, o processo chamado envia todos os parâmetros resultantes, de volta ao processo chamador, encerrando com isso a interação. É importante salientar que essa interação é inteiramente sincronizada, ambos os processos permanecem bloqueados durante o transcorrer da mesma.

A maioria das linguagens que oferece essa primitiva, procuram apresentar as chamadas remotas de procedimento tão parecidas quanto possível das chamadas normais de procedimento, porém, algumas características adicionais necessitam ser providenciadas para que se consiga uma implementação eficiente. Uma das maiores fontes de problemas encontrados nesse sentido é com relação à utilização de parâmetros do tipo ponteiro e parâmetros passados por referência, justificável pela ausência de memória compartilhada.

4.6.2.2 Dados Compartilhados

Na seção precedente, foram abordados os principais modelos de comunicação e sincronismo entre processos baseados em passagem de mensagem. Nesta seção serão introduzidos os mecanismos de interação de processos baseados em memória compartilhada, Baillie (1988).

Sempre que dois processos têm acesso a uma mesma variável, uma forma de comunicação e ou sincronismo pode ser estabelecida. Por exemplo, um processo pode atribuir um certo valor a uma variável, indicando a um segundo processo, que um

determinado conjunto de dados está disponível, ou que um determinado serviço está concluído. Para que uma situação como essa possa ocorrer, os processos devem estar ou rodando na mesma máquina onde a variável está armazenada, ou acessarem essa variável enviando uma mensagem para a máquina onde a mesma se encontra. Dessa forma, esse tópico fica melhor organizado se dividido em duas partes bem distintas.

4.6.2.2.1 Memória Compartilha em Ambiente Pseudo Paralelo

Como já mencionado anteriormente, muitas linguagens distribuídas suportam processos rodando em ambientes com um único processador, no chamado pseudo paralelismo. A utilização de variáveis compartilhadas para a comunicação e sincronismo de processos rodando em ambientes dessa natureza, tem sido extensivamente estudada e utilizada sob o tópico geral designado por controle de concorrência, Silberschatz e Peterson (1994).

Sempre que um conjunto de processos opera sobre uma área compartilhada de dados, sem que qualquer precaução seja tomada, o resultado dessa operação, muitas vezes, pode ser desastroso. A seguir, será considerada uma situação típica, na qual, o acesso simultâneo a uma variável compartilhada entre dois processos, pode conduzir a erros graves.

Processo A	*Processo B*
1. TMPA := X;	*1. TMPB := X;*
2. TMPA := TMPA + 1;	*2. TMPB := TMPB + 1;*
3. X := TMPA;	*3. X := TMPB;*
...	...

Figura 4.6 Trechos de dois processos paralelos

Assume-se que *TMPA* e *TMPB* são variáveis locais aos processos *A* e *B*, respectivamente, que *X* é uma variável global, portanto compartilhada entre os dois processos, com valor inicial igual a 0. O valor correto da variável *X*, ao final das

execuções dos fragmentos de código dos dois processos apresentados na figura 4.6 é 2. Ambos incrementam uma unidade ao valor de *X*. Esse valor correto, no entanto, só será obtido se um dos processos executar o comando 1, após o outro processo já ter executado o comando 3. Como a ordem de execução desses comandos nos dois processos é imprevisível, o valor final da variável *X* é indeterminado, podendo variar de uma execução para outra.

Dentro do controle de concorrência, uma situação como a acima descrita é designada por um problema de exclusão mútua, isto porque, para resolvê-lo é necessário que se garanta , ao processo que tenha o direito, acesso exclusivo à variável *X*.

A forma mais simples de se expressar exclusão mútua em uma linguagem é cercar as regiões dos códigos dos processos que fazem acesso aos dados compartilhados, utilizando para tanto estruturas delimitadoras. Por exemplo, utilizando os mesmos trechos dos processos anteriores e a estrutura MUTEXBEGIN/MUTEXEND (do Inglês MUTUAL EXCLUSION), um código correto, que sempre garante o valor de *X* igual a 2 pode ser escrito como:

Processo A	*Processo B*
MUTEXBEGIN	*MUTEXBEGIN*
TMPA := X;	*TMPB := X;*
TMPA := TMPA + 1;	*TMPB := TMPB + 1;*
X := TMPA;	*X := TMPB;*
MUTEXEND	*MUTEXEND*

Figura 4.7 Exemplo de exclusão mútua

Na figura 4.7, o processo que primeiro alcançar o delimitador *MUTEXBEGIN*, entra na sua região, conhecida como região crítica, impedindo que o outro processo o faça. O processo perdedor deve, então, aguardar até que o vencedor alcance o delimitador *MUTEXEND*, da sua região crítica, liberando assim o acesso à variável *X*. Em uma situação em que envolva vários processos, pode haver a formação de uma fila de processos aguardando liberação de acesso.

Muitas vezes, no entanto, a simples ação de se impedir que processos continuem suas execuções até que um outro esteja fora de sua região crítica, não é o suficiente. Se o objetivo for prevenir que um processo continue até que alguma condição mais geral ocorra, então o que está se almejando é a sincronização do processo com aquela condição. Sincronização, portanto, pode ser considerada como uma generalização da exclusão mútua.

Existem vários mecanismos que podem ser utilizados na obtenção de exclusão mútua e sincronização entre processos em ambientes pseudo paralelos, ver Silberschatz e Peterson (1994) e Tanenbaum (1992). Na prática, entretanto, um dos mais empregados pelas linguagens de programação é o semáforo. Essa ferramenta, introduzida por Dijkstra (1965), normalmente é implementada nas linguagens como um tipo de variável que só pode ser operada, única e exclusivamente, por duas operações: a operação P e a operação V. Uma operação P, sobre uma variável do tipo semáforo, diminui seu valor em uma unidade, caso esse valor seja maior ou igual a um. Caso o valor da variável seja igual a zero, essa operação tem efeito bloqueante sobre o processo que a executou. Esse processo fica bloqueado até que o valor da variável se torne igual a um. Esta alteração no valor da variável é conseqüência da execução de uma operação V, sobre essa mesma variável, realizada, eventualmente, por um outro processo. A operação V, incondicionalmente, aumenta, em uma unidade, o valor de uma variável semáforo.

Sem a pretensão de se estender demasiadamente nesse assunto, mas não deixando de ressaltar a potencialidade e a simplicidade da utilização do semáforo como uma ferramenta de exclusão mútua e sincronização, será apresentado a seguir um exemplo de sua utilização.

A exemplo do modelo cliente servidor mencionado anteriormente, um segundo esquema de programação concorrente é proposto por Andrews (1991). Este esquema, baseado em processos, utilizado na resolução de um grande número de problemas semelhantes, é o conhecido por modelo produtores e consumidores. Nesse padrão de problemas, devem existir processos que continuamente produzem algum tipo de recurso, e processos que consomem esses recursos produzidos. Tomando esse modelo como exemplo, será considerada uma situação em que vários processos continuamente

produzem mensagens e as depositam em um *buffer* de tamanho igual a *n*. Do outro lado, também vários processos, continuamente retiram mensagens do *buffer* e as consomem. Os trechos de códigos abaixo, que representam os corpos de um processo produtor e de um processo consumidor, é uma demonstração da utilização de semáforos para garantia da exclusão mútua e do sincronismo entre processos.

```
Processo Produtor
    LOOP
      Produz mensagem;
      P ( VAGA );
        P ( LIVRE );
              Coloca mesagem no buffer;
        V ( LIVRE );
      V ( MENSAGEM );

    ENDLOOP;
```

```
Processo Consumidor
    LOOP

      P ( MENSAGEM );
        P ( LIVRE );
              Retira mensagem do buffer;
        V ( LIVRE );
      V ( VAGA );
        consome mensagem;
    ENDLOOP;
```

Figura 4.8 Exemplo de utilização de semáforos

Na figura 4.8, é assumido que *VAGA*, *MENSAGEM* e *LIVRE* são variáveis do tipo semáforo, cujos valores iniciais são 10, 0 e 1, respectivamente. Um processo produtor, seqüencialmente, produz uma mensagem; disputa uma vaga no *buffer* (de 10 posições), executando *P(VAGA);* disputa o acesso ao *buffer*, executando *P(LIVRE);* coloca uma mensagem no *buffer*; libera o *buffer*, executando *V(LIVRE);* aumenta em uma unidade o número de mensagens disponíveis, executando *V(MENSAGEM);* repete o ciclo. Por outro lado, um processo consumidor, seqüencialmente: disputa uma mensagem, executando *P(MENSAGEM);* disputa o acesso ao *buffer*, executando *P(LIVRE);* retira uma mensagem do *buffer*; libera o *buffer*, executando *V(LIVRE);* aumenta em uma unidade o número de vagas existentes, executando *V(VAGA);* consome a mensagem; repete o ciclo.

As variáveis *VAGA* e *MENSAGEM* estão promovendo a sincronização entre as duas classes de processos, da seguinte maneira: quando o valor de *VAGA* é igual a 0, um processo produtor deve esperar por um consumidor, quando o valor de *MENSAGEM* é 0, um processo consumidor deve esperar por um produtor. A variável LIVRE, por sua vez, controla o acesso exclusivo ao *buffer*, para ambas as classes de processos.

O exemplo, na figura 4.8, serviu para ressaltar a eficiência e a simplicidade das variáveis semáforos quando empregadas na coordenação de processos em ambientes com memória compartilhada. Entretanto, essa simplicidade acaba sendo o seu maior problema. A utilização dessa primitiva, que espalha as operações *P* e *V* entre os comandos de um programa, sem qualquer metodologia de programação, pode gerar programas altamente desestruturados e confusos, onde a corretude dificilmente está garantida. No próprio exemplo considerado, só para se ter uma idéia da dimensão do problema, a simples troca da ordem das operações

```
  ...
P (VAGA);
P (LIVRE);
  ...
```

por

```
  ...
P(LIVRE);
P(VAGA);
  ...
```

no corpo do programa dos processos produtores, pode conduzir o sistema todo a uma situação de impasse irreparável, onde todos os processos se bloqueiam em um ciclo, um a espera de outro, no chamado bloqueio fatal *(deadlock)*.

A busca progressiva por melhores soluções para o problema da coordenação de processos em ambientes com memória compartilhada levou Hansen (1973) e Hoare (1974) a desenvolverem uma construção de linguagem de alto nível que garante uma sincronização apropriada de processos, de forma automática, conhecida por monitor.

Um monitor é um mecanismo que permite um compartilhamento seguro e eficiente de tipos abstratos de dados entre processos Silberschatz e Peterson (1994). A sintaxe de um monitor é idêntica a de uma classe da programação orientada a objetos. A diferença semântica principal é que o monitor assegura exclusão mútua, isto é, apenas um processo a cada tempo pode estar ativo dentro de um monitor. Uma vez que essa propriedade fica garantida pelo próprio monitor, o programador não necessita codificar explicitamente essas restrições de sincronização. Nos esquemas de sincronização onde se faz necessário a utilização de construções condicionais, o programador tem a sua disposição variáveis do tipo condição. Este tipo de variáveis, utilizadas somente nos procedimentos internos de um monitor, são operadas pelas operações conhecidas por *SIGNAL* e *WAIT*. A operação *SIGNAL* sobre uma variável do tipo condição, por exemplo *SIGNAL(C)*, sinaliza um processo na fila de espera por essa variável, fazendo com que o mesmo retorne a executar o procedimento, no ponto em que foi bloqueado, como conseqüência da execução de uma operação *WAIT(C)*, por ele realizada. Como no interior de um monitor somente um processo pode estar ativo em um dado momento, o processo que executa *SIGNAL(C)* e encontra a fila de espera pela variável condição *C* não vazia, deixa o monitor, colocando-se em uma fila, de alta prioridade, conhecida como fila de processos sinalizadores. No caso em que a fila de espera pela variável *C* encontra-se vazia, a operação *SIGNAL* não tem efeito algum. Uma operação *WAIT* sobre uma variável do tipo condição, por exemplo *WAIT(C)*, bloqueia, incondicionalmente, o processo executor, posicionando-o na fila de espera por *C*. A seguir, será resolvido, através da utilização de um monitor, o mesmo problema dos produtores e consumidores, anteriormente considerado.

