DEPARTAMENTO DE COMPUTAÇÃO

D E C O M

Uma Implementação Híbrida e Distribuída do Problema Job-Shop através dos Algoritmos GRASP e Genético

Wendrer Carlos Luz Scofield – 98.2.4996

Prof. Dr. Carlos Frederico M. da Cunha Cavalcanti
Prof. Dr. Marcone Jamilson Freitas Souza

Relatório Técnico 01 / 2002

U F O P

UNIVERSIDADE FEDERAL DE OURO PRETO

FOLHA DE APROVAÇÃO DA BANCA EXAMINADORA

Uma Implementação Híbrida e Distribuída do Problema Job-Shop através dos Algoritmos GRASP e Genético

Wendrer Carlos Luz Scofield

Monografia apresentada ao Departamento de Computação do Instituto de Ciências Exatas e Biológicas da Universidade Federal de Ouro Preto como requisito da disciplina COM 951 – Projeto Orientado II, do curso de Bacharelado em Ciência da Computação, e aprovada pela Banca Examinadora composta pelos seguintes membros:

Prof. Dr. Carlos Frederico M. C. Cavalcanti
Orientador
Departamento de Computação – UFOP

Prof. Dr. Marcone J. F. Souza
Co-Orientador
Departamento de Computação – UFOP

Prof. Msc. José Américo Trivellato Messias
Examinador
Departamento de Computação – UFOP

Ouro Preto, 19 de setembro de 2002

A Deus, com todo o meu carinho

Agradecimentos

Agradeço ao meu orientador Carlos Frederico e ao meu co-orientador Marcone Souza pelo grande incentivo e a oportunidade oferecida por ambos para o meu enriquecimento teórico e prático com este trabalho que conclui a minha graduação no curso Ciência da Computação.

Agradeço a Deus pela companhia ao longo dessa trajetória.

Agradeço também, em especial, a minha mãe e a minha namorada Daniela Rosa pelo apoio que foi relevante durante o desenvolvimento do projeto e não poderia de deixar de agradecer ao funcionário do laboratório do Departamento de Computação, onde foi desenvolvido o trabalho, Wilson de Souza Filho pela paciência e extrema compreensão.

Resumo

O desafio neste projeto foi produzir soluções tão próximas o quanto possível da solução ótima para o problema *Job-Shop* em um menor intervalo de tempo.

Segundo Ansari e Hou, o problema *Job-Shop* é um problema de alocação de recursos que pertence a classe de problemas *NP-Difícil.* O grande número de variáveis e restrições que podem estar presentes em instâncias do problema *Job-Shop* tornam o problema complexo exigindo um considerável esforço computacional para resolver o problema (Ansari e Hou, 1997).

Para Souza, a classe de problemas *NP-Difícil* possui uma complexidade alta, exponencial, e a conquista de soluções qualificadas para um problema desta classe, sem garantir a solução ótima, é perfeitamente desejável quando não temos os recursos exigidos e os tempos computacionais são inviáveis durante a execução do problema para obter a solução ótima (Souza, 2001).

O problema *Job-Shop* foi alvo neste projeto para implementação de algoritmos que adotam técnicas de alto desempenho para solucionar problemas dessa natureza. As técnicas de alto desempenho adotadas para o problema foram as técnicas de otimização algoritmo genético e GRASP - *Greedy Randomized Adaptive Search Procedure*, e a programação paralela.

O algoritmo genético foi a técnica de otimização base implementada nos algoritmos distribuídos e seqüencial desenvolvidos para o problema e, segundo Souza, os algoritmos genéticos procuram imitar o mecanismo de *seleção natural* para formar uma heurística capaz de encontrar soluções em tempo hábil para um problema. De acordo com Hou, Ren e Ansari, as soluções são aprimoradas de geração para geração e, em geral, convergem para uma solução qualificada em um tempo finito, porém, não há garantias de que isto ocorra (Souza, 2001; Hou, Ren e Ansari, 1992).

Os algoritmos genéticos têm como característica fundamental a preservação da qualidade das soluções durante a evolução (Souza, 2001). Em termos de tempos computacionais e memória requisitada para evoluir uma população de soluções, a cada geração, para instâncias de problemas de tamanho real, essa técnica de otimização pode se tornar inviável sem a presença de recursos computacionais capacitados. Em virtude disso é que procuramos, através da programação paralela, diminuir o tempo de resposta e distribuir a carga computacional apresentada pelo algoritmo seqüencial. Isso permitiria tratar instâncias do problema de grande porte sem perder a qualidade dos resultados apresentados pelo algoritmo genético.

Foram desenvolvidos neste projeto três modelos de algoritmos distribuídos que paralelizaram o algoritmo genético utilizando o sistema PVM - *Parallel Virtual Machine* (Geist, Beguelin, et. al., 1994) para implementação do paralelismo e configuração do ambiente distribuído. A configuração de um ambiente distribuído para execução dos algoritmos distribuídos fez com que a necessidade de recursos como memória fossem distribuídos entre os computadores que constitui o ambiente.

Palavras-chave

Job-Shop – NP-Difícil – algoritmos genéticos – seleção natural – geração – GRASP – técnicas de otimização – programação paralela – PVM – Parallel Virtual Machine – ambiente distribuído – algoritmos distribuídos – algoritmo seqüencial.

Índice

Lista de Figuras

Lista de Tabelas

1 Introdução

Segundo Walter, o escalonamento das atividades é uma das tarefas que compõem o planejamento da produção denominado programação da produção. Na programação da produção são levados em consideração uma série de elementos que disputam vários recursos por um período de tempo, recursos esses que possuem capacidade limitada (Walter, 1999).

Escalonamento são, para Pinedo e Chao, formas de tomada de decisão que possuem um papel crucial nas empresas, tanto de manufatura como de serviços. No atual ambiente competitivo, o efetivo escalonamento se tornou uma necessidade para sobrevivência no mercado. Companhias devem se esforçar ao máximo para cumprir as datas firmadas com seus clientes. O fracasso deste comprometimento pode resultar em uma perda significante da imagem da empresa perante os clientes (Pinedo e Chao,1999).

Walter ainda relata que, os elementos a serem processados são chamados de ordens de fabricação ou *jobs* e são compostos de partes elementares chamadas tarefas ou operações. Os principais objetivos tratados no problema de escalonamento podem ser resumidos no atendimento de prazos ou datas de entrega, na minimização do tempo de fluxo dos estoques intermediários e na maximização da utilização da capacidade disponível, ou mesmo na combinação desses objetivos (Walter, 1999).

A maioria dos problemas de programação da produção estudados aplica-se ao ambiente conhecido como *Job-Shop.* (Hax e Candea, 1984).

De acordo com Ansari e Hou, o problema *Job-Shop* (JSP) é um problema de alocação de um conjunto de *jobs* para as máquinas, de tal forma que os *jobs* sejam executados em um menor intervalo de tempo. Cada *job* pode consistir de diversas tarefas e cada tarefa deve ser processada numa máquina particular. Além disso, as tarefas em cada *job* estarão sujeitas à restrições de precedência (Ansari e Hou, 1997).

O problema *Job-Shop* é considerado um problema de otimização combinatória de complexidade *NP-Difícil*, ou seja, não existe algoritmo que o resolva em tempo polinomial. (Souza, 2001). Para Ansari e Hou, o tamanho do espaço de soluções para o problema *Job-Shop* cresce exponencialmente com o tamanho do problema, logo o tempo de execução do algoritmo de otimização pode alcançar níveis inviáveis quando é desejado obter a solução ótima para o problema. Em virtude desse longo tempo de execução torna-se conveniente obtermos *boas* soluções em menores intervalos de tempo sem exigir altos recursos computacionais, especialmente se tratarmos de problemas de tamanhos consideráveis e com elevado grau de restrições. (Ansari e Hou, 1997).

Os métodos heurísticos, como algoritmos genéticos e GRASP - *Greedy Randomized Adaptive Search Procedure*, segundo Souza, são métodos de aproximação de característica genérica aplicados aos diversos problemas de otimização combinatória que procuram obter "boas" soluções a um custo computacional razoável. (Souza, 2001).

Tais métodos heurísticos de otimização foram combinados para resolução do problema *Job-Shop*.

Os algoritmos genéticos fundamentam-se em uma analogia com processos da evolução biológica natural, nos quais, dada uma população, os indivíduos com características genéticas melhores têm maiores chances de sobrevivência e de produzirem filhos cada vez mais aptos, enquanto indivíduos menos aptos tendem a desaparecer ao longo das gerações (Souza, 2001). O algoritmo heurístico GRASP foi utilizado com o intuito de gerarmos indivíduos qualificados evitando a geração de vários indivíduos de pouca qualidade durante a criação da população inicial a ser evoluída pelo algoritmo genético.

No projeto desenvolvido por Scofield foi implementado um algoritmo híbrido e seqüencial para o problema *Job-Shop* utilizando as técnicas de otimização GRASP e algoritmos genéticos. Uma vez que as técnicas de otimização já se encontram implementadas para o problema, portanto, investimos todas as pesquisas no desenvolvimento de versões do algoritmo híbrido que adotam a programação paralela em conjunto com as técnicas de otimização já implementadas. Detalhes de implementação dos algoritmos genéticos e GRASP não serão abordados neste projeto, que será uma continuação do projeto desenvolvido por Scofield (Scofield, 2001).

Uma vez que os algoritmos genéticos trabalham sobre uma população de soluções e não sobre uma única solução, torna-se interessante explorar o paralelismo implícito desses algoritmos em um ambiente distribuído. Através da paralelização, é possível diminuir o tempo de resposta e tratar problemas maiores usando os algoritmos genéticos sem perder a qualidade dos resultados. Com esse objetivo foi proposto e implementado versões distribuídas para o algoritmo seqüencial e híbrido desenvolvido, paralelizando o algoritmo genético que é a técnica base implementada para a resolução do problema *Job-Shop*.

Foram avaliados os resultados obtidos dos algoritmos distribuídos por meio de testes computacionais comparando os algoritmos distribuídos e seqüencial.

O sistema PVM - *Parallel Virtual Machine* foi utilizado para a paralelização do algoritmo seqüencial e para a configuração do ambiente distribuído composto de computadores onde executamos os algoritmos distribuídos. Esse sistema foi escolhido por ser público, fácil de utilizar, possuir biblioteca de rotinas que podem ser utilizadas por programas escritos nas linguagens de programação *C* e *C++* e principalmente por permitir a configuração do ambiente distribuído sobre quaisquer computadores disponíveis em laboratórios conectadas por uma rede de comunicação.

No capítulo 2 descrevemos as metodologias para o desenvolvimento deste projeto.

No capítulo 3 descrevemos o problema *Job-Shop*.

No capítulo 4 relatamos as técnicas heurísticas de otimização algoritmos genéticos e GRASP, utilizadas no algoritmo seqüencial para solucionar o problema *Job-Shop* e que foram herdadas pelos algoritmos distribuídos.

No capitulo 5 é abordado uma noção a respeito de computação paralela.

No capítulo 6 expressamos noções de sistemas distribuídos, descrevemos os modelos de algoritmos distribuídos implementados, os testes computacionais realizados e o sistema PVM utilizado para configurar o ambiente distribuído para execução dos algoritmos distribuídos.

Finalmente, no capitulo 7, uma conclusão e proposta para novas pesquisas é apresentado.

No Apêndice A, *Glossário Genético*, mostramos um glossário voltado para a área de algoritmos genéticos definindo termos comuns utilizados pela Biologia e que foram mencionados neste projeto.

2 Metodologias

O desafio neste projeto foi produzir soluções tão próximas quanto possível da solução ótima para o problema *Job-Shop* em um menor intervalo de tempo.