Um monitor para coordenar a utilização do ***buffer*** de 10 posições pode ter a seguinte forma:

```
MONITOR BUFFER;
  VAR
      VAGA, MENSAGEM : CONDIÇÃO;
      NVAGA, NMSG    : INTEGER;
  PROCEDURE COLOCA;
```

```
BEGIN
    IF ( NVAGA = 0 ) THEN  WAIT ( VAGA );
    COLOCA MENAGEM NO BUFFER;
    NVAGA  :=  NVAGA - 1;
    NMSG    :=  NMSG  + 1;
    SIGNAL ( MENSAGEM );
END; { COLOCA }
PROCEDURE RETIRA;
BEGIN
    IF ( NMSG = 0 ) THEN  WAIT ( MENSAGEM );
    RETIRA MENSAGEM DO BUFFER;
    NVAGA  :=  NVAGA + 1;
    NMSG    :=  NMSG   - 1;
    SIGNAL ( VAGA );
END; { RETIRA }
BEGIN
  NVAGA := 10;
  NMSG   :=  0;
END; { MONITOR BUFFER }
```

Figura 4.9 Exemplo de monitor

Na figura 4.9, um processo produtor, entrando no monitor *BUFFER* através do procedimento *COLOCA* e encontrando uma vaga disponível, coloca sua mensagem no *buffer;* diminui o número de vagas em uma unidade; aumenta o número de mensagens em uma unidade; sinaliza, através da operação *SIGNAL(MENSAGEM),* um eventual processo consumidor, que já tenha entrado no monitor pelo procedimento *RETIRA,* mas que não tenha encontrado uma mensagem disponível. Se não houver uma vaga disponível, esse processo produtor é bloqueado e inserido na fila de espera pela variável condição *VAGA*.

Um processo consumidor, entrando no monitor *BUFFER* através do procedimento *RETIRA* e encontrando uma mensagem disponível: retira a mensagem do *buffer*, incrementa o número de vagas de uma unidade; decrementa o número de mensagens de uma unidade; sinaliza, através da operação *SIGNAL(VAGA)*, um eventual processo produtor que já tenha entrado no monitor com sua mensagem, mas que não tenha encontrado uma vaga para colocá-la. Se não houver uma mensagem, esse processo consumidor é bloqueado e inserido na fila de espera pela variável condição *MENSAGEM*.

Os códigos apresentados na figura 4.10, agora extremamente simplificados, representam um processo produtor e um consumidor utilizando um monitor para sincronizarem:

Processo Produtor	*Processo Consumidor*
LOOP	*LOOP*
produz mensagem;	*BUFFER.RETIRA;*
BUFFER.COLOCA;	*consume mensagem;*
END;	*END;*

Figura 4.10 Exemplo da utilização de um monitor

Muito embora o monitor represente um avanço significativo no desenvolvimento de ferramentas para coordenação de processos em ambientes com memória compartilhada, ele não se constitui em uma panacéia, havendo inclusive algumas objeções em sua utilização, principalmente relacionadas com chamadas aninhadas de monitores, a própria utilização das variáveis do tipo condição, etc. O que se pode afirmar, no entanto, é que o monitor, juntamente com o semáforo, são ferramentas que estão efetivamente sendo empregadas na prática.

4.6.2.2.2 Memória Compartilhada em Ambiente Paralelo Real

Finalmente, encerrando este capítulo, será considerada a situação em que processos em máquinas diferentes interagem entre si, utilizando memória compartilhada.

À primeira vista, a utilização de variáveis compartilhadas na comunicação e sincronismo de processos rodando em diferentes processadores parece não ter muito sentido, pois tais sistemas não possuem memórias fisicamente compartilhadas. No entanto, em muitas circunstâncias, o paradigma da programação distribuída baseado em dados compartilhados pode apresentar vantagens sobre o baseado em envio de mensagens. Por exemplo, enquanto que uma mensagem geralmente transfere informações entre dois processos específicos, um dado compartilhado pode se tornar acessível por qualquer processo. A atribuição de um dado compartilhado, conceitualmente, tem efeito imediato, enquanto que o intervalo de tempo entre o envio e o recebimento de uma mensagem pode ter um valor considerável. Por outro lado, como visto na seção imediatamente anterior, a utilização de dados compartilhados requer precauções para se prevenir que múltiplos processos alterem simultaneamente o mesmo dado. Dessa forma, como nenhum dos dois paradigmas pode ser considerado universalmente o melhor, ambos são dignos de investigação.

A princípio, qualquer tipo de dado compartilhado pode ser implementado através da simulação de memória compartilhada. Entretanto, as variáveis compartilhadas simples não são implementadas em linguagens de programação distribuída conhecidas, muito provavelmente por considerações de desempenho. No entanto, vários outros modelos de comunicação, baseados em dados compartilhados, existem e são bem mais adequados a sistemas distribuídos. Nesta seção, será abordado apenas um deles, referenciado por estruturas de dados distribuídas, que foi escolhido em função de sua usabilidade e também de sua proximidade com o trabalho proposto. Esse modelo tem sido implementado em diferentes tipos de arquiteturas e as linguagens que o implementa são úteis, principalmente, nas situações em que a distribuição fisica da memória do sistema não necessita ser do conhecimento do programador.

Esse paradigma, que permite a manipulação de uma estrutura de dados por vários processos em máquinas diferentes, simultaneamente, está fortemente baseado no conceito de tuplas, Hansen (1987). Um espaço de tuplas (*ET*) é conceitualmente uma memória global compartilhada por todos os processos de um programa distribuído, muito embora sua implementação não requeira memória fisicamente compartilhada. Os

elementos de um *ET*, designados por tuplas, são seqüências ordenadas de valores, similares ao tipo registro. Por exemplo, a seqüência:

[30 , José , FALSE]

pode representar uma tupla com três campos, sendo um do tipo inteiro, outro *string* e o último *boolean*.

Apenas três operações estão definidas sobre um *ET*: a operação *OUT*, a qual adiciona uma tupla a um espaço; a operação *READ*, que lê uma tupla de um espaço e a operação *IN*, que também lê uma tupla de um *ET*, porém remove-a do seu espaço. Essas operações são atômicas, isto é, várias operações simultaneas realizadas sobre uma mesma tupla têm o efeito final idêntico ao causado por essas operações, quando executadas em alguma ordem seqüencial (indefinida) sobre essa mesma tupla. Diferentemente das variáveis compartilhadas, as tuplas não possuem endereços, são acessadas pelos seus conteúdos. Uma tupla pode ser denotada através da especificação do valor ou do tipo de cada campo que a compõe. Essa especificação pode ser expressa, em uma operação, pelo fornecimento de um parâmetro real ou por um parâmetro formal, respectivamente. Por exemplo, supondo que *IDADE* seja uma variável do tipo inteiro e *CASADO* uma variável do tipo *boolean*, então a tupla anteriormente mostrada pode, ser recuperada do seu espaço através da operação:

READ(VAR IDADE, "José" , VAR CASADO)

Essa mesma tupla pode ser lida e removida do seu espaço através da operação:

IN(VAR IDADE, "José" , VAR CASADO)

Em ambas as operações, a variável *IDADE* recebe o atributo do primeiro campo, ou seja 30, e a variável *CASADO* fica com o atributo do último campo, ou seja *FALSE*.

O primeiro passo na execução de uma operação de *READ* ou *IN* é a busca no *ET* de uma tupla cujos campos casem, em valor ou tipo, com os parâmetros fornecidos. Se várias tuplas forem encontradas, uma delas é escolhida aleatoriamente. Não havendo nenhum casamento, a operação e o processo executor são bloqueados até que um outro eventual processo adicione uma tupla no espaço que venha realizar o casamento, utilizando, para tanto, uma operação *OUT*. Não há operação que modifique uma tupla em

um *ET*. A troca de uma tupla se faz através de uma operação de remoção *IN*, seguido de uma inserção *OUT*.

Em uma implementação distribuída de *ET*, uma grande responsabilidade do suporte é a distribuição das tuplas entre os processadores. Várias estratégias são possíveis, tais como, replicação total do *ET* em cada processador, distribuição das tuplas em processadores específicos, ou armazenagem de uma tupla no processador que realizou a operação *OUT*.

CAPÍTULO 5

O Método Desenvolvido

Neste capítulo, é descrita a forma proposta de combinar os algoritmos genético (AG) e *simulated annealing* (SA) na determinação do *makespan* mínimo em problemas de programação da produção do tipo *job shop* (PPJS). Inicialmente, o problema é formalmente definido com a apresentação do modelo matemático utilizado. Em seguida, é apresentada as principais abordagens existentes no tratamento desse tipo de problema. O método desenvolvido é, então, descrito nas suas duas versões: seqüencial e paralela. O capítulo se encerra com a realização de testes computacionais e análise dos resultados obtidos .

5.1 Definição Formal do Problema

Na definição matemática do problema, são utilizados três conjuntos, a saber: um conjunto J dos n *jobs* a serem realizados, um conjunto M das m máquinas disponíveis e um conjunto O das N operações a serem executadas por essas máquinas na realização dos *jobs* em J. A cada operação está associado um *job*, uma máquina e um tempo de processamento. Ou seja, para cada operação $v \in O$, existe um job $J_v \in J$ a qual ela pertence, uma maquina $M_v \in M$, na qual será realizada, e um tempo de processamento t_v, pertencente ao conjunto dos números naturais. O conjunto O é decomposto em n partições através de uma relação binária "$\rightarrow$" existente entre duas operações sucessivas de um mesmo *job*. Isto é, se uma operação v antecede w, $v \rightarrow w$, então elas pertencem a

um mesmo *job*, ou seja $J_v = J_w$ e não existe uma operação $x \notin \{v, w\}$ tal que $v \rightarrow x$ ou $x \rightarrow w$.

O problema, então, pode ser definido como: determinar, para cada operação $v \in O$, o seu tempo de início, ti_v, tal que, $\max_{v \in O} (ti_v + t_v)$ seja minimizado, sujeito às seguintes restrições:

se $v \rightarrow w$, então $(ti_w - ti_v) \geq t_v, \quad \forall v, w \in O$ (5. 1)

se $M_v = M_w$, então $(ti_w - ti_v) \geq t_v \vee (ti_v - ti_w) \geq t_w, \quad \forall v, w \in O$ (5.2)

$ti_v \geq 0 \ , \ \forall v \in O$ (5.3)

A primeira restrição determina que o intervalo entre os tempos iniciais de duas operações seja maior ou igual ao tempo de processamento da primeira operação executada. A segunda, conhecida como restrição disjuntiva, determina que se duas operações v e w são executadas pela mesma máquina, então, ou w é executada após v, ou v é executada após w. A terceira restrição simplesmente especifica que todos os tempos de início de operação sejam positivos ou zero.

Na resolução de problemas desse tipo também são assumidas as seguintes condições:

- o tempo de processamento de cada operação é previamente conhecido;
- todos os *jobs* estão disponíveis no tempo zero e
- não há máquinas inoperantes.

5.2 Modelagem do Problema

A forma empregada para se representar o problema está baseada no modelo de grafo disjuntivo, proposto por Roy e Sussman (1964). Segundo esse modelo, o grafo disjuntivo que pode representar um problema de programação da produção do tipo *job shop*, é definido como $G = (V, A, E)$, onde V é o conjunto dos vértices do grafo e corresponde às operações em O. Ao conjunto V são adicionadas duas operações fictícias enumeradas com 0 e $N+1$, ou seja $V = O \cup \{0, N+1\}$.

O peso de cada vértice em V é dado pelo tempo de processamento da operação correspondente, t_v. As operações 0 e $N+1$, por definição, recebem peso igual a zero.

O conjunto A em G é formado pelos arcos que conectam operações consecutivas de um mesmo *job*. Também estão incluídos nesse conjunto os arcos que conectam a operação 0 à primeira operação real de cada *job*, bem como a última operação de cada um deles à operação $N+1$. Dessa forma o conjunto A é expresso por:

$$A = \{(v, w) \ / \ v, w \in O, v \rightarrow w\} \cup$$
$$\{(0, w) \ / w \in O \text{ e não existe } v \in O / v \rightarrow w\} \cup$$
$$\{(v, N+1) / v \in O \text{ e não existe } w \in O / v \rightarrow w\}$$

Finalmente, o conjunto E de G é formado pelas arestas que conectam duas operações a serem realizadas pela mesma máquina, ou seja:

$$E = \{\{v, w\} / M_v = M_w\}$$

A figura 5.1 ilustra um grafo disjuntivo modelando um problema *job shop* com três jobs, cada um dos quais consistindo de quatro operações.

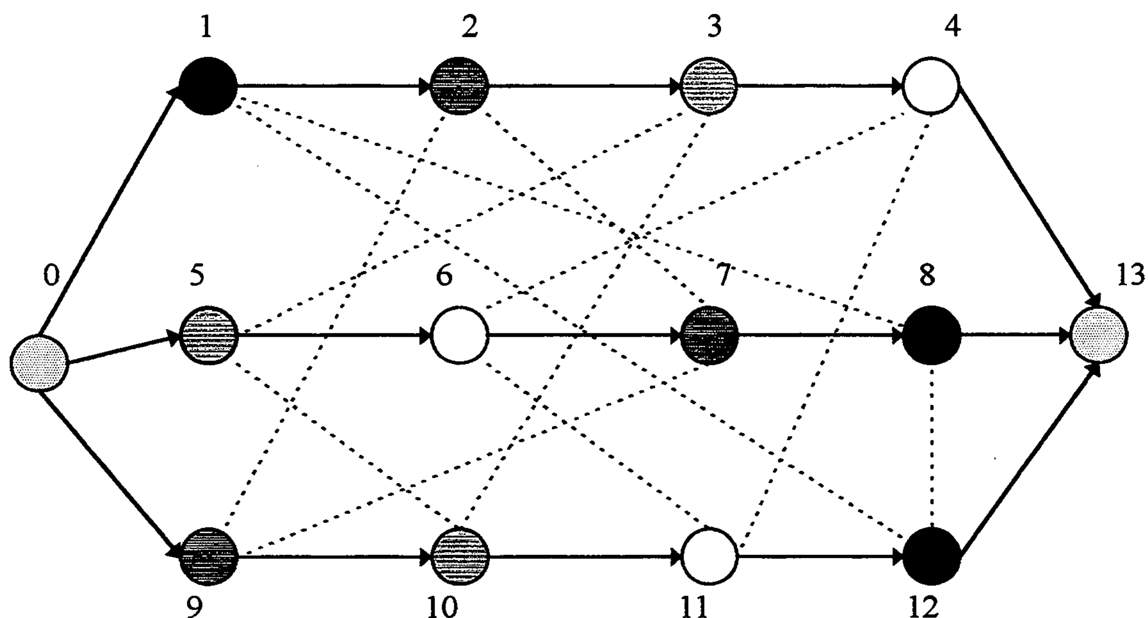


Figura 5.1 Exemplo de representação de um problema *job shop*, com três *jobs* e quatro máquinas, através de um grafo disjuntivo.

Como pode ser observado na figura 5.1, as operações rotuladas com 1, 8 e 12 são executadas por uma mesma máquina, por exemplo *M1*. Cada uma dessas operações pertence a um único *job*, por exemplo *J1, J2 e J3*, respectivamente. De forma similar, pode-se convencionar que as operações 2, 7 e 9 são executadas pela máquina *M2* e pertencem aos *jobs J1, J2 e J3*, respectivamente; as operações 3, 5 e 10 são executadas pela máquina *M3* e pertencem aos *jobs J1, J2 e J3*, respectivamente; as operações 4, 6 e 11 são executadas pela máquina *M4* e pertencem aos *jobs J1, J2 e J3*, respectivamente. As operações inicial e final, rotuladas com 0 e 13 respectivamente, são fictícias, não são executadas, porém, encontram-se na seqüência de operações de todos os *jobs*.

Para cada par de operação $v, w \in O$ com $v \rightarrow w$, a restrição (5.1) é representada por um arco (v, w) em A (uma seta no grafo da figura 5.1). No entanto, para cada par de operações $v, w \in O$, pertencente à mesma máquina, ou seja $M_v = M_w$, a restrição (5.2) é representada por uma aresta $\{v, w\}$ em E (uma linha pontilhada no grafo da figura 4.1). As duas maneiras possíveis de se estabelecer a disjunção ($\vee$) imposta por essa restrição corresponde as duas orientações possíveis que se pode dar à aresta $\{v, w\}$.

Um conjunto de orientações de todas as arestas do grafo G da figura 5.1, transforma-o em um grafo orientado, que representa uma das soluções possíveis do problema modelado. O valor dessa solução, ou seja o *makespan*, corresponde ao tamanho do maior caminho entre os vértices 0 e $N+1$ desse digrafo. Em outras palavras, um conjunto de orientação de todas as arestas de G, define, para cada máquina em M, uma ordenação ou permutação das operações por ela processadas, decompondo também o conjunto O em m partições. Para cada conjunto de permutação, um novo conjunto de arcos é definido em G e um novo digrafo é gerado.

Dessa forma, utilizando esse modelo de representação, o problema de PPJS passou a ser tratado considerando os conjuntos de permutações de máquinas, isto é, determina-se, para cada conjunto de permutação, o maior caminho no digrafo correspondente. A melhor solução será aquela que apresentar o menor valor desse caminho.

Na aplicação, tanto de SA como AG, na resolução de qualquer problema de otimização combinatorial, é fundamental que se defina precisamente as configurações ou soluções do problema. Uma representação natural do espaço de soluções para o problema em questão é, portanto, aquela que contém todos os conjuntos de permutação. Define-se, assim, uma solução *i* para o problema, como sendo o conjunto de permutações das máquinas:

$$\Pi_i = \{\pi_{i1}, \pi_{i2}, \dots, \pi_{im}\}$$

onde cada elemento desse conjunto, π_{ik}, deve ser interpretado como a ordem na qual as operações da máquina *k* serão processadas, naquela solução Π_i. Conseqüentemente, o número de soluções possíveis, ou seja o tamanho do espaço, para um problema com m máquinas é dado por $\Pi_{k=1}^{m} m_k!$, onde m_k é o número de operações a ser processado pela máquina *k*.

Retornado à figura 5.1, uma solução possível para o problema ilustrado é:

$$\{ (2, 3, 1), (1, 3, 2), (1, 2, 3), (3, 1, 2) \}$$

Esta expressão deve ser interpretada como: a máquina 1 executa as operações correspondentes aos *jobs* 2, 3 e 1, nesta ordem; a máquina 2 executa as operações correspondentes aos *jobs* 1, 3 e 2, e assim por diante. Tendo em vista que a abordagem proposta no presente trabalho envolve a combinação dos algoritmos SA e AG, essa definição de solução é importante, pois pode ser perfeitamente utilizada por ambos.

Formalmente, cada configuração *i* define, a partir do grafo disjuntivo básico $G = (V, A, E)$, um digrafo $D_i = (V, A \cup E_i)$, onde:

$$E_i = \{ (v, w) \,/\, \{v, w\} \in E, \text{ e } \pi_{ik}^{p}(v) = w, \text{ para algum } k \in M \text{ e } 1 \le p \le m_k - 1 \},$$

com $\pi_{ik}^{p}(v)$ denotando a operação que segue *v*, após *p* operações sucessivas.

Como aludido anteriormente, a avaliação de uma solução *i* de um problema modelado por um grafo $G = (V, A, E)$, implica na determinação do valor do maior caminho entre os vértices *0* e *N+1* do digrafo correspondente D_i. Na realidade, a função de avaliação não atua sobre o conjunto E_i completo, mas sim, apenas com aqueles arcos de E_i que conectam operações sucessivas de uma mesma máquina, ou seja, essa função trabalha diretamente sobre o digrafo $D1_i = (V, A \cup E1_i)$, onde:

$El_i = \{ (v, w) \; / \; \{v, w\} \in E, \; e \;\; \pi_{ik}(v) = w$, para algum $k \in M\}$

Como o tamanho do maior caminho do digrafo D_i é o mesmo que o de Dl_i, essa redução em El_i é realizada com o objetivo de diminuir a complexidade da função de avaliação, uma vez que esta é proporcional ao número de arcos do digrafo de entrada.

A implementação dessa função está baseada nas equações de Bellman (1958) e no algoritmo para determinação do maior caminho de um digrafo apresentado em Christofides (1975). Ela foi desenvolvida especialmente para esse problema, uma vez que os vértices dos digrafos gerados não são ordenados e, sendo assim, ciclos correspondentes a soluções inviáveis necessitam ser rapidamente detectados. Essa função, após a determinação do valor do maior caminho de um digrafo acíclico, inicia um processo recursivo onde determina todos os caminhos existentes nesse digrafo com aquele valor.

5.3 Abordagens Existentes

Grande parte dos métodos propostos para a resolução do problema descrito nas seção 5.1 e 5.2, que buscam soluções ótimas, são baseados em técnicas conhecidas como *branch-and-bounds* e talvez um dos mais importantes e expressivos foram desenvolvidos por Carlier e Pinson (1989). A principal desvantagem, típica desse tipo de abordagem, consiste no esforço computacional exigido. O tempo de computação necessário para se determinar uma solução pode atingir níveis intoleráveis, tornando-os, muitas vezes, inviáveis em aplicações práticas.

Como alternativa, atualmente, diversos métodos de aproximação estão sendo propostos. São abordagens que buscam soluções próximas às soluções ótimas, porém, com tempos de processamento viáveis. Entretanto, a grande maioria desses métodos de aproximação requer a incorporação de conhecimentos prévios específicos de cada problema a ser tratado. Exigem do usuário profundos conhecimentos da estrutura combinatorial do problema específico. Essa dependência decorre do fato de serem métodos heurísticos que utilizam regras de prioridades, isto é, regras para escolha de uma

operação de um subconjunto de operações ainda não escalonadas. O método *Shifting Bottleneck Heuristic*, desenvolvido por Adams, Balas e Zawack (1988), é provavelmente o mais poderoso representante desse tipo de abordagem.