Para tal fim, foram combinadas técnicas de otimização GRASP e algoritmo genético, contribuindo para produção de soluções de qualidade relevantes para o problema. Também foram utilizadas técnicas de paralelização (programação paralela), com o principal intuito de minimizar o tempo e a carga computacional para a produção de tais soluções.

A técnica de otimização GRASP foi implementada para criação das denominadas soluções iniciais requisitadas pelos algoritmos genéticos para o processo evolutivo.

Os algoritmos genéticos têm como forte princípio a qualidade das soluções produzidas ao longo das gerações e não o tempo gasto na evolução dos indivíduos (soluções). Em virtude disso, neste projeto, foram implementadas três versões distribuídas para o algoritmo híbrido e seqüencial já implementado.

O objetivo de pesquisar e desenvolver modelos distribuídos para o algoritmo seqüencial foi avaliar cada versão distribuída comparada a versão seqüencial em relação ao tempo de geração dos indivíduos. O algoritmo seqüencial e híbrido prioriza a qualidade das soluções obtidas em tempo computacional viável. Nesse projeto, não deixaremos a parte tal objetivo, mas, por meio de técnicas de paralelização, também priorizaremos o menor tempo para obter tais soluções de boa qualidade e também a distribuição da carga de dados que demanda as técnicas de otimização aplicadas ao problema.

As versões distribuídas representam o paralelismo adotado na versão seqüencial do algoritmo em três diferentes modelos e não deverão jamais prejudicar a qualidade dos indivíduos gerados durante a evolução imposta pelas técnicas de otimização adotadas.

As versões distribuídas e híbridas para o problema *Job-Shop*, utilizando as técnicas de otimização algoritmo genético e GRASP, foram implementadas na linguagem de programação C/C++ e são apresentadas como produto final de pesquisa neste relatório. Também são apresentados testes e conclusões obtidas comparando os resultados obtidos dos algoritmos na versão seqüencial e distribuída.

A seguir destacamos as principais atividades realizadas no projeto.

1. Recodificação do Algoritmo Seqüencial

Antes de desenvolvermos as versões distribuídas para o algoritmo híbrido que trata o problema *Job-Shop*, foi necessário recodificar o algoritmo seqüencial para a plataforma em que desenvolveríamos os algoritmos distribuídos. Isso, facilitaria compararmos posteriormente os algoritmos e possibilitaria a obtenção de resultados mais concretos com a comparação.

2. Configuração de uma Arquitetura Distribuída

Uma arquitetura distribuída foi elaborada para aplicação dos algoritmos híbridos na versão distribuída.

Foi utilizado o sistema PVM, que possibilitou a configuração de uma arquitetura distribuída simulando uma máquina virtual onde as aplicações distribuídas foram executadas. O sistema também contém rotinas que foram utilizadas nos algoritmos para implementação do paralelismo.

3. Implementação dos Algoritmos Distribuídos

Como foi dito anteriormente, foram desenvolvidas três versões distribuídas para o algoritmo seqüencial e híbrido que trata o problema *Job-Shop* por meio de técnicas de otimização algoritmo genético e GRASP.

Como o algoritmo genético foi a base de implementação do algoritmo seqüencial e híbrido, temos as seguintes versões distribuídas que possuem como base de implementação o algoritmo genético paralelizado :

- **Modelo Centralizado ao Nível de População** – MCPOP – neste modelo o processo mestre mantém a população e envia um ou mais indivíduos para cada processo escravo para que seja gerado os descendentes. A população inicial também é gerada por cada processo escravo. O processo mestre se encarrega de coletar os indivíduos gerados pelos processos escravos e informações deles provenientes.
- **Modelo Distribuído ao Nível de População** – MDPOP – neste modelo a população original é dividida em diversas populações segundo o modelo de ilhas. As evoluções ocorrem em paralelo com trocas dos melhores indivíduos entre as sub-populações vizinhas a cada geração. O modelo MDPOP se comporta como um modelo de difusão quando temos somente um único indivíduo por processo. ou seja, cada indivíduo "vive" no seu local (ilha) e interage com o seu vizinho por meio de operações de cruzamento e mutação, com certa probabilidade, ocorrendo a difusão progressiva de bons indivíduos através de uma rede em *anel unidirecional*. O processo mestre se encarrega de coletar os melhores indivíduos gerados pelos escravos e exibir o melhor entre estes. O processo escravo se encarrega de manter a sua sub-população e evoluí-la ao longo das gerações trocando a cada geração o seu melhor individuo com a sub-população vizinha.
- **Modelo Distribuído ao Nível de Processos** – MDPROC – neste modelo o processo mestre inicia a execução de processos escravos em paralelo, diferentes e independentes entre si. Cada processo escravo é um programa diferente podendo variar entre si parâmetros requisitados pelos algoritmos genéticos. O processo mestre se encarrega de coletar o melhor indivíduo produzido por cada escravo e exibe o melhor entre estes.

As implementações distribuídas foram realizadas no algoritmo genético seqüencial, sem perder sua característica evolucionária, com a intenção de obtermos soluções cada vez mais próximas da solução ótima para o problema *Job-Shop* em menor intervalo de tempo. Outro motivo que nos motivou a paralelizar o algoritmo genético foi a sua natureza paralela implícita, uma vez que este algoritmo heurístico trata uma população de soluções e não uma única solução a cada iteração.

4. Experimentos Computacionais

Os experimentos computacionais foram realizados comparando os tempos gastos para gerar indivíduos a cada geração pelos algoritmos distribuídos e seqüencial. Outras informações também foram obtidas por meio de uma bateria de testes aplicados a cada algoritmo.

3 A Classe do Problema Tratado

Apresentaremos, especificamente, a classe do problema, conhecida como *Job-Shop*, que se trata no projeto. Para tanto, recorreu-se às definições e premissas existentes na teoria clássica da programação.

3.1 Definições e Premissas

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

Segundo Rodammenr e White, 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:

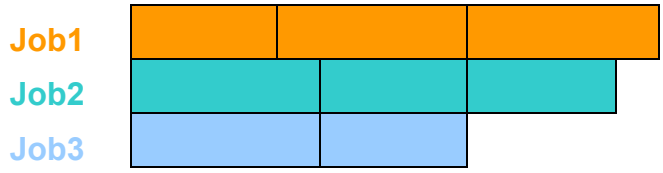
- Operações não podem ser interrompidas e cada máquina pode processar apenas uma operação de cada vez;
- Cada *job* só pode estar sendo processado em uma única máquina de cada vez.
- Cada *job* é fabricado por uma seqüência conhecida de operações.
- Não existe restrição de precedência entre operações de diferentes *jobs*.
- Não existe qualquer relação de precedência entre as operações executadas por uma mesma máquina.

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 (Rodammer e White, 1988).

Normalmente o número de restrições é muito grande para o problema *Job-Shop*. Instâncias do problema que envolva um grande número de restrições demandam consideráveis esforços computacionais para obter a solução ótima (Ansari e Hou,1997).

Um problema *Job-Shop* é normalmente referenciado, de acordo com Ansari e Hou, como um *n/m* JSP, onde *n* é número de *jobs* e *m* o número de máquinas (Ansari e Hou,1997).

Será apresentado na figura 3.1, a seguir, um exemplo ilustrativo de um problema do tipo *Job-Shop*, problema *3/3* JSP, e um escalonamento para o problema na tabela 3.1.



- Cada linha representa um *job* e cada operação X,Y,Z representa uma tarefa desse.
- Para cada operação X,Y,Z :
  - Z – máquina em que a operação deverá ser executada.
  - Y – número da operação.
  - X – *job*.
- O tamanho de cada bloco é proporcional ao tempo de processamento necessário para executar a operação. A tabela a seguir exibe os tempos de processamento de cada operação em unidades de tempo *u*.

Operação (X,Y,Z)	Tempo de Processamento (u)
1,1,2	5
1,2,2	7
1,3,1	7
2,1,3	7
2,2,2	5
2,3,1	5
3,1,2	7
3,2,1	5

Figura 3.1 – Problema 3/3 JSP

A seguir uma solução para o problema *3/3* JSP.

M1	I	3,2,1	I	2,3,1	I	1,3,1
M2	3,1,2	1,1,2	2,2,2	1,2,2		
M3	2,1,3					

Nota: I é intervalo de ociosidade.

Tabela 3.1 – Escalonamento para o problema 3/3 JSP

A tabela 3.1 é o gráfico de Gantt que representa uma das soluções para o problema expresso na figura 3.1. É importante observar que a ordem de processamento necessária das operações em cada *job* foi preservada na solução apresentada. Por exemplo, o *job* 2 foi processado nas máquinas 3, 2 e 1, respectivamente, como especificado na figura 3.1.

A solução exemplo apresentada na tabela 3.1 representa uma seqüência de operações que requer um tempo total de 31 unidades de tempo para completar o processamento de todos os 3 *jobs*.

3.2 Escalonamento num Ambiente Job-Shop

Segundo Souza, o problema de escalonamento *Job-Shop* é classificado na literatura como *NP-difícil*, isto é, é um problema para o qual não existem algoritmos que o resolva em tempo polinomial. Trata-se de um "problema de otimização combinatória" (Souza, 2001).

De acordo com Pacheco e Santoro, os algoritmos de otimização podem ser classificados em duas classes: os modelos matemáticos, que procura a solução ótima, e os heurísticos, que realiza a procura por aproximação em busca de soluções qualificadas sem garantir a solução ótima (Pacheco e Santoro, 1999).

- **Modelos Matemáticos:** Modelo que enfatiza a obtenção de resultados ótimos em função de algum parâmetro de desempenho. Este pode ser, por exemplo, a minimização dos tempos de produção ou a maximização do uso dos recursos. Dependendo da complexidade do problema tratado, pode consumir muito tempo para obter a solução ótima.
- **Modelos Heurísticos:** Modelo que enfatiza a obtenção de soluções qualificadas, próxima da solução ótima, sem, no entanto, garantir a solução ótima. Os modelos híbridos são caracterizados pela obtenção de soluções desejáveis para um problema em tempos de computação viáveis.

Existem também os modelos ponto-a-ponto onde a busca pela melhor solução se processa sempre de um único ponto para outro (inicia-se com um único candidato), no espaço de decisão, através da aplicação de alguma regra de transição, ou seja, o método realiza a busca sempre na vizinhança do ponto corrente, constituindo, portanto, um método de busca local. 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.

A escolha do modelo mais adequado à solução de problemas reais do tipo *Job-Shop* deve levar em consideração alguns aspectos como (Pacheco e Santoro, 1999):

- O número de máquinas envolvidas.
- Os roteiros das ordens de fabricação (*jobs*).
- O regime de chegada das ordens.
- A variabilidade dos tempos de processamento entre outras características.

Devido ao grande número de soluções possíveis e a complexidade do problema de escalonamento, torna-se impraticável modelar todas as variáveis envolvidas no problema utilizando modelos matemáticos. Além disso, o tempo de resposta para os modelos matemáticos eleva-se consideravelmente, tornando impraticável a obtenção de soluções ótimas em tempos viáveis. Dessa forma, algoritmos matemáticos são computacionalmente viáveis quando aplicados a problemas pequenos, com objetivos limitados (Pacheco e Santoro, 1999).

Utilizamos as técnicas heurísticas de otimização GRASP e algoritmo genético com o intuito de obtermos soluções qualificadas, podendo, inclusive, obter a solução ótima, em tempos computacionais viáveis. Abordagens distribuídas foram estudas e implementadas com o objetivo de alcançarmos maior desempenho computacional com a programação paralela utilizando os recursos disponíveis numa rede de comunicação.