Recentemente observa-se um aumento considerado no interesse por abordagens baseadas em técnicas de otimização de busca local para o tratamento do problema da PPJS. Basicamente, essas técnicas consistem no refinamento de uma dada solução que, através de um processo iterativo, é submetida a uma série de alterações simples. Cada alteração realizada resulta em uma nova solução. Esses métodos podem ser vistos como ferramentas de busca que atuam em um espaço de soluções viáveis, procurando por melhores soluções, dentro de limitações de tempos razoáveis. É fundamental ressaltar que, nesse tipo de abordagem, o importante é desenvolver técnicas para a localização rápida de soluções de alta qualidade que se encontram em espaços de soluções imensos e complexos, porém, sem oferecer qualquer garantia de otimização.

Em situações reais, devido as suas características genéricas, os métodos de otimização de busca local podem se apresentar como uma ótima opção, além de que esse tipo de abordagem geralmente é de fácil implementação. Figuram nesta categoria, dentre outros, os métodos baseados nos algoritmos genético, *simulated annealing* e *tabu search*. Uma descrição detalhada de cada um desses métodos pode ser encontrada em Cândido (1998), sendo que os dois primeiros também foram descritos no capítulo 4 do presente trabalho. Igualmente, Blazewic, Domschke e Pesch (1996) fazem um estudo detalhado de todas as abordagens existentes dedicadas ao problema da PPJS. Eles concluem que os métodos de otimização de busca local constituem-se nas melhores ferramentas disponíveis para esse tipo de problema. Destacam, entretanto, que qualquer método baseado puro e simplesmente em um desses algoritmos, não podem competir com aqueles métodos que incorporam conhecimentos específicos do problema.

É importante salientar a diferença fundamental entre os algoritmos AG, SA e *tabu search*. O primeiro trabalha sobre uma população de soluções ao invés de soluções individuais. Os outros dois têm como base sempre uma única solução e buscam por soluções melhores na vizinhança desta.

Uma característica peculiar com relação ao algoritmo *tabu search* é a construção de uma lista de movimentos proibidos, isto é, há transições que não são permitidas em determinadas iterações. As mais recentes e eficientes abordagens para o problema PPJS, com base em *tabu search,* foram propostas por Dell'Amico e Trubian (1993) e Barnes (1994) e Balas e Vazacopoulos (1995). Alguns métodos relevantes, para o mesmo problema, que utilizam AG, podem ser investigados em Dorndorf e Pesch (1995), Della Groce et al. (1995), Aarts et al. (1994) e Nakano e Yamada (1991). Os mais recentes e representativos enfoques do problema PPJS, com base no algoritmo SA, foram propostos por, Van Laarhoven et al. (1992) e Lourenço (1995).

5.4 Uma Nova Abordagem

5.4.1 Introdução

A proposta inicial deste trabalho foi construir um algoritmo paralelo para a resolução de problemas da PPJS, que tivesse como base o algoritmo genético. A principal meta era explorar o paralelismo implícito desse algoritmo em um ambiente distribuído. O capítulo 4 foi produto do estudo realizado em sistemas distribuídos para esse fim. Entretanto, ainda durante os testes da implementação da sua versão seqüencial, pode-se constatar que o mesmo não apresentava potencialidade suficiente para resolver problemas desse tipo. Seu desempenho ficou aquém das expectativas, não conseguindo obter soluções suficientemente próximas das ótimas, como se esperava.

Na tentativa de melhorar o seu desempenho, sem recorrer a métodos heurísticos, desenvolveu-se um novo modelo, no qual a operação de mutação passou a ser realizada por um algoritmo do tipo *simulated annealing*. Em contraste com o operador clássico de mutação em que, com pequena probabilidade, alguns genes de uma nova solução são aleatoriamente alterados, esse operador passou a ter características bastante peculiares. A realização de uma mutação sobre uma solução passou a ser realizada empregando o algoritmo SA, de maneira tal que a solução resultante fosse aceita em função de sua

qualidade em relação à solução original e da geração corrente. Dessa forma, essa operação que antes alterava uma solução aleatoriamente, passou a ter um certo controle através de um processo estocástico, proveniente do algoritmo SA.

Os resultados tiveram uma melhora significativa, porém, não suficiente para se concluir o estudo. Após 4 meses de pesquisas onde foram realizados testes com os mais diferentes tipos de operadores de *crossover* encontrados na literatura, decidiu-se por abandonar a idéia original e buscar novos rumos para a pesquisa. Entretanto, a idéia da combinação desses dois algoritmos persistiu, desta feita, porém, com a atenção voltada para o algoritmo SA.

O produto final acabou por se constituir em um método de otimização de busca local, no qual se tem o algoritmo SA como base e o AG como uma complementação. A forma como esses dois algoritmos foram combinados é descrita a seguir.

5.4.2 Descrição do Método

Em um projeto que envolva a aplicação do algoritmo SA, além da definição do conjunto de parâmetros específicos do problema a ser tratado, como visto anteriormente, também devem ser determinados os parâmetros associados com o esquema de controle da temperatura. A maioria desses projetos adota esquemas nos quais a temperatura T é retirada de uma faixa definida pelos limites superior (valor inicial da temperatura) e inferior. Uma dada temperatura é mantida constante por um número predeterminado de repetições e, então, multiplicada por um valor real denominada fator de redução. O processo se repete até T atingir o limite inferior da faixa. Apesar de não garantir qualquer convergência assintótica para uma solução ótima global, esses esquemas parecem funcionar muito bem na prática. A figura 5.2 representa um esboço simplificado de um algoritmo SA básico. Como já apresentado no capítulo 3, trata-se de um algoritmo iterativo que, em cada passo, gera uma nova solução que pode se tornar a solução corrente. Essa condição de troca de soluções tem probabilidade igual a 1, caso a solução gerada seja melhor que a corrente. Caso contrário, a probabilidade passa a valer $e^{-DIF/T}$.

Onde *DIF* representa a diferença entre as duas soluções consideradas. Uma troca efetuada na segunda condição é conhecida como movimento de *upphill* e é utilizada na tentativa de se livrar de uma solução ótima local, mantendo a exploração naquela região do espaço de soluções. A probabilidade da ocorrência de um movimento *upphill* é controlada pela temperatura *T* que é gradualmente reduzida de um valor alto, onde a maioria desses movimentos são aceitos, até um valor baixo, onde tais movimentos raramente são aceitos.

A estrutura básica do método desenvolvido é um algoritmo SA tal como o apresentado na figura 5.2.

```
obtenha uma solução inicial S;
obtenha os limites superior e inferior de temperatura LS e LI;
obtenha a constante α e o número de repetições NR;
VS = F ( S ); // VS recebe o valor da avaliação da solução S
T = LS; TMIN = LI;
enquanto ( T > TMIN ) faça
  início
    para I de 1 até NR faça
      início
        SL  = V ( S ); // SL recebe uma solução gerada na vizinhança de S
        VSL = F ( SL );
        DIF = SL - S;
        se ( DIF < = 0 ) então
            S = SL
         senão
            S = P ( SL, T, DIF ); // S recebe SL com probabilidade e^(-DIF/T)
    fim para
    T = T * α
  fim enquanto
```

Figura 5.2 Um esboço do algoritmo SA.

No método aqui proposto, a principal modificação envolve o processo de atualização da solução corrente. Sempre que uma troca é efetivada, uma operação adicional é realizada. Essa operação corresponde a um cruzamento (*crossover*) envolvendo a solução recém gerada e a melhor solução encontrada até então. Isto corresponde, na figura 5.2, introduzir, após o comando *S = P (SL,T, DIF)*, um novo comando *S = CROSS (S, SMAX)*. A função *CROSS( )* é responsável pelo cruzamento da solução *S*, a qual acabou de ser gerada, com a solução *SMAX*, a melhor solução corrente. A maneira como essa função cruza os materiais genéticos de duas soluções para gerar uma terceira é descrito nas seções 5.4.3 e 5.4.4. É importante observar que os elementos genéticos da solução *SMAX*, que no caso do problema de PPJS são arcos, têm grande probabilidade de serem de boa qualidade, uma vez que os mesmos compõem a melhor solução até então encontrada. Parte deste material será enxertado na solução corrente através da operação *crossover*.

5.4.3 Estrutura de Vizinhança

A estrutura de vizinhança utilizada, baseada na proposta apresentada em Van Laarhoven et al. (1992), é de implementação simples e revelou-se como uma das mais eficientes dentre as estruturas analisadas e experimentadas, ver Blazewicz et al. (1996)

Seja uma solução corrente $\Pi_i = \{\pi_{i1}, \pi_{i2}, \ldots, \pi_{im}\}$. Uma transição é efetuada, ou seja, uma nova solução nas vizinhanças de Π_i é gerada, primeiramente, escolhendo no seu digrafo correspondente D_i, os vértices v e w, tal que:

1) v e w sejam operações sucessivas de uma mesma máquina k;

2) o arco $(v, w) \in E_i$ faça parte do maior caminho de D_i;

Após a escolha de um arco (v, w), uma nova solução é obtida pela reversão da ordem na qual as operações, correspondentes v e w , são processadas na máquina k. Dessa forma, no digrafo D_i, uma transição como esta resulta na reversão do arco que conecta v e w e na substituição dos arcos (u, v) e (w, x) por (u, w) e (v, x), respectivamente, sendo

$u = \pi_{i(k-1)}(v)$ e $x = \pi_{ik}(w)$. A figura 5.3 ilustra um processo de reversão de um arco (v, w), considerando 5.3(a) como a situação original e 5.3(b) como a situação final.

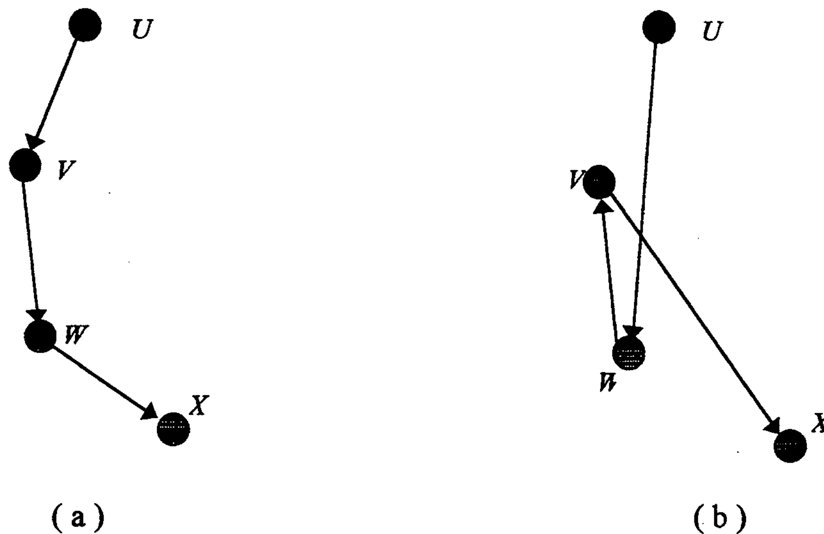


Figura 5.3 Ilustração de um processo de reversão do arco (v, w)

5.4.4 O Operador Genético

Tal como na estrutura de vizinhança, o *crossover*, isto é, a combinação de duas soluções para formar uma terceira, também está baseado na técnica de reversão de arcos pertencentes aos maiores caminhos nos seus digrafos correspondentes, ver Aarts et al. (1994). Sejam as soluções i e j e seus respectivos digrafos D_i e D_j. Um cruzamento é realizado quando for possível reverter alguns arcos do maior caminho de D_j, tal que os arcos resultantes pertençam a D_i. Em outras palavras, é escolhido aleatoriamente um arco (v, w), do dígrafo D_i. Se o arco (w, v) ocorrer em D_j e pertencer ao seu maior caminho, ele é revertido e um terceiro digrafoD_{j1} é gerado. Uma nova escolha é realizada, mas agora envolvendo os digrafos D_i e D_{j1}. Este processo se repete até que nenhum arco possa mais ser invertido. Pode-se afirmar, então, que o resultado de um cruzamento é uma solução obtida da solução j, através da implantação de um sub conjunto de arcos de D_i em D_j.

Como visto, a operação utilizada, tanto na estrutura de vizinhança, como no cruzamento de duas soluções, está baseada na reversão de arcos pertencentes a caminhos críticos dos seus digrafos correspondentes. A utilização deste processo com esta característica justifica-se por dois motivos importantes. O primeiro é que algumas transições que não diminuem o valor do maior caminho, ou seja, não melhoram a solução corrente, são antecipadamente excluídas. Esta afirmação é facilmente comprovada, uma vez que se a reversão de uma arco, não pertencente a um caminho crítico de D_i, produzir um digrafo D_j acíclico, um maior caminho q em D_j não pode ser menor do que o maior caminho p em D_i, uma vez que D_j continua contendo o caminho p. O segundo motivo é que, como mostra o teorema 5.1, a reversão de um arco com esta característica em um digrafo acíclico produz sempre um segundo, também acíclico. Conseqüentemente, o algoritmo trabalha sobre um espaço de soluções somente viáveis.

Teorema 5.1

Seja $e = (v, w) \in E_i$ um arco de um caminho crítico de um digrafo acíclico D_i e seja D_j um digrafo obtido de D_i através da reversão do arco e. Então, D_j é também acíclico.

Prova do teorema 5.1

Por hipótese, considera-se que D_j seja um digrafo cíclico. Uma vez que D_i é acíclico, o arco (w, v) encontra-se no ciclo produzido em D_j. Conseqüentemente, um caminho $(v, x, y, ..., w)$ em D_j é criado. Mas este caminho deve também existir em D_i e, naturalmente, é um caminho maior entre os vértices v e w do que o arco (v, w). Esta conclusão, entretanto, contradiz a hipótese de que (v, w) pertence a um maior caminho em D_i. Dessa forma, prova-se por contradição que D_j é um digrafo acíclico.

É importante notar que em cada operação de cruzamento pode ocorrer uma série de reversões, porém, como todas envolvem arcos tomados em caminhos críticos, o resultado é sempre uma solução viável.

5.4.5 O Controle da Temperatura

Para o controle da temperatura, um novo esquema foi proposto. Ao invés dos valores da temperatura serem retirados de uma única faixa, como na maioria dos projetos envolvendo SA, propõem-se a existência de múltiplas faixas. Inicia-se o processo com uma faixa e seus dois extremos: limite superior, *ls*, limite inferior, *li* >> *0* e três constantes: α, β e δ. Em cada faixa o valor da temperatura é reduzido de *ls* até *li* utilizando $t = t^*\alpha$. Após a conclusão da varredura dos pontos ao longo de uma faixa, a temperatura salta abruptamente para o limite superior da nova faixa, obtido através de $ls = ls^*\beta$. O novo limite inferior é calculado por $li = li^*\delta$.

Todos os parâmetros têm seus valores empiricamente atribuídos e estão relacionados pela expressão:

$$p1 = exp\,\{(\ln(p2/p3))/p4\} \qquad (5.4)$$

Onde *p1* representa uma das constantes: α, β e δ; *p2* o limite inferior das faixas; *p3* o limite superior das faixas e *p4* o número de pontos tomados em cada faixa ou o número total de faixas.

Essas perturbações, causadas pelos aumentos bruscos nas temperaturas nos limites inferiores de cada faixa, têm como objetivo procurar evitar convergência para solução ótima local. Elas permitem a realização de modificações acentuadas nas soluções correntes. É uma estratégia simples que procura desviar a busca de regiões já intensamente exploradas, dirigindo a procura para uma nova região do espaço de soluções. Como visto anteriormente, o algoritmo SA utiliza os movimentos conhecidos por *uphill* com este mesmo objetivo, porém, sem a intenção de deixar a região de busca corrente.

Em Lourenço (1995), é proposta uma abordagem que utiliza uma outra estratégia, cujo objetivo é também abandonar regiões do espaço de soluções intensivamente exploradas. A estratégia foi originalmente proposta por Martin, Otto e Felten (1992) e as abordagens que a utilizam são conhecidas por métodos de otimização

de passo largo. Uma abordagem deste tipo se caracteriza, basicamente, por executar, em cada iteração, um passo largo, seguido da execução de um método de otimização de busca local. O algoritmo empregado no passo largo fica responsável pela modificação acentuada realizada na solução corrente. Cabe ao método de otimização de busca local, o refinamento da solução encontrada no passo largo.

Na figura 5.3 é apresentado um esboço geral do método proposto. Trata-se, praticamente, da figura 5.2, apresentada anteriormente, onde foram introduzidos um novo laço, o qual realiza o controle das faixas de temperaturas. O enxerto de material genético da melhor solução *SMAX* na solução corrente *S* é realizado através da função *CROSS(S, SMAX)*, detalhada na seção 5.4.4. Ao final de cada faixa, os comandos $ls = ls*\beta$ e $li = li*\delta$ determinam os novos limites de temperatura para a uma nova faixa, conforme descrição encontrada na seção 5.4.5.

obtenha uma solução inicial S;

obtenha LS, LI, α, β, δ e o número de faixas NF;

VS = F (S); // VS recebe o valor da avaliação da solução S

T = LS;

TMIM = LI;

Para** FAIXA **de** 1 **até** NF **faça

  enquanto** (T > TMIN) **faça

    início

      para** I **de** 1 **até** NR **faça

        início

          SL = V (S); // SL recebe uma solução gerada na vizinhança de S

          VSL = F (SL);

          DIF = SL - S;

          se** (DIF < = 0) **então

            S = SL

          senão

            S = P (SL, T, DIF); // S recebe SL com probabilidade $e^{-DIF/T}$

$S = CROSS(S, SMAX);$

fim para

$T = T * \alpha$

fim enquanto

$LS = LS * \beta;$

$LI = LI * \delta;$

$T = LS;$

$TMIN = LI;$

fim para

Figura 5.4 Um esboço do algoritmo proposto.

5.5 A Versão Distribuída do Método Proposto

5.5.1 Introdução

Devido a natureza seqüencial do algoritmo SA, que constitui a base do método desenvolvido, uma primorosa implementação paralela se torna difícil de ser alcançada. Caso a idéia original do trabalho tivesse sido mantida, essa tarefa seria muito mais natural uma vez que o AG é intrinsecamente paralelo, ver Park e Carter (1995). Essa grande diferença na natureza dos dois algoritmos decorre do simples fato de que o primeiro, como já visto, atua sobre uma única solução, já o segundo com um conjunto delas.

A paralelização de um algoritmo genético pode ser alcançada, por exemplo, distribuindo o trabalho da reprodução de uma população entre os processadores existentes. A paralelização da tarefa que antecede o processo de reprodução, ou seja, a avaliação dos elementos de uma população, também pode ser facilmente alcançada.

No caso do algoritmo SA, vários modelos de paralelização têm sido propostos, porém, com mínimas alterações de sua natureza seqüencial e, conseqüentemente, pouco sucesso com relação ao aumento da velocidade alcançada, ver Casotto et al. (1987), Chen (1998) e Stiles (1993).

5.5.2 O Modelo de Paralelização

Neste trabalho duas tentativas foram realizadas. Na primeira, a tarefa da transição de uma solução para outra foi paralelizada. Uma cópia da solução corrente é distribuída entre os processos que individualmente realizam essa operação. Apesar de utilizarem a mesma estrutura de vizinhança, os resultados são diferentes, pois envolvem processos aleatórios, tanto na escolha do caminho crítico, como na determinação dos arcos a serem invertidos. Cada processo envia sua solução gerada a um processo centralizador que determina a melhor solução dentre todas as produzidas, redistribuindo-a novamente entre eles. O tempo gasto na comunicação entre os processos no envio e recebimento das mensagens inviabilizou este enfoque. Não só o tamanho da mensagem é relativamente grande, uma vez que envolve uma estrutura de dados que compõem cada mensagem, como também a freqüência na troca de mensagens é alta.

O modelo efetivamente implementado é bastante simples. Cada processo executa um algoritmo SA de forma assíncrona, ignorando totalmente a execução dos demais processos. Ao final da sua execução, cada processo envia a sua melhor solução encontrada a um processo centralizador que determina, ao término de todas as execuções, a melhor solução gerada. No modelo proposto, apesar do tamanho da mensagem trocada entre os processos continuar grande, a freqüência nas trocas foi reduzida. Cada processo envia apenas uma mensagem, no final de sua execução, ao processo coordenador. A redução do tempo de processamento alcançada é proporcional ao número de processadores existentes. Esse modelo de implementação tem o seu valor prático, uma vez que, em se tratando de um método aproximado, normalmente o valor final é retirado de uma série de execuções de um mesmo problema.

A plataforma de desenvolvimento utilizada foi um equipamento IBM-SP2, com nove processadores do tipo POWER II, que se comunicam através de um sistema de chaveamento de alta velocidade. Esse ambiente tem como base o sistema AIX da IBM, e

como suporte para programação paralela uma camada chamada PE (*Parallel Environment*).

Na implementação do algoritmo paralelo foi utilizado uma biblioteca de rotinas conhecida por MPI (*Message Passing Interface*), Snir et al. (1996) e Pacheco (1998). Trata-se de um sistema de passagem de mensagens padronizado, que funciona em uma grande variedade de computadores paralelos e cuja utilização vem se disseminando a nível mundial. O modelo de programação utilizado pelo MPI é extremamente simples. Cada processo, identificado por um número inteiro (IP), recebe uma réplica do código do programa paralelo como um todo. O trecho de código particular de cada processo (se houver) é identificado por um comando de seleção onde a condição envolve o seu IP. A seguir, na figura 5.4, é apresentado um esboço de um programa paralelo, onde dez processos trocam mensagens. Cada processo envia uma mensagem para o seu sucessor e recebe outra do seu predecessor. O primeiro processo (*IP = 0*) recebe do último (*IP = 9*). O conteúdo de uma mensagem consiste simplesmente do número de identificação do processo que a enviou. O programa está escrito na linguagem de programação *C* e os comandos em negrito fazem parte da biblioteca *MPI.H*.

```
#include "mpi.h"
#include <stdio.h>
main ( int argc, char **argv )
{   int totalproc, ip, cont, fonte, destino, tag, msgsend, msrec;
1  MPI_init ( &argc, &argv );
2  MPI_TYP_COMMIT( );
3  MPI_Comm_size ( MPI_COMM_WORLD, &totalproc );
4  MPI_Comm_rank ( MPI_COMM_WORLD, &ip        );
5  if ( ip == ( totalproc – 1 ) )
       destino = 0;
   else
       destino = ip + 1;
6  msgsend = ip;
   tag     = 1;
```

```
   cont       = 1;
7  MPI_Send (&msgsend, cont, MPI_INT,destino,tag,MPI_COMM_WORLD );
8   if ( ip == 0 )
       fonte  =  totalproc - 1;
   else
       fonte  =  ip - 1;
9  MPI_Recv  ( &msgrec, cont, MPI_INT, fonte, tag, MPI_COMMWORLD,
                &status );
10 printf( " \n\n PROCESSO : %d    RECEBEU MENSAGEM DE: %d",
             ip, msgrec );
11 MPI_Finalize( );
}
```

Figura 5.5 Um exemplo de um programa paralelo

Todos os dez processos recebem um código idêntico ao apresentado na figura 5.5. Através dos comandos das linhas 3 e 4 cada processo obtém, respectivamente, o número total de processos existentes (*totalproc=10*) e o seu próprio número de identificação (*ip= 0, 1, ... , 9*). Na execução do comando 5, cada processo, com exceção do último, define seu endereço destino para o qual enviará sua mensagem (*destino=ip+1*). O último processo (*ip= totalproc – 1*) envia para o primeiro. O envio da mensagem é realizado através do comando 7 (*MPI_Send*), no qual cada processo especifica a mensagem, que no exemplo é simplesmente o seu *ip*, o tipo da mensagem, o destino, etc. Através do comando 9 (*MPI_Recv*), cada processo aguarda, bloqueado, o recebimento de uma mensagem cuja fonte foi especificada por ele no comando 9. Após o recebimento da mensagem um processo escreve o seu número de identificação (o seu *ip*) e o conteúdo da mensagem recebida (o *ip* do processo que lhe enviou).