4 Algoritmos de Otimização Implementados

As técnicas heurísticos de otimização implementadas no algoritmo seqüencial e que foram herdadas pelos algoritmos distribuídos foram: os algoritmos genéticos e o algoritmo GRASP.

4.1 Os Algoritmos Genéticos

A técnica heurística de otimização algoritmos genéticos foi a técnica base para a implementação do algoritmo seqüencial e também a técnica paralelizada devido a sua natureza paralela implícita que pesquisa uma população de soluções no processo evolutivo e não uma única solução por vez.

Os algoritmos genéticos fundamentam-se em uma analogia com processos da evolução biológica natural, nos quais, dada uma população, os indivíduos com características genéticas melhores têm maiores chances de sobrevivência e de produzirem filhos cada vez mais aptos, enquanto indivíduos menos aptos tendem a desaparecer (Souza, 2001).

De acordo com Neto, tais algoritmos operam numa população de soluções potenciais aplicando o princípio de *ajustamento de sobrevivência*, que possibilita as soluções mais aptas terem maiores condições de serem selecionadas para produzir cada vez mais aproximações melhores para a solução. A cada geração, um novo conjunto de aproximações é criado pelo processo de seleção de indivíduos nos quais são aplicados os operadores de natureza genética de reprodução. Esse processo cobre a evolução de populações de indivíduos que são mais compatíveis com seus ambientes do que indivíduos que os criaram, tal como da adaptação natural (Neto,1997).

Segundo Souza, nos algoritmos genéticos, cada cromossomo representa um indivíduo da população, cada indivíduo representa uma solução do problema associado e cada gene está associado a uma componente da solução. Um mecanismo de reprodução, baseado em processos evolutivos, é aplicado sobre a população com o objetivo de explorar o espaço de busca e encontrar melhores soluções para o problema (Souza, 2001).

Mais especificamente, um algoritmo genético inicia sua busca com uma população $P(0) = \{s^0_1, s^0_2, \ldots, s^0_n\}$, aleatoriamente escolhida, a qual é chamada de população no tempo *0*.

A cada tempo t, existe um mecanismo que permite a renovação da população $P(t+1)=\{s^{t+1}_1, s^{t+1}_2, \ldots, s^{t+1}_n\}$ no tempo $t+1$ a partir de uma população $P(t)$ do tempo t. Nesse processo, existem os chamados operadores genéticos cuja função, é definir regras para uma renovação eficaz de uma população. Para atingir esse objetivo, os indivíduos da população do tempo t passam por uma fase de reprodução, a qual consiste em selecionar indivíduos para operações de cruzamento e/ou mutação. Na operação de cruzamento, os genes de dois cromossomos pais são combinados de forma a gerar um ou dois cromossomos filhos (descendentes), de sorte que para cada cromossomo filho há um conjunto de genes de cada um dos cromossomos pais. A operação de mutação consiste em alterar aleatoriamente uma parte dos genes de cada cromossomo e ocorre com baixas probabilidades normalmente sobre os descendentes. Ambas as operações são realizadas com uma certa probabilidade denominada taxa de cruzamento e taxa de mutação.

Gerada a nova população do tempo $t+1$, define-se a população sobrevivente, isto é, as n soluções que integrarão a nova população. Para definir a população sobrevivente, cada

solução s^{t+1} é avaliada segundo uma função que mede o nível da sua aptidão, ou seja, a qualidade do indivíduo para o problema por critérios pré-definidos. Desta forma, aqueles indivíduos considerados mais aptos sobrevivem, isto é, tendem a passar para a população da iteração seguinte e aqueles considerados menos aptos tendem a serem descartados.

Normalmente, os critérios usados para escolher os cromossomos sobreviventes são os seguintes: (1) aleatório; (2) roleta, onde a chance de sobrevivência de cada cromossomo é proporcional ao seu nível de aptidão e (3) misto, isto é, uma combinação dos dois critérios anteriores. Em qualquer um desses critérios admite-se, portanto, a sobrevivência de indivíduos menos aptos. Isto é feito de forma a tentar-se escapar de ótimos locais.

O algoritmo genético finaliza a evolução dos indivíduos, em geral, quando um certo número de gerações (tempo) é percorrida ou quando a melhor solução encontrada atinge um certo nível de aptidão desejável, ou ainda, quando não há melhora após um certo número de gerações.

Os parâmetros principais de controle dos algoritmos genéticos são: o tamanho n da população, a probabilidade da operação de cruzamento (taxa de cruzamento), a probabilidade de ocorrer uma mutação (taxa de mutação), o número de gerações e o número de gerações sem melhora a partir do melhor individuo encontrado.

O pseudo-código de um algoritmo genético básico está descrito na figura 4.1.

```
procedimento AG
1     t ← 0;
2     Gere a população inicial P(t);
3     Avalie P(t);
4     enquanto (os critérios de parada não forem obedecidos) faça
5           t ← t + 1;
6           Gere P(t) a partir de P(t – 1);
7           Avalie P(t);
8           Defina a população sobrevivente;
9     fim enquanto
fim AG
```

Figura 4.1 – Algoritmo genético

4.2 O Algoritmo GRASP

Segundo Souza, GRASP - *Greedy Randomized Adaptive Search Procedure,* ou procedimento de busca adaptativa gulosa e aleatória, é uma técnica iterativa proposta por Feo e Resende que consiste de duas fases: uma fase de construção, na qual uma solução é gerada, elemento a elemento, e de uma fase de busca local, na qual um ótimo local na vizinhança da solução construída é pesquisado. A melhor solução encontrada ao longo de todas as interações GRASP realizadas é retornada como resultado do algoritmo de otimização GRASP (Souza, 2001; Feo e Resende, 1995).

O pseudo-código descrito pela figura 4.2 ilustra o algoritmo GRASP.

```
procedimento GRASP (f(.),g(.),N(.),GRASPmax,s)
1     f* ← ∞;
2     para (Iter = 1,2,...,GRASPmax) faça
3           Construcao(g(.),α,s)
4           BuscaLocal(f(.),N(.),s);
5           se (f(s) < f*) entao
6                 s* ← s;
7                 f* ← f(s);
8           fim se
9     fim para
10    s ← s*;
11    retorne s;
fim GRASP
```

Figura 4.2 – Algoritmo GRASP

Consideraremos para o algoritmo GRASP, uma função objetivo *f* que associa uma solução *s* a um valor real *f(s)* e que procuramos minimizar, a solução *s** como sendo a melhor solução encontrada, logo, *f* = f(s*)* e o parâmetro *GRASPmax,* tido como critério de parada do algoritmo, que determina o número máximo de iterações permitidas sem melhora da função objetivo. A vizinhança de uma solução *s* é representada pelo conjunto de soluções vizinhas *N(s)* e será definida na fase de busca local. A função gulosa é representada pela função *g* e a taxa gulosa pelo valor α e serão abordados na fase de construção do algoritmo GRASP

4.2.1 As Fases do Algoritmo GRASP

O algoritmo GRASP consiste de duas fases. São elas:

Fase de Construção:

Na fase de construção, uma solução é iterativamente construída, elemento por elemento. A cada iteração dessa fase, os próximos elementos candidatos a serem incluídos na solução são colocados em uma lista *C* de candidatos, seguindo um critério de ordenação pré-determinado. O processo de seleção é baseado em uma função adaptativa gulosa $g : C \rightarrow R$, que estima o benefício da seleção de cada um dos elementos. A heurística é adaptativa porque os benefícios associados com a escolha de cada elemento são atualizados em cada iteração da fase de construção para refletir as mudanças oriundas da seleção do elemento anterior. A componente probabilística $\alpha \in [0,1]$, denominada taxa gulosa, reside no fato de que cada elemento é selecionado de forma aleatória a partir de um subconjunto restrito formado pelos melhores elementos que compõem a lista de candidatos. Este subconjunto recebe o nome de lista de candidatos restrita *LCR*. Essa técnica de escolha permite que diferentes soluções sejam geradas em cada iteração GRASP.

O pseudo-código representado pela figura 4.3 descreve a fase de construção GRASP.

```
procedimento Construcao (g(.),α,s)
1     s ← ∞;
2     Inicialize o conjunto C de candidatos;
3     enquanto (C ≠ {}) faça
4           t_min = min{g(t) | t ∈ C};
5           t_max = max{g(t) | t ∈ C};
6           LCR = {t ∈ C | g(t) ≤ t_min + α(t_max − t_min)};
7           Selecione aleatoriamente um elemento t ∈ LCR;
8           s ← s ∪ {t};
9           Atualize o conjunto C de candidatos;
10    fim enquanto
11    retorne s;
fim Construcao
```

Figura 4.3 – Fase de construção de um algoritmo GRASP

O parâmetro α controla o nível de gulosidade e aleatoriedade na geração das soluções durante as iterações GRASP. Um valor $\alpha = 0$ faz gerar soluções puramente gulosas, enquanto $\alpha = 1$ faz produzir soluções totalmente aleatórias.

As soluções geradas pela fase de construção do GRASP provavelmente não são localmente ótimas com respeito à definição de vizinhança que adotaremos a seguir na fase de busca local. Daí a importância da fase de busca local, a qual objetiva melhorar a solução construída.

Fase de Busca Local:

A fase de busca local está baseada na noção de vizinhança. A função *N*, a qual depende da estrutura do problema tratado, associa a cada solução viável *s* sua vizinhança *N(s)*. Cada solução $s' \in N(s)$ é chamado de vizinho de *s*. É denominado *movimento* a modificação *m* que transforma uma solução *s* em outra *s'*, que esteja em sua vizinhança. Representa-se essa operação por $s' \leftarrow s \oplus m$.

Em linhas gerais, a fase de busca local, começando de uma solução obtida pela fase de construção GRASP, navega pelo espaço de pesquisa passando de uma solução para outra, que seja sua vizinha, em busca de um ótimo local.

A figura 4.4 descreve o pseudo-código de um algoritmo básico de busca local com respeito a uma certa vizinhança *N(.)* de *s*.

```
procedimento BuscaLocal (f(.),N(.),s)
1     s* ← s;
2     V = {s' ∈ N(s) | f(s') < f(s)};
3     enquanto (|V| > 0) faça
4           Selecione s ∈ V;
5           se (f(s) < f(s*)) entao s* ← s;
6           V = {s' ∈ N(s) | f(s') < f(s)};
7     fim enquanto
8     s ← s*;
9     retorne s;
fim BuscaLocal
```

Figura 4.4 – Fase de busca local de um algoritmo GRASP

A eficiência da busca local depende da qualidade da solução construída na fase de construção. A fase de construção tem então um papel importante na busca local, uma vez que as soluções construídas constituem bons pontos de partida para a busca local, permitindo assim acelerá-la.

4.2.2 Considerações sobre a Taxa Gulosa

O parâmetro α que determina o tamanho da lista de candidatos restrita, é basicamente o único parâmetro a ser ajustado na implementação de um algoritmo GRASP. Valores da taxa gulosa α que levam a uma lista de candidatos restrita de tamanho muito limitado, ou seja, o valor α próximo da escolha gulosa, implicam em soluções finais de qualidade muito próximas àquela obtida de forma puramente gulosa, obtidas com um baixo esforço computacional. Em contrapartida, provocam uma baixa diversidade de soluções construídas. Já uma escolha de α próxima da seleção puramente aleatória leva a uma grande diversidade de soluções construídas mas, por outro lado, muitas das soluções construídas são de qualidade inferior, tornando mais lento o processo de busca local.

O algoritmo GRASP procura, portanto, conjugar bons aspectos dos algoritmos puramente gulosos, com aqueles dos algoritmos aleatórios de construção de soluções.