No programa paralelo, desenvolvido neste trabalho, cada processo recebe um código que consiste, basicamente, na implementação do algoritmo apresentado na seção 5.3. Após a execução do programa, cada processo envia a um coletor (processo de número zero), através de um comando *MPI_Send*, a melhor solução por ele encontrada.

O processo coletor, também executa o programa básico comum a todos, porém, o seu código de encerramento é diferente. Em uma estrutura de repetição ele aguarda as 8 mensagens correspondentes às soluções encontradas pelos demais processos. Termina sua execução, escolhendo entre todas, aquela que apresentar o menor valor

5.6 Análise dos Resultados Computacionais

Durante o desenvolvimento do sistema, todos os testes realizados utilizaram o *benckmark* proposto por Fisher e Thompson (1963). Trata-se do notório problema envolvendo 10 máquinas e 10 *jobs* que permaneceu insolúvel por mais de 20 anos. Embora, atualmente não seja mais um desafio, este teste é uma forma simplificada de se apurar a potencialidade de um método proposto. Naturalmente, para se conseguir comparações mais detalhadas entre métodos (se é que isto seja possível, principalmente quando se envolvem máquinas e ambientes diferentes), apenas este não é suficiente. Existem muitos outros problemas *benchmarks* disponíveis na bibliografia do assunto, alguns com soluções ótimas ainda desconhecidas. Nos testes finais foram utilizados, além do de Fisher e Thompson, alguns *benckmarks* proposto por Taillard (1993). Este último fornece o algoritmo gerador dos testes, permitindo, assim, que os interessados criem seus próprios problemas.

Na tabela 5.1 estão relacionados os resultados referentes a problemas propostos por Fisher e Thompson. FIS1 simboliza o problema das 10 máquinas e 10 *jobs* aludido anteriormente, FIS2 é um outro problema proposto por eles envolvendo 20 máquinas e 5 *jobs*. Para cada método comparado são tabelados: a média dos valores encontrados em cinco rodadas (VM), o desvio padrão (DP), a média dos tempos de processamento em segundos (TM) e o melhor resultado alcançado (MV). Foram tabelados os valores obtidos por Laarhoven et al. (1992), Lourenço (1995) e pelo método desenvolvido neste trabalho, na sua forma seqüencial, referenciado por SAAGS (*simulated annealing* algoritmo genético seqüencial). Os três métodos relacionados possuem em comum a utilização do algoritmo SA como base, porém, com estruturas gerais diferenciadas. O

método desenvolvido em Laarhoven et al. (1992) emprega um sofisticado sistema de controle de temperatura que garante uma convergência assintótica para uma solução ótima global, porém, a um custo alto com relação ao tempo de processamento. Em Lourenço (1995), é apresentado um método de otimização de passo largo, de acordo com o descrito na seção 5.4.5, onde o algoritmo SA é utilizado no refinamento de soluções.

Tabela 5.1 Resultados dos problemas *benchmarks* de Fisher e Thompson (1963)

Pro-blema	Laarhoven et al. (1992)				Lourenço (1995)				SAAGS			
	VM	DP	TM	MV	VM	DP	TM	MV	VM	DP	TM	MV
FIS1	933,4	3,1	57.772	930*	—	—	101.000	937	934,0	3,6	25.000	930*
FIS2	1.173,8	5,2	62.759	1.165*	—	—	—	—	1.177,0	2,3	14.000	1.173

Os números tabelados em 5.1, seguidos de um asterisco, correspondem ao valores ótimos dos testes correspondentes. Nos casos referentes a Laarhoven et al. (1992) e SAAGS, foram realizados cinco testes para cada problema e o valor ótimo do problema FIS1, foi obtido três e duas vezes, respectivamente.

Apesar dos resultados relacionados na tabela 5.1 terem sido produzidos por métodos de otimização de busca local e apresentarem em comum a utilização do algoritmo SA como base, a relação entre os tempos médios de processamento não exprime significado muito importante. Eles foram rodados em máquinas e sistemas operacionais diferentes. No caso de Laarhoven et al. (1992), foi utilizado um sistema VAX-785. Lourenço (1995) utilizou a linguagem C, em uma estação de trabalho do tipo *Sparcstation* 1, sistema operacional UNIX. O método desenvolvido neste trabalho (SAAGS) foi programado na linguagem C, em um computador pessoal com processador tipo Pentium 200, no ambiente *Windows*.

Na tabela 5.2 estão relacionados os resultados referentes a problemas propostos por Taillard (1993). Para cada método comparado é tabelado o melhor valor obtido. Nas três últimas colunas encontram-se tabelados os valores obtidos pelo próprio Taillard (1993), por Mattfeld (1996) e SAAGS, respectivamente. As duas primeiras colunas informam o código e a dimensão de cada problema utilizado.

Tabela 5.2 Resultados de problemas *Benchmarks* propostos por Taillard(1993)

Problema	Dimensão	Taillard (1993)	Mattfeld (1996)	SAAGS
Ta01	15X15	1231	1247	1225
Ta02	15X15	1244	1247	1186
Ta03	15X15	1223	1221	1201
Ta04	15X15	1181	1181	1188
Ta05	15X15	1234	1233	1235
Ta06	15X15	1243	1178	1247
Ta07	15X15	1228	1228	1234
Ta08	15X15	1221	1296	1207
Ta09	15X15	1289	1296	1219
Ta10	15X15	1261	1255	1222
Ta11	20X15	1376	1411	1402
Ta15	20X15	1366	1391	1371
Ta17	20X15	1480	1496	1454
Ta26	20X20	1679	1700	1663
Ta32	30X15	1853	1894	1823

Os testes relacionados na tabela 5.2 foram escolhidos, principalmente, por se tratarem de problemas com dimensões maiores do que os normalmente publicados e, desta forma, mais próximo dos problemas reais da programação da produção industrial. Não foram tabelados os valores e tempos médios de processamento por dois motivos principais. No trabalho de Taillard (1993), onde estes testes encontram-se publicados não

foram incluídos essas estatísticas. O outro motivo é decorrente do fato de que os métodos relacionados são conceitualmente distintos. A abordagem utilizada por Taillard é baseada em *tabbo search*. Em Mattfeld (1996) fui empregado um algoritmo genético híbrido, onde regras heurísticas são utilizadas.

Com relação ao método SAAGS, para cada teste, o melhor valor foi retirado de cinco rodadas consecutivas. Já em Mattfeld (1996), para cada teste, esses valores foram retirados de um ensaio com trinta rodadas. No caso de Taillard (1993), esta informação não encontra-se disponível.

Finalizando esta seção, são analisados os resultados obtidos com a versão distribuída do método desenvolvido neste trabalho, referenciado por SAAGD (*Simulated Annealing* Agoritmo Genético Distribuído). Em Krishna et al. (1995), foi desenvolvido um método que emprega o algoritmo SA de forma distribuída, em problemas da produção do tipo *Job Shop*. Na realidade, foram propostas duas versões diferentes do mesmo método. Um deles procura dividir o espaço de soluções de um problema e, então, distribuir as partições obtidas entre os vários processadores integrantes de uma rede. Essa divisão do espaço de soluções é conseguida através da fixação de arcos. São escolhidos alguns arcos cujas orientações permanecem inalteradas. Esta versão foi referenciada pelos autores por LE (*Locking Edges Algorithm*). A outra versão, referenciada por TMA (*Temperature Modifier Algorithm*), basicamente, implementa a mesma idéia considerada na implementação do SAAGD. Cada processador executa um algoritmo *simulated annealing* independentemente dos demais e, somente no final, entrega a melhor solução encontrada a um processador central que escolhe a melhor de todas as recebidas. A principal diferença nos dois modelos de paralelismo é que no caso do TMA cada processador executa com temperatura inicial e modificador diferente dos demais.

A tabela 5.3 relaciona os resultados obtidos por LE, TMA e SAAGD, na resolução dos problemas *benckmarks* proposto por Fisher e Thompson (1963). Os testes são os mesmos apresentados na tabela 5.1, ou seja, FIS1 (10 máquinas e 10 *jobs*), FIS2 (20 máquinas e 5 *jobs*). São tabelados, para cada método considerado, a média das melhores soluções encontradas VM e a média dos tempos de processamento gasto em segundos TM.

Tabela 5.3 Resultados dos problemas *benchmarks* de Fisher e Thompson (1963)

Proble-ma	Krishna et al. (1995) LE		Krishna et al. (1995) TMA		SAAGD	
	VM	TM	VM	TM	VM	TM
FIS1	983,3	2869	985	1.800	935,2	8.184
FIS2	1.217,5	7.711,6	1.227	18.200	1.178	8.184

Os métodos LE e TMA foram implementados em um ambiente denominado *Distributed Task Sharing System* (DTSS), projetado especialmente para implementação e análise de algoritmo distribuído. Este ambiente integra três estações de trabalho do tipo Sun 3/60, que atuam como servidores e quinze Sun 3/50 que constituem o conjunto de clientes. Todas as estações estão conectadas através de uma rede do tipo ETHERNET.

O método SAAGD foi implementado, como referido na seção 5.5.2, em um computador SP2, composto por nove processadores do tipo *power* II, interligados por um sistema de chaveamento de alta velocidade. Nos testes com o problema FIS1 foi alcançada uma vez a solução ótima, ou seja, 930. Em todas as rodadas efetuadas com o teste FIS2 foi obtida solução com valor 1.178 (o valor da solução ótima corresponde a 1.165). Com relação ao tempo de processamento, é importante frisar que a utilização do computador durante os testes não foi dedicada, ou seja, o mesmo estava sendo compartilhado por outros usuários.

Em todos os seis testes relacionados na tabela 5.3, os valores médios apresentados foram calculados considerando cinco rodadas.

CAPÍTULO 6
Considerações Finais

São destacados, neste capítulo, a relevância do tema abordado no presente trabalho e a colaboração que o mesmo oferece através dos itens de originalidade aqui propostos e desenvolvidos. Também são sugeridos alguns tópicos que podem ser explorados em futuras pesquisas.

6.1 Conclusões

Dentre os principais fatores que compõem o controle e o gerenciamento da produção industrial encontra-se a programação da produção. Este fator, especialmente em se tratando de programação da produção do tipo *job shop* (PPJS), se traduz em uma tarefa árdua devido a sua grande complexidade. Trata-se de um dos problemas mais difíceis de otimização combinatorial conhecidos.

Atualmente observa-se 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, em um tempo de processamento aceitável.

Neste trabalho, foi proposto e investigado a potencialidade de mais um método geral, baseado na combinação dos algoritmos *simulated annealing* (SA) e genético (AG). O termo "geral" aqui empregado denota a independência do método com relação

ao problema que está sendo tratado. A grande maioria dos métodos de aproximação existente utiliza regras heurísticas de prioridade durante o processo de escalonamento. Esta característica, que obriga a incorporação de conhecimentos particulares do problema ao método de resolução, torna-o "dependente" do problema.

Os algoritmos SA e AG têm sido vastamente utilizados na resolução de problemas de otimização combinatorial de maneira geral. No caso específico do problema PPJS, normalmente, eles não são empregados na sua forma original. Devido ao alto grau de dificuldade na resolução desse tipo de problema, as abordagens que os utilizam geralmente propõem formas diferentes para complementá-los. Três propostas originais com este objetivo foram desenvolvidas neste trabalho.

A primeira, que foi o motivo original deste trabalho, considera o AG como base e emprega o SA na realização da operação de mutação. Em contraste com o operador clássico de mutação em que, com pequena probabilidade, alguns genes de uma nova solução são aleatoriamente alterados, esse operador passou a ter características bastante peculiares. A realização de uma mutação sobre uma solução passou a ser realizada empregando o algoritmo SA, de maneira tal que a solução resultante seja aceita em função de sua qualidade em relação à solução original e da geração corrente. Dessa maneira, essa operação que antes alterava uma solução aleatoriamente, passou a ter um certo controle através de um processo estocástico, proveniente do algoritmo SA.

A segunda proposta apresenta o SA como o algoritmo base e utiliza a operação de *crossover* do AG como uma complementação. Sempre que uma transição de uma solução para outra é efetivada, através do algoritmo SA, uma operação adicional é realizada. Essa operação corresponde a um cruzamento (*crossover*) envolvendo a solução recém gerada e a melhor solução encontrada até então. Os elementos genéticos da melhor solução corrente, que no caso do problema de PPJS são arcos, têm grande probabilidade de serem de boa qualidade. Parte desse material será, então, enxertado na solução que acabou de ser gerada, através da operação de *crossover* proveniente do AG.

A terceira e última proposta desenvolvida neste trabalho está relacionada com o controle de temperatura. Em todo projeto que envolve a aplicação do algoritmo SA, deve ser definido o esquema de controle da temperatura. A maioria desses projetos adota esquemas, no qual, normalmente, as temperaturas são retiradas de uma faixa definida

pelos limites superior (valor inicial da temperatura) e inferior. Uma temperatura t é mantida constante por um número predeterminado de repetições e, então, multiplicada por uma constante real denominada fator de redução. Este processo se repete até a temperatura atingir o limite inferior correspondente ao final da sua faixa. Pelo esquema proposto, ao invés dos valores da temperatura serem retirados de uma única faixa, passaram a existir múltiplas faixas. Inicia-se o processo com uma faixa e seus dois extremos: limite superior, ls, limite inferior, $li >> 0$, e três constantes: α, β e δ. Em cada faixa o valor da temperatura é reduzido de ls até li utilizando $t = t*\alpha$. Após a conclusão da varredura dos pontos ao longo de uma faixa, a temperatura salta abruptamente para o limite superior da nova faixa, obtido por $ls = ls * \beta$. O novo limite inferior é calculado por $li = li *\delta$. Essas perturbações, causadas pelos aumentos bruscos nas temperaturas nos limites inferiores de cada faixa, têm como objetivo procurar evitar convergência para solução ótima local. Elas permitem a realização de modificações acentuadas nas soluções correntes. É uma estratégia simples que procura desviar a busca de regiões já intensamente exploradas, dirigindo a procura para uma nova região do espaço de soluções.

A qualidade da abordagem proposta neste trabalho, fruto das inovações realizadas, foi enfatizada através dos testes realizados com problemas *benchmarks* bem conhecidos na bibliografia referente a problema da programação da produção do tipo *job shop*. Entretanto, deve-se ressaltar que o algoritmo aqui proposto pode se adaptar a qualquer problema que possa ser resolvido por otimização combinatorial.

6.2 Propostas de Novas Pesquisas

A idéia original deste trabalho, que foi a utilização do algoritmo *simulated annealing* atuando na operação de mutação de um algoritmo genético, apresentou-se como potencialmente viável. Uma pesquisa mais aprofundada nesta direção poderá resultar em um método de otimização de busca local importante. Esta pesquisa pode-se tornar ainda mais interessante se considerar a possibilidade da criação de um método distribuído com essas características.

Um método de otimização de passo largo, como descrito neste trabalho (seção 5.4.5) é, na realidade, uma junção de dois algoritmos. O algoritmo de otimização de busca local, responsável pelo refinamento de uma solução, é embutido no algoritmo empregado na realização do passo largo. O desenvolvimento de uma abordagem dessa natureza, utilizando como o algoritmo interno o método desenvolvido neste trabalho, é uma proposta interessante para uma pesquisa. O esquema de controle de temperatura poderia ser substituído por algum algoritmo que causasse o mesmo efeito, ou seja, uma alteração radical na solução corrente.

Resultados computacionais atestam que o algoritmo *simulated annealing* aplicado a problemas da programação da produção industrial, consegue encontrar soluções melhores se comparado a outros algoritmos de aproximação recentemente propostos para o mesmo fim. Entretanto, o tempo de processamento necessário para rodar um algoritmo deste tipo em problema desta natureza, é considerado muito alto. Uma alternativa para se tentar reduzir esse tempo de execução é implementá-lo como um algoritmo distribuído, onde várias tarefas possam ser realizadas em paralelo. A natureza seqüencial do algoritmo *simulated annealing* dificulta esse tipo de implementação, porém, não impossibilita. Apesar da existência de algumas propostas neste sentido, como por exemplo a realizada neste trabalho, essa alternativa se constitui em uma área de pesquisa a ser explorada.

Referências Bibliográficas

AARTS, E. H. L., VAN LAARHOVEN, P. J. M., LENSTRA, K. K., ULDER, N. L. J., *A computational Study of Local Search Shop Scheduling*, ORSA Journal on Computing, v. 6, 1994, p. 118-125.

ADAM, N., SURKIS, J., *A comparison of Capacity Planning Techniques in a Job Shop Control System*, Managment Science, v. 23/9, 1977, p. 1011-1015.

ADAMS, J., BALAS, E., ZAWACK, D., *The Shifting Bottleneck Procedure for Job Shop Scheduling*, Managment Science, v. 34, 1988, p. 391-401.

ADSHEAD, N. S., PRICE, D. H. R., *Adaptive Scheduling: The Use of Dynamic Due Dates to Accommodate Dem and Variance in Make-to-Stock Shop*, International Journal of Production Research, v. 16/7, 1988, p. 1241-1258.

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

BAILLIE, C.F., *Comparing Shared and Distributed Memory Computers*, Parallel Computing, North-Holland, v. 8, 1988, p. 101-110.

BAKER, K. R., *Introduction to Sequencing and Scheduling*, New York, JohnWiley, 1974.

BAKER, K. R., BERTRAND, J. W. M., *An Investigation of Due-Date Assignment Rules with Constrainted Tightness*, Journal of Operations Management, v. 1/3, 1981, p. 109-120.

BAKER, K. R., BERTRAND, J. W. M., *A Dynamic Priority Rule for Scheduling Against Due-Dates*, Journal of Operations Management, v. 3/1, 1982, p. 37-42.

BAKER, K. R., *Sequencing Rules and Due-Date Assignment in a Job Shop*,Management Science, v. 30/9, 1984, p. 1093-1104.

BAKER, J. E., *Reducing Bias and Inefficiency in Selection Algorithm*, Proceedings of the Second International Conference on Genetic Algorithms, 1987, p.14-24.

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.

BARNES, J.W., CHAMBERS, J. B., *Solving the Job Shop Scheduling Problem using Tabu Search*, IIE Transactions v. 27/2, 1994.

BELLMAN, R. E., *On a Routing Problem*. Quart. Appl. Math., v. 16, 1958, p. 87-90.

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

BERTRAND, J. W. M., *The Use of Work Load Information to Control Job Lateness in Controlled and Uncontrolled Release Production Systems*, Journal of Operations Management, v. 3/2, 1983, p. 79-92.

BITRAN, G. R., HAAS, E. A., HAX, A. C., *Hierarchical Production Planning: A Two Stage System*, Operations Research, v. 30/2, 1982, p. 232-251.

BIRREL, A. D., NELSON, B. J., *Implementing Remote Procedure Calls,* ACC Transactions on Computing Systems, v. 2, n. 1, 1984, p. 39-59.

BLAZEWICZ, j., DOMSCHKE, W., PESCH, E., *The Job Shop Scheduling Problem: Conventional and New Solution Techniques,* Europen Journal of Operational Research, v. 93, 1996, p. 1-33.

BOWMAN, E. H., *The Scheduling-Sequencing Problem*, Oper. Res., v. 7, 1959, p. 52-63.

BROOMELL, G., HEATH, J. R., *Classification Categories and Historical Development of Circuit Switching Topologies,* ACM Computing Surveys,v. 15, n. 2, 1983, p. 95-133.

BRUNO, G., ELIA, A., LAFACE, P., A Rule-Based System to Schedule Production, IEEE Trans. Comput., v. C-19, n. 7, 1986, p. 32-40.

BRUNS, R., *Direct Chromosome Representation and Advanced Genetic Operators for Production Scheduling,* Proceeding of Third International Conference on Genetic Algorithms, San Mateo, 1989, p. 352-359.

CÂNDIDO, M. A. B., *A Hybrid Approach to Solve Real Make-to-Order Job Shop Scheduling Problems,* Tese de Doutorado PPGEP / UFSC, 1997.

CARLIER, J., PINSON, E., *An Algorithm for Solving the Jop Shop Problem,* Management Science, v. 35, 1989, p. 164 – 176.

CASOTTO, A., ROMEO, F., SANGIOVANNI-VINCENTELLI, A., *A Parallel Simulated Annealing Algorithm for Placement of Macro-Cells*, IEEE Transactions Computer-Aided Design, v. 6, n. 5, 1987, p. 838-847.

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

CHEN, H., FLANN, N. S., WATSON, D. W., *Parallel Genetic Simulated Annealing: A Massively Parallel SIMD Algorithm*, IEEE Transactions on Parallel and Distributed Systems, v. 9, n. 2, 1998, p. 126-136.

CHRISTOFIDES, N., *Graph Theory An Algorithmic Approach*, Academic Press, 1975.

CLEVELAND, G. A., SMITH, S. F., *Using Genetic Algorithms to Schedule Flow-Shop Releases*, Proceedings of the Third International Conference on Genetic Algorithms, San Mateo, 1989, p. 160-169.

COULOURIS, G., ET AL., *Distributed Systems – Concepts and Design*, Addison-Wesley, 1994.

DANIELS, R. L., CHAMBERS, R. J., *Multiobjective Flow-Shop Scheduling*, Naval Research Logistics, v. 37, 1990, p. 981-995.

DARWIN, C., *The Origin of Species*, New York, Mentor Books, 1953.

DAVIS, L. D., *Job Shop Scheduling with Genetic Algorithms*, Proceeding of the 1st International Conference on Genetic Algorithms and their Applications, 1985.