4.2.3 Implementação

A fase de construção do algoritmo GRASP foi implementada no algoritmo seqüencial e nos algoritmos distribuídos para gerar uma população inicial de soluções para o algoritmo genético. O objetivo foi evitar a possibilidade de gerar, com um método aleatório, elevada quantidade de indivíduos de qualidade inferir para a população inicial, acelerando o processo de convergência para soluções qualificadas. A taxa gulosa deverá ser controlada para que a população inicial não tenha tendências a ser homogênea, evitando assim a convergência prematura.

5 Computação Paralela

A computação paralela ocorre quando, num dado instante, existe mais de um processador trabalhando na resolução de um mesmo problema (Colouris, Dollimore e kindberg, 1996).

Considerando a definição de computação paralela acima, temos duas maneiras de obter um processamento paralelo (Colouris, Dollimore e kindberg, 1996):

1. Usando um computador que possua mais de um processador e um sistema operacional capaz de escaloná-los. A esse tipo de paralelismo damos o nome de paralelismo interno.
2. Usando processadores de computadores distintos para obter o paralelismo. Chamaremos esse tipo de paralelismo de paralelismo externo.

O paralelismo interno é a maneira clássica de se obter um processamento paralelo e é também a mais rápida e eficiente. Por outro lado, esta solução exige um *hardware* especial cuja complexidade de projeto aumenta exponencialmente com o número de processadores. Exige um sistema operacional que consiga gerenciar o processamento paralelo e programadores treinados em desenvolvimento de código destinado a ser executado em mais de um processador, possivelmente de tipos diferentes. Tudo isso se reflete em um aumento dos custos que podem inviabilizar o projeto de uso de computação paralela.

O paralelismo externo pressupõe o uso de uma rede de comunicação que interligue os processadores dos computadores para que estes possam trabalhar em conjunto. Além da rede de comunicação, é necessário uma camada de *software* que possa gerenciar o uso paralelo destes computadores.

Por um lado, o paralelismo externo é potencialmente mais lento que o interno devido às baixas velocidades de transmissão das redes locais em comparação com a velocidade do *barramento* interno que liga os processadores de um computador paralelo. Por outro lado, o paralelismo externo é muito mais acessível, pois utiliza equipamentos de linha que são mais baratos, não exige investimentos em software, pois existem softwares gratuitos que implementam o gerenciamento das tarefas paralelas e também não exige conhecimentos especiais dos programadores que aprendem rapidamente a usar as primitivas de paralelização.

Enfatizaremos, neste projeto, o paralelismo externo implementado nos programas distribuídos propostos com o uso do sistema de paralelização PVM.

6 Paralelização do Algoritmo Seqüencial Híbrido

A paralelização de algoritmos em geral tem como objetivos diminuir o tempo de resposta e aumentar o tamanho do problema que pode ser resolvido em um tempo viável, não sendo diferente para os algoritmos genéticos. Com esses objetivos, explorando a natureza inerentemente paralela presente nos algoritmos genéticos, foram pesquisados e desenvolvidos algoritmos distribuídos para o algoritmo seqüencial e híbrido, que combina técnicas heurísticas de otimização GRASP e algoritmo genético para implementação do problema *Job-Shop*.

O algoritmo seqüencial, isto é, algoritmo não paralelizado, utilizou a técnica GRASP para criação da população inicial de indivíduos. A população inicial, seja ela gerada por meio de quaisquer técnicas, é um parâmetro requisitado pelos algoritmos genéticos para iniciar o processo de evolução dos indivíduos. Nesse sentido, percebe-se que o algoritmo genético foi a técnica de otimização utilizada como base de implementação do problema *Job-Shop* e é nessa técnica que investimos nossas pesquisas em busca do paralelismo, uma vez que ela trata uma população de soluções para um problema a cada geração e não uma única solução por vez.

Nesse capítulo, relatamos três versões distribuídas para o algoritmo seqüencial híbrido e avaliamos as versões em termos de tempo de processamento e distribuição da carga computacional comparados ao modelo seqüencial. Também descrevemos nesse capítulo o sistema PVM utilizado para configurar o ambiente distribuído para execução dos algoritmos paralelizados e realização de experimentos computacionais.

6.1 Configurando o Ambiente Distribuído

A fase de configuração do ambiente distribuído consiste em preparar uma arquitetura para que sejam executados os programas paralelizados e realizado os testes computacionais.

O ambiente utilizado para avaliação e execução dos algoritmos distribuídos consiste de um conjunto de três computadores *K7-Duron* 850 *MHz*, com 128 *MB* de memória cada, sistema operacional *Linux Conectiva 7.0 Kernel 2.4*. Cada computador contém um único processador e estes são interconectados por uma rede *Ethernet*, com 10 *Mbps* de banda. Os dados são lidos e os resultados armazenados no disco do computador em que foi executado o programa distribuído. Para a comunicação e distribuição dos processos entre os computadores foi utilizado o sistema PVM, versão 3.4.4, que trata o ambiente distribuído como se fosse uma única e potente máquina virtual para os processos em execução.

A programação paralela implementada nas versões distribuídas do algoritmo seqüencial aumentará ainda mais o desempenho computacional já presente com a utilização de técnicas de otimização GRASP e algoritmo genético. A computação com alto desempenho foi aplicada ao problema *Job-Shop* visando a obtenção de soluções qualificadas para o problema sem grandes esforços computacionais e é discutida a seguir.

6.1.1 Computação com Alto Desempenho

O alto desempenho não depende exclusivamente da utilização de *hardware* mais eficiente, mas inclui novas técnicas de processamento para solucionar um problema. Um exemplo é a programação paralela, técnicas de otimização, programação vetorial, entre outros. A computação com alto desempenho é a integração de vários conceitos de *hardware* e *software* no intuito de resolver desafios computacionais num menor período de tempo (CCUEC e FINEP, 2002).

Entre os vários objetivos de realizarmos computação com alto desempenho, destaca-se a redução do tempo de processamento de uma aplicação e a distribuição da carga computacional entre os vários processos em execução.

Abrangemos, na seção 6.1.2 *Sistema Computacional Distribuído*, o conceito de sistemas distribuídos e suporte para programação distribuída para fornecer um auxílio teórico para a criação dos programas distribuídos e configuração do ambiente distribuído utilizando o PVM.

6.1.2 Sistema Computacional Distribuído

Segundo Bal, Steiner e Tanenbaum, um sistema computacional distribuído consiste 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 (Bal, Steiner e Tanenbaum, 1989).

Sistemas dessa natureza têm proporcionado um grande poder computacional através do paralelismo. 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 (Bal, Steiner e Tanenbaum, 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 tornar um fator determinante. Através da exploração das partes de um programa que, potencialmente, possam ser executadas independentemente em processadores diferentes pode-se alcançar redução significativa no tempo de processamento. Um exemplo típico dessa categoria de aplicação são as aplicações distribuídas para o problema *Job-Shop* desenvolvidas neste projeto. Nas aplicações distribuídas desenvolvidas um espaço de soluções possíveis para o problema deve ser vasculhado por vários processos cooperantes, simultaneamente, almejando uma redução no tempo gasto para obter soluções qualificadas.

Programa distribuído é um programa paralelo ou concorrente que em execução, executa diversos processos que são distribuídos por entre vários processadores, sendo todos, executados concorrentemente independentes uns dos outros. Os processos componentes

comunicam-se através de passagens de mensagens e é através da coordenação dos processos por meio de passagens de mensagens que a lógica do programa paralelo é mantida. 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.

Sistemas computacionais distribuídos são potencialmente mais confiáveis do que os tradicionais uma vez que possuem a propriedade de falha parcial. 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, automação industrial, etc, essa propriedade é altamente desejável. Dessa forma, a 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.

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.

6.1.2.1 Suporte para a Programação Distribuída

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.

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 à 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. O programador é quem deverá escrever o código 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 (Bem, 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.

6.1.2.2 Suporte para Sistemas Computacionais Heterogêneos

Numa rede homogênea todos os processadores são iguais dispondo das mesmas capacidades no que diz respeito aos recursos genéricos como, por exemplo, velocidade de comunicação. Esta situação é radicalmente diferente numa rede que tipicamente comporta diferentes arquiteturas de computadores, diferentes compiladores e outros recursos.

A exploração de um sistema heterogêneo de computadores tem que ter atenção aos seguintes tópicos:

- Arquitetura
- Formato dos dados
- Velocidade de processamento
- Carga computacional
- Carga de comunicações.

Segue-se então a descrição de cada tópico:

6.1.2.2.1 Arquitetura

O conjunto de computadores na rede inclui uma vasta gama de arquiteturas que contemplam tanto os mais simples computadores pessoais como os super-computadores. Esta diversidade ocasiona os seguintes problemas:

- Cada arquitetura tem o seu próprio modelo de programação.
- Uma máquina virtual paralela pode também, por sua vez, ser constituída por outros computadores paralelos.
- A mesma arquitetura pode usar formatos binários incompatíveis.
- É necessário compilar tarefas em cada uma das máquinas.

6.1.2.2.2 Formato dos Dados

O formato dos dados são muitas vezes incompatíveis. Esta incompatibilidade é de enorme importância se considerarmos a necessidade de transferência de dados entre diferentes tarefas possivelmente distribuídas em diferentes arquiteturas. O método de passagem de mensagem gera diferenças entre os formatos dos dados necessitando conversões apropriadas quer na origem quer no destino das mensagens para que o método seja utilizado em todo e qualquer computador.

6.1.2.2.3 Velocidade de Processamento

Para a heterogeneidade, contribuem ainda as diferentes velocidades de processamento dos computadores. Se considerarmos o caso em que um supercomputador concorre com outros computadores na mesma máquina virtual, há que assegurar que o programa tem que ser desenvolvido de forma a evitar que o supercomputador fique á espera dos dados que deverão ser enviados por outros computadores de mais baixo desempenho. O problema da velocidade de processamento abrange também a situação, em que as máquinas são diferenciadas pelos seus recursos.

6.1.2.2.4 Carga Computacional

Com efeito, a carga computacional em cada máquina e em cada momento é caracterizada pelos recursos computacionais exigidos pelas diferentes aplicações em execução.

6.1.2.2.5 Carga de Comunicação

Da mesma forma que a carga computacional, o tempo usado para enviar uma mensagem na rede depende da carga de comunicação na rede imposta por todos os processos que fazem uso da rede. Quanto maior a comunicação na rede maior será a sua latência. Este tempo de comunicação pode ser de grande relevância em programas sensíveis ao instante de chegada de mensagens ou quando isso se reflete numa efetiva perda de desempenho, ocasionada pela necessidade de uma tarefa ter de suspender a sua atividade até descongestionar o sistema de comunicações e a mensagem chegar ao destino.

6.1.2.3 Vantagens da Computação Distribuída

Apesar das numerosas dificuldades provocadas pela heterogeneidade a computação distribuída oferece também muitas vantagens:

- Baixo custo: rentabilização de equipamento já existente.
- Otimização de desempenho: atribuição a cada máquina da tarefa mais apropriada à arquitetura.
- Exploração da natureza heterogênea da aplicação: promovendo o acesso a diferentes serviços em diferentes computadores, como é o caso das bases de dados ou processadores específicos, para certas tarefas da aplicação que só podem ser executadas em plataformas específicas.
- Escalabilidade : o desempenho de um programa melhora se for executado em uma máquina mais eficiente à nível de hardware e dependendo da natureza da aplicação, do paralelismo adotado e do hardware, podemos ter maior

desempenho de um programa quando executado em um maior número de máquinas que compõe o sistema distribuído.

- Estabilidade e perícia: normalmente, cada máquina instalada cumpre um fim determinado, o que favorece a criação de especialistas.
- Favorece a programação: cada tarefa pode ser criada utilizando o editor, compilador mais apropriado, disponível em cada máquina.
- Tolerância a falhas: com a depuração no acesso às rotinas e funcionalidades das rotinas que implementam o paralelismo em uma aplicação. Pode ser realizada tanto ao nível da tarefa como da aplicação global.
- Desenvolvimento: favorece o trabalho cooperativo.

6.1.3 Utilizando o Sistema PVM

O sistema PVM - *Parallel Virtual Machine*, escolhido para configurarmos o ambiente distribuído, nos permite explorar o alto desempenho via programação paralela. O sistema oferece rotinas para implementação do paralelismo nas aplicações seqüenciais e programas para configuração de uma arquitetura distribuída. Essa arquitetura distribuída simula uma máquina virtual poderosa, permitindo o processamento paralelo da aplicação entre os processadores que compõe o sistema distribuído (Geist, Beguelin, et. al., 1994).

A execução de uma aplicação paralela sobre a máquina virtual pode ter consideravelmente o tempo de processamento reduzido em relação a uma aplicação seqüencial e a carga computacional da aplicação será distribuída entre os processadores disponíveis que ficarão encarregados de processá-las.

6.1.3.1 O Sistema PVM

O sistema PVM é um conjunto integrado de bibliotecas e programas que fazem a emulação de um sistema flexível, de uso geral numa rede distribuída e heterogênea de computadores. Faz com que um conjunto de computadores na rede distribuída seja visto como um multicomputador MIMD - *Multiple Instruction, Multiple Data*, onde o paralelismo é efetuado conectando-se múltiplos processadores e cada processador executa um conjunto de instruções sobre um conjunto de dados independentemente dos outros processadores, apresentando ao usuário uma máquina virtual (Geist, Beguelin, et. al., 1994).

Máquina virtual é o conjunto de computadores, cada um operando, através do PVM, como se fosse uma máquina única. Essa máquina virtual dividirá o programa em processos para executarem nos processadores que a compõem.

6.1.3.1.1 Visão Geral

O PVM gera de forma transparente todo o encaminhamento de mensagens, a conversão de dados e o escalonamento de tarefas que se realiza numa rede heterogênea de computadores.

O sistema PVM oferece:

- Programação simples - uma aplicação é uma coleção de tarefas cooperativas.
- Interface (console) - para pedir recursos á máquina virtual como disparar processos, adicionar ou remover computadores da máquina virtual.
- Biblioteca de rotinas:
    - Para iniciar e terminar tarefas através da rede.
    - Para sincronização a comunicação entre tarefas.
    - Para adicionar ou remover outros computadores na máquina virtual.
- Passagem de mensagens - orientada para operações em máquinas heterogêneas.
- Tipos fortes de dados - tanto na emissão com na recepção de mensagens.
- Primitivas básicas de comunicação - vários modos de envio e recepção de mensagens.
- Primitivas elaboradas de comunicação - para difusão e sincronização de grupos.

6.1.3.1.2 Características

As principais características do sistema PVM são :

- **Promessa de efetuar computação paralela, utilizando-se de qualquer conjunto de computadores, disponíveis em laboratórios:** as tarefas, que constituem cada aplicação particular, executam num conjunto de computadores selecionados que compõem a máquina virtual. Podem ser selecionados quaisquer tipos de computadores, uniprocessador ou multiprocessador. O conjunto inicial de computadores pode ser aumentado ou reduzido dinamicamente.
- **Acesso transparente ao** *hardware***:** a aplicação paralela *enxerga* a máquina virtual como uma coleção de processadores sem características especiais ou, em alternativa, pode aproveitar as vantagens da arquitetura de cada processador da máquina virtual colocando determinadas tarefas em computadores determinados.
- **Computação baseada em processos:** a unidade de paralelismo em PVM é a tarefa, que consiste em um código seqüencial a ser executado em um computador da máquina virtual alternando entre computação e comunicação. O modelo não força qualquer tipo de relação processo-processador em particular e múltiplas tarefas podem ser executadas num único processador.
- **Reduz o tempo total de execução de um programa.**
- **Paralelização escalável.**
- **Opera em ambientes heterogêneos de máquinas, redes e aplicações**.

- **Comunicação por passagem de mensagens:** no que diz respeito ao modelo de comunicação, modelo de passagem de mensagens explícito, as tarefas que formam o programa paralelo dividem o trabalho através da troca de mensagens determinada pelo programador à serem realizada pelas mesmas. O método de comunicação passagem de mensagens se baseia na transmissão de dados *send*/*receiv*, via uma rede de interconexão, seguindo as regras de um protocolo de rede. A manutenção da lógica é de responsabilidade do programador.
- **Multiusuário:** múltiplos usuários podem criar a sua própria máquina virtual, com os recursos associados, e cada usuário poderá executar múltiplas aplicações simultaneamente.
- **Pode trabalhar com FORTRAN ou C:** atualmente, as linguagens hospedeiras que podem chamar rotinas PVM são o C, o C++ e o Fortran que foram escolhidas por se considerar que a maioria das aplicações atualmente desenvolvidas são escritas nessas linguagens.
- *Software* **de domínio público:** desenvolvido por Oak Ridge National Laboratory em 1989.

6.1.3.1.3 Composição do PVM

O sistema PVM é composto de duas partes o servidor *pvmd* e uma biblioteca de rotinas para que o programador possa fazer uso no desenvolvimento das aplicações paralelas.

O servidor pvmd:

Para tratar um conjunto de computadores interligados por uma rede como um grande computador multiprocessador virtual, o PVM executa um programa chamado servidor *pvmd* em cada uma das máquinas componentes da máquina virtual. Toda a comunicação entre dois computadores da máquina virtual passa pelos programas *pvmd* que executam nos computadores em questão. Por exemplo, se um processo de um computador A quer mandar uma mensagem para um processo no computador B, esta mensagem será enviada para o *pvmd* do computador A, passará para o *pvmd* do computador B e só então será repassada para o processo destino no computador B.

Montar uma máquina virtual no PVM significa iniciar um *pvmd* em cada um dos computadores que queremos presente na máquina virtual. O primeiro *pvmd* que é iniciado na máquina virtual funciona como mestre e é o único capaz de reconfigurar a máquina virtual como, por exemplo, acrescentar ou remover computadores. A partir do *pvmd* mestre, os demais *pvmds* são iniciados como escravos. Pedidos de reconfiguração da máquina virtual vindos de processos cujo *pvmd* local seja um *pvmd* escravo são repassados ao *pvmd* mestre e executados de lá.

No sistema PVM todas as tarefas, ou processos, são identificadas por um número inteiro *tid*, único em toda a máquina virtual, que é criado automaticamente pelo servidor *pvmd* local sempre que uma nova tarefa é gerada. As tarefas comunicam enviando e recebendo, explicitamente, mensagens que identificam o destinatário e o remetente com base nos respectivos números de identificação *tids*.

Algumas das tarefas do servidor *pvmd* se encontram: autenticação de mensagens; execução dos processos nos computadores; detecção de falhas; roteamento de mensagens e armazenamento temporário das mensagens até que o computador possa lê-las.

Biblioteca de Rotinas:

O sistema PVM contém uma biblioteca de rotinas a ser incluída no programa do usuário. A biblioteca contém um repertório completo de rotinas utilizada para efetuar a comunicação entre processos paralelos distribuídos entre os processadores que compõem a máquina virtual. Ela inclui rotinas acessíveis ao programa do usuário, para a criação de tarefas, a passagem de mensagens, a sincronização entre tarefas e a alteração da máquina virtual.

6.1.3.1.4 O Console do PVM

O console do PVM é um programa chamado *pvm* que acompanha a distribuição do PVM cuja função é fornecer uma interface para que o usuário modifique a configuração da máquina virtual e possa executar diversas tarefas de gerenciamento.

Algumas tarefas comuns são: acrescentar e remover computadores da máquina virtual; iniciar e encerrar tarefas em computadores que fazem parte da máquina virtual; listar parte ou todas as tarefas na máquina virtual.

6.1.3.2 Desenvolvendo uma Aplicação em PVM

Em geral, os passos para o desenvolvimento de uma aplicação em PVM são:

- Escrita de programas: em C, C++ ou Fortran que contém chamadas às bibliotecas do PVM, correspondendo cada programa a uma tarefa da aplicação.
- Compilação: dos programas para cada arquitetura e respectiva instalação em localização acessível.
- Execução: da aplicação, que corresponde a uma cópia da tarefa inicial mestre, em qualquer um dos computadores que constituem a máquina virtual se encarregando de iniciar a execução em paralelo das demais tarefas da aplicação, que também possuem cópias em qualquer dos computadores da máquina virtual.

6.2 Custos com a Computação Paralela

Os custos que mais se destacaram com a paralelização do algoritmo seqüencial e híbrido levando ao surgimento dos algoritmos distribuídos foram:

1. Tempo dispensado para configuração da máquina virtual paralela;
2. Tempo dispensado para estudo das rotinas PVM;
3. Tempo para depuração do programa seqüencial;

    Foi depurado o programa seqüencial e verificado quais os procedimentos, *loops* e rotinas que mais consumiram tempo de processamento.

4. Tempo dispensado em analisar o código seqüencial para paralelizar;

    - Foi analisado e avaliado se os procedimentos, *loops* e rotinas que mais consumiram tempo de processamento poderiam ser executados concorrentemente em outros processadores.
    - Foi verificado a necessidade de transmissão de dados entre esses procedimentos. Rotinas específicas da biblioteca PVM foram usadas para garantir a sincronização, isto é, espera por algum resultado.
    - Não houve o surgimento de dependências para uma específica arquitetura. A máquina virtual consistia de computadores e aplicações homogêneas como sistema operacional, compiladores, etc numa rede Ethernet 10 Mbps.

5. Tempo dispensado para codificar o programa seqüencial no formato paralelo;

    Tempo para incluir as rotinas PVM apropriadas para paralelização no programa fonte.

6. Paralelização;

    O custo na paralelização é inevitável, mas que deve ser controlado e minimizado de forma a garantir a eficiência da aplicação paralela em termos de tempo de processamento comparado a aplicação seqüencial.

    O custo de paralelização envolve:

    - Tempo para iniciar um processo.
    - Tempo para finalizar um processo.
    - Tempo de sincronização dos processos.
    - Tempo gasto na comunicação entre os processos.

Para qualquer modelo que adota paralelismo é importante diminuir a comunicação entre os diversos processadores, por causa da latência das redes que interconectam os processadores. Caso contrário, podem ser obtidos resultados piores que os seqüenciais. Porém, a comunicação é realizada para atualizar o estado dos diversos processos, devendo haver um balanceamento entre comunicação e processamento para que os resultados obtidos não sejam prejudicados.

Os algoritmos distribuídos foram implementados visando o menor custo possível com a comunicação entre processos, entretanto, para que o usuário pudesse visualizar a evolução dos indivíduos, que corresponde a melhor solução gerada em cada geração e o tempo de processamento da geração, foram adicionados custos de tempo com a coleta de tais informações provenientes dos processos em execução.

Os custos que conseguimos reduzir com a paralelização do algoritmo seqüencial foram:

1. Tempo total de processamento requisitado pelo programa seqüencial.

   Minimizamos com a paralelização os longos tempos de espera por melhores resultados do programa seqüencial em relação ao tamanho da população de soluções a serem evoluídas, especialmente para os casos em que temos uma população muito grande.

2. Necessidade de memória pelo programa seqüencial.

   A utilização da memória de cada computador da máquina virtual para o armazenamento de sub-populações de soluções distribuída entre estes, minimizou a necessidade de memória requisitada pelo programa seqüencial no computador em que executa. Quanto maior o número de computadores na máquina virtual maior será a quantidade de memória, em geral, disponibilizada para o programa paralelo para evolução de uma população com maior número de indivíduos.