DAVIS, L. D., *Handbook of Genetic Algorithms,* New York, Van Nostrand Reinhold, 1991.

DE JONG, K. A., SPEARS, W. M. , *Using Genetic Algorithms to Solve NP-Complete Problems*, Proceeding of the Third International Conference on Genetic Algorithms, San Mateo, 1989, p. 124-132.

DELLA CROCE, F., TADEI, R., VOLTA, G, *A genetic Algorithm for Job Shop Problem*, Computers & Operations Research, v. 22, 1995, p. 15-24.

DELL'AMICO, M., TRUBIAN, M., *Applying Tabu-Search to the Job Shop Scheduling Problem*, Annals of Operations Research, v. 41, 1993, p. 231-252.

DIJKSTRA, E. W., *Solution of a Problem in Concurrent Programming Control,* Communication of ACM, v. 8, n. 9, 1965, p. 569.

DORNDORF, U., PESCH, E., *Evolution Based Learning in a Job Shop Scheduling Environment*, Computers & Operations Research, v. 22, 1995, p. 25-40.

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

ERCHLER, J., ESQUIROL, P., *Decision-Aid Job Shop Scheduling: A Knowledge Base Approach*, IEEE Conference on robotic and automation, San Francisco, 1986, p. 1651-1656.

FENG, T., *A Survey of Interconnection Networks,* IEEE Computer, v. 12, n. 14, 1992, p. 12-27.

FINKEL, R. A., SOLOMON, M. H., *Processor Interconnection Strategies,* IEEE Transaction on Computers, v. c-29, n. 5, 1980, p. 360-371.

FISHER, H., THOMPSON, G. L., *Probabilistic Learning Combinations of Local Job-Shop Scheduling Rules.* In Industrial Scheduling. J. F. Muth and G. L. Thompson (eds.). Prentice Hall, Englewood Cliffs, N. J. , 1963, p. 225-251.

GAUTHIER, F. A. O., *Programação da Produção: Uma Abordagem Utilizando Algoritmos Genéticos*, Tese de Doutorado, PPGEP / UFSC, 1993.

GELDERS, L. F., Van Wassenhove, L. N. , *Production planning: A review,* European Journal of Operational Research, v. 7, 1981, p. 101-110.

GERSHWIN, S. B., HILDEBRANT, R. R., SURI, R., MITTER, S. K., *A Control Perspective on Recent Trends in Manufacturing Systems*, IEEE Contr. Syst., 1986, p. 3 -15.

GOLDARTT, E. M., *Computerised Shop Floor Scheduling*, International Journal of Production Research, v. 26/3, 1988, p. 443-455.

GOLDBERG, D. E., *Genetic Algorithms in Search Optimization and Machine Learning*, New York, Addison-Wesley, 1989.

GOYAL, S. K., SRISKANDARAJAH, C., *No-Wait Shop Scheduling: Computational Complexity and Approximate Algorithms*, Operations Research, v. 25, n. 4, 1988, p. 220-244.

GRAVES, S. C., *A Review of Production Scheduling*, Operations Research, v. 29/4, março de 1981, p. 647, 675.

GROOVER, M. P., *Automation, Production Systems, and Computer-Integrated Manufacturing*, Prentice-Hall, New Jersey, 1987.

HAJEK, B., *Cooling Schedules for Optimal Annealing*, Mathematics of Operations Research, 13, 1988, p. 311-329.

HANSEN, P. B., *Concurrent Programming Concepts*, Computing Surveys, v. 5, n. 4, 1973, p. 223-245.

HANSEN, P. B., *Joyce - A programming Language for Distributed Systems*, Software-Practice and Experience, v. 17, n. 1, 1987, p. 29-50.

HAN,W., DEJAX, P., *An Efficient Heuristic Based on Machines Workloads for the Flow-Shop Scheduling Problem with Setup and Removal*, Laboratoire Economique, Industriel et Social, Ecole Centrale, Paris, 1991, p. 1-17.

HANSEN, P. B., *Studies in Computational Science - Parallel Programming Paradigms*, Prentice Hall, 1995.

HE, Z., YANG, T., DEAL, D. E., *A Multiple-Pass Heuristic Rule for Job Shop Scheduling with Due Dates*, International Journal of Production Research, 1993.

HOARE, C. A., *Monitors: An Operating System Structuring Concept*, Communication of the ACM, v. 17, n. 10, 1974, p. 549-557.

HOLLAND, J. H., *Adaptation in Natural and Artificial Systems*, Cambridge, MIT Press, p. 211, 1993.

HOU, E. S. H., LI, H. Y., *Task Scheduling for Flexible Manufacturing Systems Based on Genetic Algorithms*, IEEE, 1991, p. 397-402.

HUSBANDS, P., MILL, F., WARRINGTON, S., *Genetic Algorithms, Production Plan Optimization and Scheduling*, Proceedings, 1992, p. 80-84.

HWANG,K., *Advanced Computer Architecture-Parallelism, Scalability, programmability,* McGraw-Hill, 1993.

JACKSON, J. R., *Jobshop-Like Queueing Systems,* Management Sci., v. 10, n. 1, 1963, p. 131-142.

JOHNSON, S. M., *Optimal Two-and-Three Stage Production Schedules with Set-Up Times Included*, Naval Research Logistics Quarterly, v. 1, 1954, p. 61-68.

JOHNSON, D. S., ARAGON, C. R., MCGEOCH, L. A., SCHEVON, C., *Optimization by Simulated Annealing: An Experimental Evaluation (Part I)*. Opns. Res., n. 37, 1993, p. 865-892.

KIM, G. H., LEE, C. S. G., *An Evaluation Approach to the Job-Shop Scheduling Problem*, IEEE, 1994, p. 501-506.

KING, J. R., SPACHIS, A. S., *Scheduling: Bibliography & Review*, International Journal of Physical Distribution and Materials Management, v.10, n. 3, 1980, p. 105-132.

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

KRISHNA, K., GANESHAN, K., JANAKI RAM, D., *Distributed Simulated Annealing Algorihtms for Job Shop Scheduling*, IEEE Transactiions on Systems, Mean, and Cybernetics, v. 25, n. 7, july 1995.

LAW, A. M., *Simulation Series; Part I: Introducing Simulation: A Tool for Analyzing Complex Systems*, Ind. Eng. , 1986, p. 46-63.

LEVITIN, G., RUBINOVITZ, J., *Genetic Algorithm for Linear and Cyclic Assignment Problem, Computers Operations Research*, v. 20/6, 1993, p. 575-586.

LIN, T. T., KAO, C. Y., HSU, C. C., *Applying the Genetic Approach to Simulated Annealing in Solving some NP- Hard Problems*, IEEE Transactions on Systems, Man and Cybernetics, v. 23, n. 6, 1993, p. 1752-1767.

LOURENÇO, H. R., *Job Shop Scheduling: Computational Study of Local Search and Large-Step Optimization Methods*, European Journal of Operational Research, v. 83, 1995, p. 347-364.

MARTIN, O., OTTO, S. W., FELTEN, E. W., *Large-step Markov Chains for TSP Incorporations Local Search Heuristics*, Operations Research Letters, v. 11, 1992, p. 219-224.

MATTFELD, D. C., *Evolutionary Search and the Job Shop*, Physica-Verlag, Heidelberg, 1996.

METROPOLIS, N., ROSENBLUTH, A., ROSENBLUTH, M., TELLER, A., AND TELLER, E.,*Equation of State Calculations by Fast Computing Machines*, Journal of Chemical Physics 21, 1953, p. 1087-1092.

MULLENDER, S. J., ET AL., *Amoeba: A Distributed Operating System for The 1990s*, IEEE Computer, v. 23, maio de 1990, p. 44-53.

NAKANO, R., YAMADA, T., *Conventional Genetic Algotihm for Job Shop Problems*, Proc. 4th. International Conf. On Genetic Algorithms, Morgan Kaufmann, 1991, p. 474- 479.

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

PARK, Y. B., PEGDEN, C. D., ENSCORE, E. E., *A Survey and Evaluation of Static Flow-Shop Scheduling Heuristics*, International Journal of Production Reseach, v. 22, 1984, p. 127-141.

PARK, K., CARTER, B., *On the Effectiveness of Genetic Search in Combinatorial Optimization*, Computer Science Department, Boston University, 1995.

RAMAZANI, R., YOUNIS, N., *Repetitive Pure Flow Shop Problems: A Permutation Approach*, Computers Industrial Engineering, v. 24/1, 1993, p. 125-129.

RANKY, P. G., *Computer Integrated Manufacturing*, Prentice-Hall, New Jersey, 1986.

REED, D. A., FUJIMOTO, R. M., *Multicomputer Networks: Message-Based Parallel Processing*, MIT Press, 1987.

RODAMMER, F. A., WHITE, K. P., *A Recent Survey of Production Scheduling*, IEEE Transactions on Systems, Man, and Cybernetics, v. 118/6, 1988, p. 841-851.

ROY, B., SUSSMANN, B., *Les Problèmes d'Ordonnancement avec Constraints Disjonctives*, SEMA, Note D.S., n. 9, Paris, 1964.

SILBERSCHATZ, A., PETERSON, J. L., *Operating Systems Concepts*, Addison-Wesley, 1994.

SMITH, S. F., FOX, M. S., OW, P. S., *Constructing and Maintaining Detailed Production Plans: Investigations into the Development of Knowledge-Based Factory Scheduling Systems*, AL Magazine, v. 7, n. 4, 1986, p. 45-61.

SNIR, M., OTTO, S.W., HUSS-LEDERMAN, S., WALKER, D. W., DONGARRA, J., *MPI: The Complete Reference*, The MIT Press, Massachusetts, 1996.

STILES, G. S., *On the Speedup of Simultaneously Executed Randomized Algorithms*, IEEE Transactions Parallel and Distributed Systems, 1993.

TAILLARD, E., *Some Efficient Heuristic Methods for The Flow Shop Sequencing Problem*, European Journal of Operational Research, v. 47, 1990, p. 278-285.

TAILLARD, E., *Benchmarks for Basic Scheduling Problems*, European Journal of Operational Research, v. 64, p. 278-285, 1993.

TANESE, R., *Parallel Genetic Algorithm for a Hypercube*, Proceedings of the Second International Conference on Genetic Algorithms and Their Applications, p. 177-183, 1987.

TANESE, R., *Distributed Genetic Algorithms for Function Optimization*, Ph. D. Dissertation, University of Michigan, 1989.

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

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

UCKUM, S., BAGACHI, S., KAWAMURA, K., *Managing Genetic Search in Job-Shop Scheduling*, IEEE Expert, v. 8, n. 5, 1993, p. 15-24.

VAN LAARHOVEN, P. J. M., AARTS, E. H. L, LENSTRA, J. K., *Job Shop Scheduling by Simulated Annealing*, Operations Research, v. 40, n. 1, 1992, p. 113 - 125.

WERNER, F., *On The Heuristic Solution of The Permutation Flow Shop Problem by Path Algorithms*, Computers Operations Research, v. 10/7, 1993, p. 707-722.

WHITLEY, D., STARKWEATHER, T., FUQUAY, D., *Scheduling Problems and Traveling Salesmen: The Genetic Edge Recombination Operator*, Proceeding of the Third International Conference on Genetic Algorithms, San Mateo, 1989, p. 33-140.

YOUN, H. Y., CHEN, C. C. Y., *A Comprehensive Performance Evoluation of Crossbar Networks,* IEEE Transactions on Parallel and Distributes Systems, v. 4, n. 5, 1993, p. 481- 489.