6.3 Modelos de Paralelização Implementados

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 (Bal, Steiner e Tanenbaum, 1989).

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.

Como foi disponibilizado para o desenvolvimento deste projeto três computadores de mesma arquitetura e mesmas configurações de *hardware* para criação da máquina virtual PVM, resolvemos que cada algoritmo distribuído seria composto de três processos trabalhadores para distribuição da carga computacional, sendo um processo para cada computador. Os processos trabalhadores são responsáveis pela execução paralela das tarefas computacionais enquanto o processo mestre normalmente abstrai informações desejadas dos processos trabalhadores.

A manutenção lógica da comunicação entre os processos é de extrema responsabilidade do programador explicitada através do uso de rotinas PVM na aplicação, enquanto que o sistema PVM se encarrega de manter a transparência na comunicação entre os processos inclusive de garantir a comunicação sincronizada entre estes (Geist, Beguelin, Dongarra, Jiang, Manchek e Sunderam, 1994).

A combinação entre a lógica de comunicação definida pelo programador e a própria comunicação realizada pelo sistema PVM definirá a interação entre os processos para os algoritmos distribuídos implementados no projeto.

Segue-se os três modelos distribuídos implementados para a versão seqüencial e híbrida do algoritmo que trata o problema *Job-Shop*:

- MCPOP - Modelo Centralizado ao Nível de População.
- MDPOP - Modelo Distribuído ao Nível de População.
- MDPROC - Modelo Distribuído ao Nível de Processos.

Uma recodificação da versão seqüencial foi realizada para a arquitetura em que foram implementadas as versões distribuídas, apenas com o objetivo de tornar legítimas as comparações a serem realizadas entre as versões durante os testes computacionais, em especial os tempos de processamento de cada versão.

6.3.1 Modelo Centralizado ao Nível de População

O algoritmo híbrido com técnicas de otimização algoritmo genético e GRASP e distribuído para o problema *Job-Shop* que tem como característica fundamental a centralização da população de indivíduos em um dos processadores que compõem a máquina virtual foi implementado segundo o modelo centralizado ao nível de população.

Nesse modelo, os cálculos básicos do algoritmo genético seqüencial são realizados por processadores escravos, enquanto o mestre mantém a população.

A população inicial é gerada pelos processos trabalhadores também denominados processos escravos onde cada processo cria uma sub-população e a envia para o processo mestre que mantém o seu controle.

O processo mestre envia uma sub-população de indivíduos para cada processo trabalhador para que seja efetuado as operações de cruzamento, mutação e clonagem e coleta os descendentes gerados assim como informações de cada geração durante a evolução. As informações coletadas e registradas em cada geração são: o melhor indivíduo produzido na geração e o tempo gasto para evolução dos indivíduos na geração. Ressaltamos que, o melhor indivíduo na geração é obtido comparando todos os indivíduos mantidos por todos os

processos trabalhadores na geração. O tempo da geração é obtido quando todos os processos trabalhadores finalizam a geração enviando seus indivíduos gerados ao processo mestre.

O processo mestre também se encarrega de coletar e registrar o melhor indivíduo produzido durante a evolução entre todos os processos trabalhadores assim como informar o tempo de execução do programa paralelo, ou seja, o tempo de evolução dos indivíduos.

Veja a seguir o pseudo-código para o algoritmo distribuído que mantém a população centralizada.

Pseudo-código mestre :

```
jspGraspAg_CentralPop_Mestre (TAMPOP, MAX_INDIVS_POR_PROC, MAX_GERACOES,
                                TAXA_CROSSOVER, TAXA_MUTACAO, TAXA_GULOSA)
{
inicialisar tempo;
inicializar geracao;

prob = gerar Problema Job-Shop;
// definir número de processos trabalhadores a serem disparados
NPROCS = ((TAMPOP mod MAX_INDIVS_POR_PROC) == 0)?
      TAMPOP / MAX_INDIVS_POR_PROC :(TAMPOP / MAX_INDIVS_POR_PROC) + 1;

registrar processo mestre;

// inicializar NPROCS processos trabalhadores na Máquina Virtual
inciar_processo(jspGraspAg_CentralPop_Escravo,NPROCS);

informar aos trabalhadores TAXA_CROSSOVER, TAXA_MUTACAO, TAXA_GULOSA e
MAX_GERACOES;
informar aos processos trabalhadores o número de indivíduos que cada deverá
criar para constituir a população inicial;
receber indivíduos gerados pelos trabalhadores; // sub-populacao inicial
// registrar melhor individuo na geração
obter melhor_indiv_geração;
atualizar tempo;
print(geracao, fo(melhor_idiv_geração), tempo);
// registrar melhor individuo na evolução
melhor_indiv_evolução = melhor_indiv_geração;
geracao_melhor_indiv_evolucao = geração;

while ( geracao < MAX_GERACOES )
{
      //processo mestre distribui pares de individuos para os trabalhadores
      //gerarem descendentes
      para cada processo trabalhador faca {
         enviar ao processo trabalhador pares de indivíduos; // pais
      }
      receber os descendentes de cada processo trabalhador;

      // registrar melhor individuo na geração
      obter melhor_indiv_geração;
      atualizar tempo;
      print(geracao+1, fo(melhor_idiv_geração), tempo);

      // registrar melhor individuo na evolução
      atualizar melhor_indiv_evolução;
      atualizar geracao_melhor_indiv_evolucao;
```

```
      definir população sobrevivente, segundo critério Roleta Russa, para a
      geração seguinte;
      re-atualizar melhor_indiv_geração;
      avancar geracao;
}
// imprimir melhor individuo na evolução
print(fo(melhor_indiv_evolução),melhor_indiv_evolução(prob),
      geracao_melhor_indiv_evolucao);
// imprimir tempo de evolução
atualizar tempo;
print("Tempo de evolução = ", tempo);

finalizar processo mestre;
} // jspGraspAg_CentralPop_Mestre
```

Pseudo-código trabalhador :

```
jspGraspAg_CentralPop_Escravo ()
{
inicializar geracao;

prob = gerar Problema Job-Shop;
registrar processo trabalhador;

receber do processo mestre informações como TAXA_CROSSOVER, TAXA_MUTACAO,
TAXA_GULOSA e MAX_GERACOES;
n_indivs = receber do processo mestre o número de indivíduos que deverá ser
criado; // sub-populacao inicial
para 1 ... n_indiv faca {
  criar_individuo(prob, TAXA_GULOSA);
  empacotar o individuo criado;
}
enviar individuos empacotados para o processo mestre;
while ( geracao < MAX_GERACOES )
{
  receber do processo mestre os pares de indivíduos pais para gerar
  descendentes;
  para cada par_de_indivíduos {
    sorteio = sortear valor [0,1];
    if (sorteio <= TAXA_CROSSOVER)
      descendentes = gerar_filhos(par_de_indivíduos);
    else descendentes = par_de_indivíduos;
    sorteio = sortear valor [0,1];
    if (sorteio <= TAXA_MUTACAO)
      descendentes = mutar_descendentes(descendentes);
    empacotar descendentes;
  }
  enviar os descendentes para o processo mestre;
  avancar geracao;
}
finalizar processo trabalhador;
} // jspGraspAg_CentralPop_Escravo
```

6.3.2 Modelo Distribuído ao Nível de População

O algoritmo híbrido com técnicas de otimização algoritmo genético e GRASP e distribuído para o problema *Job-Shop* que tem como característica fundamental a distribuição da população de indivíduos entre os processadores que compõem a máquina virtual foi implementado segundo o modelo distribuído ao nível de população.

Neste modelo, ocorre a aplicação direta do paralelismo ao nível das populações denominado modelo de ilhas.

Segundo Costa, o modelo de ilhas tem como objetivo explorar o paralelismo explícito dos algoritmos genéticos visto que o alto grau de independência entre os indivíduos torna viável induzir sua evolução em paralelo. Nesta abordagem a população total é dividida em um conjunto de sub-populações ou ilhas para cada processador e estas evoluem paralelamente. A cada geração um número pré-determinado de indivíduos é copiado ou trocado entre as populações, em uma operação conhecida como migração. A taxa de migração define este número e deve ser um valor que evite a evolução completamente independente das sub-populações, onde o problema de convergência prematura tende a ocorrer com maior freqüência. A convergência prematura ocorre quando não há diversidade dos indivíduos na população minimizando as possibilidades de obter soluções qualificadas durante a evolução que torna a população homogênea. Outro fator a ser considerado no momento de escolha da taxa de migração é o custo que esta operação oferece, ou seja, quanto maior a taxa de migração maior é os custos obtidos com a comunicação entre os processos (Costa, 1999).

Para distribuir o processamento entre os processadores que constituem a máquina virtual, a população total foi dividida entre os processadores, de tal forma que, cada processador gera sua própria população inicial.

A evolução da população em cada processador ocorre em paralelo com trocas dos melhores indivíduos entre as sub-populações vizinhas.

Para a troca dos melhores indivíduos entre os processadores, utilizamos um esquema em *anel unidirecional*. Após a evolução dos indivíduos a cada geração, cada processador envia para o seu vizinho o seu melhor indivíduo, isto é, a melhor solução encontrada na geração corrente. As trocas acontecem no intuito de preservar a diversidade da população total de indivíduos, ou seja, desejamos evitar que algumas sub-populações tenha uma grande quantidade de indivíduos “ruins” (de pouca qualidade) em relação às outras sub-populações durante a evolução. Com as trocas de melhores indivíduos entre vizinhos desejamos atingir o equilíbrio, durante a evolução, da qualidade dos indivíduos entre as sub-populações, através da distribuição na rede em *anel unidirecional* de “bons” indivíduos entre os processadores.

A figura 6.1 ilustra o esquema de distribuição do processamento. Cada processador, representado por uma elipse, possui sua própria população, representada por um quadrado vermelho. As trocas dos indivíduos, representadas pelas setas azuis, ocorrem de um processador para o seu vizinho apenas, representando uma rede em *anel unidirecional*. Desta forma, evita-se uma sobrecarga no número de mensagens transmitidas para sistemas com um grande número de processadores.

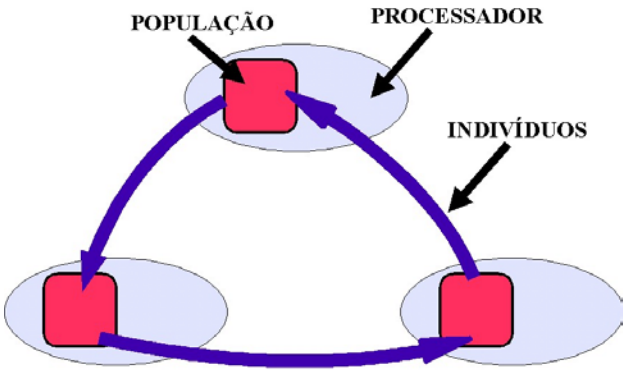


Figura 6.1 – Esquema em anel unidirecional

A comunicação é realizada através de *receives* síncronos. Cada processador envia sua mensagem e espera pela mensagem do próximo. Como a carga computacional e os processadores são uniformes, ou seja, a população é distribuída entre os processadores e estes são homogêneos, o sincronismo não constitui um problema em termos de atraso de tempo, pois os processadores chegam aproximadamente ao mesmo tempo no ponto de sincronização e, desta forma, não esperam muito pelas mensagens dos seus respectivos vizinhos.

O modelo distribuído também pode se comportar como um modelo de difusão, derivado do modelo de ilhas pelo aumento do número de sub-populações, de modo que exista um único indivíduo em cada ilha. Neste caso, conectamos os indivíduos entre si através da rede em *anel unidirecional*. Cada indivíduo "vive" no seu local e ocasionalmente interage com o seu vizinho, isto é, com certa probabilidade ocorre o cruzamento de um indivíduo com o seu vizinho gerando "filhos". Entre os pais e os filhos gerados, os indivíduos mais aptos tendem a sobreviver e ocupar as ilhas ocorrendo a difusão progressiva de bons indivíduos através da rede de interconexão em *anel unidirecional*.

No modelo de difusão temos somente um único indivíduo por processador e, por isso, temos uma maior granularidade, isto é, um maior número de processos trabalhadores em execução e, no entanto, uma menor quantidade de tarefa computacional por processo trabalhador, ou seja, menor tempo de execução do processo. Por ser maior o número de processos trabalhadores em execução é também maior a quantidade de comunicação realizada entre os processos no modelo de difusão em relação ao modelo de ilhas.

Embora o aumento da sobrecarga de comunicação aconteça com a maior quantidade de comunicação entre os processos devido a latência da rede de comunicações que interliga os processadores, o modelo de difusão se faz útil quando executamos instâncias do problema *Job-Shop* de dimensões consideráveis, que necessita de memória disponível para armazenar cada indivíduo a ser distribuído entre os processadores. O problema é que o modelo de difusão, em comparação ao modelo de ilhas, cria uma dependência de um maior número de processadores para processar cada indivíduo que constitui a população. Para uma população de poucos indivíduos (poucos processadores) a evolução se tornará lenta devido ao pequeno teor de diversidade dos indivíduos e com possibilidades de ocorrer com maior freqüência a convergência prematura dos indivíduos durante o processo evolucional.

Enquanto os processos trabalhadores criam a sua sub-população de indivíduos e a evoluem com trocas dos melhores indivíduos entre si, o processo mestre se encarrega de coletar informações em cada geração durante a evolução. As informações coletadas e registradas em cada geração são: o melhor indivíduo produzido na geração e o tempo gasto para evolução dos indivíduos na geração. Ressaltamos que, o melhor indivíduo na geração é obtido comparando todos os indivíduos presentes em cada sub-população na geração. O tempo da geração na *ilha* em que o melhor individuo foi gerado é obtido quando o processo trabalhador que contém tal individuo finaliza a geração.

O processo mestre também se encarrega de coletar e registrar o melhor indivíduo produzido durante a evolução entre todas as sub-populações assim como informar o tempo de execução do programa paralelo, ou seja, o tempo de evolução dos indivíduos.

Veja a seguir o pseudo-código para o algoritmo distribuído que evolui a população distribuída entre os processadores da máquina virtual .

Pseudo-código mestre :

```
jspGraspAg_DistrPop_Mestre (TAMPOP, MAX_INDIVS_POR_PROC, MAX_GERACOES,
                            TAXA_CROSSOVER, TAXA_MUTACAO, TAXA_GULOSA)
{
inicialisar tempo;
inicializar geracao;

prob = gerar Problema Job-Shop;
// definir número de processos trabalhadores a serem disparados
NPROCS = ((TAMPOP mod MAX_INDIVS_POR_PROC) == 0)?
      TAMPOP / MAX_INDIVS_POR_PROC :(TAMPOP / MAX_INDIVS_POR_PROC) + 1;

registrar processo mestre;

// inicializar NPROCS processos trabalhadores na Máquina Virtual
inciar_processo(jspGraspAg_DistrPop_Escravo,NPROCS);

informar aos trabalhadores TAXA_CROSSOVER, TAXA_MUTACAO, TAXA_GULOSA e
MAX_GERACOES;
informar aos trabalhadores os seus processos vizinhos (anterior e
posterior) e o tamanho da populacao a criar;
repetir enquanto (geracao <= MAX_GERACOES)
{
  // registrar melhor individuo entre os trabalhadores na geração
  para cada processo trabalhador {
    obter fo_melhor_indiv_trabalhador_geração;
    obter tempo_da_geração_trabalhador;
    if (fo_melhor_indiv_trabalhador_geração é o
        menor valor makespan na geracao) {
        fo_melhor_indiv_geração = fo_melhor_indiv_trabalhador_geração;
        tempo_da_geração = tempo_da_geração_trabalhador;
    }
  }
  print(geracao, fo_melhor_idiv_geração, tempo_da_geração);
  atualizar geração;
}
// receber de cada trabalhador o melhor individuo gerado durante a evolução
// registrar melhor individuo na evolução
para cada processo trabalhador {
  obter melhor_indiv_trabalhador;
  obter geracao_melhor_indiv_trabalhador;
```

```
  if ( (fo(melhor_indiv_trabalhador) é o menor valor makespan na evolução)
       ou (o melhor valor makespan obtido ate então é igual ao
           fo(melhor_indiv_trabalhador) e geracao_melhor_indiv_trabalhador
                                        < geracao_melhor_indiv_evolucao))
  {
      melhor_indiv_evolucao = melhor_indiv_trabalhador;
      geracao_melhor_indiv_evolucao = geracao_melhor_indiv_trabalhador;
  }
}
// imprimir melhor individuo na evolução
print(fo(melhor_indiv_evolução),melhor_indiv_evolução(prob),
      geracao_melhor_indiv_evolucao);
// imprimir tempo de evolução
atualizar tempo;
print("Tempo de evolução = ", tempo);

finalizar processo mestre;
} // jspGraspAg_DistrPop_Mestre
```

Pseudo-código trabalhador :

```
jspGraspAg_DistrPop_Escravo ()
{
inicialisar tempo;
inicializar geracao;

prob = gerar Problema Job-Shop;
registrar processo trabalhador;

receber do processo mestre informações como TAXA_CROSSOVER, TAXA_MUTACAO,
TAXA_GULOSA e MAX_GERACOES;
receber do processo mestre identificadores dos vizinhos, vizinho_anterior e
vizinho_posterior;
n_indivs = receber do processo mestre o número de indivíduos que deverá ser
criado; // sub-populacao
// criar população inicial (sub-populacao)
para 1 ... n_indiv faca {
  criar_individuo(prob, TAXA_GULOSA);
}
atualizar tempo;
// registrar melhor individuo na geração
obter melhor_indiv_geração;
// registrar melhor individuo na evolução
melhor_indiv_evolução = melhor_indiv_geração;
geracao_melhor_indiv_evolucao = geração;
// enviar ao processo mestre makespan do melhor individuo na geracao
enviar ao processo mestre {
   fo(melhor_indiv_geração);
   tempo; // tempo da geração.
}
informar ao vizinho_anterior o tamanho da população do trabalhador;
n_indivs_vizinho_posterior = receber do vizinho_posterior o tamanho da sua
população;

while ( geracao < MAX_GERACOES )
{
      if (n_indivs_vizinho_posterior == 1)
        enviar melhor_indiv_geração para vizinho_posterior;
      if (n_indivs == 1) {
        pai_vizinho = receber o individuo pai do vizinho_anterior;
        par_de_indivíduos = pai_vizinho + único individuo da populacao;
        sorteio = sortear valor [0,1];
```

```
        if (sorteio <= TAXA_CROSSOVER)
             descendentes = gerar_um_filho(par_de_indivíduos);
        else descendentes = par_de_indivíduos;
        sorteio = sortear valor [0,1];
        if (sorteio <= TAXA_MUTACAO)
             descendentes = mutar_descendentes(descendentes);
      }
      else {
         // sortear individuos pais e fazer cruzamento e mutação
         faca de 1 ... n_indivs/2 {
           sortear par_de_indivíduos pais da população;
           sorteio = sortear valor [0,1];
           if (sorteio <= TAXA_CROSSOVER)
             descendentes = gerar_filhos(par_de_indivíduos);
           else descendentes = par_de_indivíduos;
           sorteio = sortear valor [0,1];
           if (sorteio <= TAXA_MUTACAO)
             descendentes = mutar_descendentes(descendentes);
         }
      }
      atualizar tempo;
      // registrar melhor individuo na geração
      obter melhor_indiv_geração;
      // registrar melhor individuo na evolução
      melhor_indiv_evolução = melhor_indiv_geração;
      geracao_melhor_indiv_evolucao = geração+1;
      // enviar ao processo mestre makespan do melhor individuo na geracao
      enviar ao processo mestre {
         fo(melhor_indiv_geração);
         tempo; // tempo da geração.
      }
      // trocar os melhores individuos gerados na geração com o
      // vizinho_posterior
      if (n_indivs_vizinho_posterior ≠ 1 )
        enviar melhor_indiv_geração para o vizinho_posterior;
      if (n_indivs ≠ 1 ) {
        receber melhor individuo na geração do vizinho_anterior;
        atualizar melhor_indiv_geração;
      }
      definir população sobrevivente, segundo critério Roleta Russa, para a
      geração seguinte;
      re-atualizar melhor_indiv_geração;
      avancar geracao;
}
//enviar ao processo mestre o melhor individuo gerado durante a evolução
enviar ao processo mestre {
       melhor_indiv_evolucao;
       geracao_melhor_indiv_evolucao;
}

finalizar processo trabalhador;
} // jspGraspAg_DistrPop_Escravo
```

6.3.3 Modelo Distribuído ao Nível de Processos

O algoritmo híbrido com técnicas de otimização algoritmo genético e GRASP e distribuído para o problema *Job-Shop* que tem como característica fundamental a execução em paralelo de diversos algoritmos genéticos diferentes e independentes entre si nos

processadores que compõem a máquina virtual foi implementado segundo o modelo distribuído ao nível de processos.

O modelo efetivamente implementado é bastante simples. Cada processo trabalhador disparado pelo processo mestre executa um algoritmo genético de forma assíncrona, ignorando totalmente a execução dos demais processos trabalhadores. Ao final da sua execução, cada processo trabalhador envia o seu melhor indivíduo encontrado ao processo mestre que registra, ao término de todas as execuções, o melhor indivíduo gerado durante a evolução entre todos os processos trabalhadores. O processo mestre também informa o tempo de execução do programa paralelo, ou seja, o tempo de evolução dos indivíduos. O processo mestre encarrega de disparar um único algoritmo genético em cada processador na máquina virtual a fim de evitar sobrecarga no processador, que por sinal já processa uma população. Para cada algoritmo genético, podemos definir o número de indivíduos que constitui a população inicial a ser evoluída em cada processador, o número de gerações limite para evolução entre outros parâmetros.

O processo mestre também se responsabiliza de coletar informações em cada geração durante a evolução dos indivíduos em cada algoritmo genético. As informações coletadas e registradas em cada geração são: o melhor indivíduo produzido na geração e o tempo gasto para evolução dos indivíduos na geração. Ressaltamos que, o melhor indivíduo na geração é obtido comparando todos os indivíduos presentes em cada população, na geração, em cada algoritmo genético. O tempo da geração no algoritmo genético em que o melhor individuo foi gerado é obtido quando o algoritmo genético que contém tal indivíduo finaliza a geração.

Esse modelo de implementação tem seu valor prático, uma vez que, em se tratando de um método de aproximação, normalmente o indivíduo final é retirado de uma série de execuções de um mesmo problema.

Veja a seguir o pseudo-código para o algoritmo distribuído que executa diversos Algoritmos Genéticos diferentes e independentes entre si nos processadores da máquina virtual.

Pseudo-código mestre :

```
jspGraspAg_DistrProc_Mestre (NPROCS)
{
inicialisar tempo;
inicializar geracao;

prob = gerar Problema Job-Shop;
registrar processo mestre;

// inicializar NPROCS processos trabalhadores na Máquina Virtual
inciar_processo(jspGraspAg_DistrProc_Escravo,NPROCS);

para cada processo trabalhador {
    receber n_indivs_trabalhador;
    receber max_geraçoes_trabalhador;
    TAMPOP = acrescentar n_indivs_trabalhador;
}
MAX_GERACOES= maior valor max_geraçoes_trabalhador dentre os trabalhadores;
repetir enquanto (geracao <= MAX_GERACOES)
{
  // registrar melhor individuo entre os trabalhadores na geração
  para cada processo trabalhador não finalizado {
```

```
    obter fo_melhor_indiv_trabalhador_geração;
    obter tempo_da_geração_trabalhador;
    if (fo_melhor_indiv_trabalhador_geração é o
        menor valor makespan na geracao) {
        fo_melhor_indiv_geração = fo_melhor_indiv_trabalhador_geração;
        tempo_da_geração = tempo_da_geração_trabalhador;
    }
  }
  print(geracao, fo_melhor_idiv_geração, tempo_da_geração);
  atualizar geração;
}
// receber de cada trabalhador o melhor individuo gerado durante a evolução
// registrar melhor individuo na evolução
para cada processo trabalhador {
  obter melhor_indiv_trabalhador;
  obter geracao_melhor_indiv_trabalhador;
  if ( (fo(melhor_indiv_trabalhador) é o menor valor makespan na evolução)
       ou (o melhor valor makespan obtido ate então é igual ao
           fo(melhor_indiv_trabalhador) e geracao_melhor_indiv_trabalhador
                                        < geracao_melhor_indiv_evolucao))
  {
      melhor_indiv_evolucao = melhor_indiv_trabalhador;
      geracao_melhor_indiv_evolucao = geracao_melhor_indiv_trabalhador;
  }
}
// imprimir melhor individuo na evolução
print(fo(melhor_indiv_evolução),melhor_indiv_evolução(prob),
      geracao_melhor_indiv_evolucao);
// imprimir tempo de evolução
atualizar tempo;
print("Tempo de evolução = ", tempo);

finalizar processo mestre;
} // jspGraspAg_DistrProc_Mestre
```

Pseudo-código trabalhador :

```
jspGraspAg_DistrProc_Escravo (TAMPOP, MAX_GERACOES,
                              TAXA_GULOSA, TAXA_CROSSOVER, TAXA_MUTACAO)
{
inicialisar tempo;
inicializar geracao;

prob = gerar Problema Job-Shop;
registrar processo trabalhador;

enviar ao processo mestre TAMPOP e MAX_GERACOES;
// criar população inicial (sub-populacao)
para 1 ... TAMPOP faca {
  criar_individuo(prob, TAXA_GULOSA);
}
atualizar tempo;
// registrar melhor individuo na geração
obter melhor_indiv_geração;
// registrar melhor individuo na evolução
melhor_indiv_evolução = melhor_indiv_geração;
geracao_melhor_indiv_evolucao = geração;
// enviar ao processo mestre makespan do melhor individuo na geracao
enviar ao processo mestre {
   fo(melhor_indiv_geração);
   tempo; // tempo da geração.
```

```
}
while ( geracao < MAX_GERACOES )
{
      // sortear individuos pais e fazer cruzamento e mutação
      faca de 1 ... n_indivs/2 {
           sortear par_de_indivíduos pais da população;
           sorteio = sortear valor [0,1];
           if (sorteio <= TAXA_CROSSOVER)
             descendentes = gerar_filhos(par_de_indivíduos);
           else descendentes = par_de_indivíduos;
           sorteio = sortear valor [0,1];
           if (sorteio <= TAXA_MUTACAO)
             descendentes = mutar_descendentes(descendentes);
      }
      atualizar tempo;
      // registrar melhor individuo na geração
      obter melhor_indiv_geração;
      // registrar melhor individuo na evolução
      melhor_indiv_evolução = melhor_indiv_geração;
      geracao_melhor_indiv_evolucao = geração+1;
      // enviar ao processo mestre makespan do melhor individuo na geracao
      enviar ao processo mestre {
         fo(melhor_indiv_geração);
         tempo; // tempo da geração.
      }
      definir população sobrevivente, segundo critério Roleta Russa, para a
      geração seguinte;
      re-atualizar melhor_indiv_geração;
      avancar geracao;
}
//enviar ao processo mestre o melhor individuo gerado durante a evolução
enviar ao processo mestre {
       melhor_indiv_evolucao;
       geracao_melhor_indiv_evolucao;
}

finalizar processo trabalhador;
} // jspGraspAg_DistrProc_Escravo
```

6.3.4 Considerações para os Algoritmos Distribuídos

A eficiência dos algoritmos distribuídos foi altamente dependente da decomposição do algoritmo serial em processos independentes. Essa decomposição em processos independentes ou concorrentes levou em conta questões como:

- O tamanho (tempo de execução) dos processos independentes em relação ao tempo de execução do algoritmo serial correspondente. Essa medida é chamada de grão de paralelização.
- O tempo gasto em transmissão dos dados de cada grão para os processadores envolvidos na paralelização.

O tamanho escolhido para o grão de paralelização deve levar em conta os tempos de transmissão dos dados do grão em relação ao seu tempo de execução. Quanto maior for o tempo de execução do grão (maior o grão), menores serão os custos relativos à comunicação,

pois, menor será o número de processos disparados pelo programa paralelo. Problemas da classe *NP-Difícil,* como o problema *Job-Shop* tratado neste projeto, tendem a apresentar um tempo de solução que cresce exponencialmente com o tamanho do problema. Esta característica facilitou a decomposição dos processos e permitiu elevarmos a velocidade de execução dos algoritmos distribuídos. O grão foi dimensionado de tal forma que a transmissão dos dados na rede não comprometesse o desempenho dos algoritmos distribuídos, isto é, os algoritmos distribuídos forneceram pontos positivos que justificaram o esforço da paralelização.

6.4 Análise dos Testes Computacionais Realizados

Os testes computacionais foram realizados executando 5 vezes cada modelo do algoritmo distribuído e 5 vezes o algoritmo seqüencial.

Todos os testes levaram em consideração os seguintes parâmetros para o algoritmo genético :

Parâmetros	
Taxa Gulosa	50 %
Taxa de Cruzamento	95 %
Taxa Mutação	5 %
Tamanho da População	50 indivíduos
Máximo de Gerações	500 gerações
Desvio	1.0 (constante de desvio)
Algoritmos Distribuídos	
Máximo indivíduos / processo	17 indivíduos

Tabela 6.1 – Parâmetros considerados nos testes computacionais

A condição de parada dos algoritmos seqüencial e distribuídos é atingir um número pré-determinado de gerações, sendo este fixado em 500 gerações.

O número máximo de indivíduos a ser produzido e/ou processado em cada processo trabalhador, ou processador, para os algoritmos distribuídos, é de 17 indivíduos.

Para os algoritmos distribuídos, a máquina virtual está configurada com três processadores homogêneos para uma população de 50 indivíduos distribuídos entre estes, sendo 17 indivíduos para dois processadores e 16 indivíduos para um (1) processador.

Para o algoritmo seqüencial, temos uma população de 50 indivíduos a ser evoluída.

Durante o desenvolvimento dos algoritmos distribuídos, todos os testes realizados utilizaram uma instância do problema *Job-Shop* que envolve 10 *jobs* e 5 *máquinas* (problema 10/5 JSP) para apurar a potencialidade dos algoritmos distribuídos propostos em relação ao algoritmo seqüencial. O algoritmo seqüencial foi recodificado e compilado para executar nas máquinas homogêneas que fazem parte do ambiente distribuído.

Na tabela 6.2 estão relacionados os resultados referentes ao problema 10/5 JSP. Para cada algoritmo distribuído comparado são tabelados: a média dos melhores 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 dos algoritmos distribuídos desenvolvidos neste projeto: modelo centralizado ao nível de população (MCPOP), modelo distribuído ao nível de população (MDPOP) e modelo distribuído ao nível de processos (MDPROC) e também tabelamos os valores obtidos do algoritmo seqüencial em cinco rodadas.

Problema 10/5 JSP															
Algoritmo Seqüencial				**Algoritmos Distribuídos**											
				MCPOP				MDPOP				MDPROC			
VM	DP	TM	MV	VM	DP	TM	MV	VM	DP	TM	MV	VM	DP	TM	MV
613	0.0	**281.38**	613	613	0.0	**111.64**	613	613	0.0	**102.91**	613	613	0.0	**94.09**	613

Nota: Para todos os testes foram encontrados como melhor valor *makespan*, durante a evolução dos indivíduos, o valor 613.

Tabela 6.2 – Dados obtidos nos testes computacionais

A relação entre os tempos médios de processamento mostra qual algoritmo evolui uma mesma população de indivíduos em um menor intervalo de tempo. Os algoritmos foram executados em máquinas homogêneas, de mesmo sistema operacional e mesmo compilador. Como já foi dito anteriormente, os algoritmos distribuídos foram implementados e executados em um ambiente distribuído que se comporta como uma máquina virtual PVM que integra três máquinas homogêneas com sistema operacional Conectiva Linux 7.0 Kernel 2.4 interligadas por uma rede de comunicação Ethernet 10 Mbps.

Com um processador e nenhuma sobrecarga de comunicação ou inicialização, para o algoritmo seqüencial, o tempo de execução médio ficou em torno de 238.38 segundos. Para os algoritmos distribuídos, executados na máquina virtual PVM, o tempo de execução médio ficou em torno de 111.64 segundos (MCPOP), 102.91 segundos (MDPOP) e 94.09 segundos (MDPROC). O algoritmo MDPROC teve um ganho de tempo de execução significativo representado aproximadamente um terço do tempo de execução do algoritmo seqüencial. O motivo que levou o algoritmo MDPROC ter menor tempo de execução em relação aos modelos de algoritmos MCPOP e MDPOP foi o menor custo na comunicação entre os processos.

Embora o algoritmo MDPROC tenha menor tempo de processamento em relação aos algoritmos MCPOP e MDPOP, ele não apresenta durante o processo evolutivo melhor qualidade na geração de indivíduos que os demais algoritmos distribuídos. Isso acontece porque a evolução em cada processador ocorre sem nenhum mecanismo de iteração entre os indivíduos de outros processos, ou seja, a população de indivíduos em um processador evolui totalmente isolada da população de indivíduos de outro processador.

Relatamos também que o tempo médio de processamento do algoritmo MCPOP é maior que o tempo médio de processamento do algoritmo MDPOP.

Isso ocorre porque o MCPOP gasta mais tempo com o custo de comunicação durante o tráfego de indivíduos, a cada geração, entre o processo mestre e trabalhadores para:

- Manter centralizada a população geral no processo mestre, que recebe os indivíduos descendentes gerados pelos processos trabalhadores.
- Enviar aos trabalhadores, a partir do processo mestre, indivíduos selecionados para reprodução genética.

A centralização da população permitiu ao algoritmo MCPOP possuir a característica de manter a mesma qualidade na evolução dos indivíduos comparado ao algoritmo seqüencial. O algoritmo MDPOP, durante a propagação dos melhores indivíduos a cada geração entre os processadores ou *ilhas* onde se encontra uma sub-população, é o único fator que contribui para o equilíbrio da qualidade dos indivíduos nas sub-populações durante a evolução.

Com base nas análises realizadas nos testes computacionais para os algoritmos distribuídos convém então concluirmos que o algoritmo distribuído MCPOP garante a qualidade na geração de soluções, mas necessita de memória para manter centralizada a população em um processador. Caso a disponibilidade de memória para manter centralizada a população em um processador seja uma dificuldade, o algoritmo MDPOP seria apropriado, entretanto, teríamos uma convergência mais prolongada para soluções de melhor qualidade comparando a convergência realizada no algoritmo MCPOP, podendo necessitar do usuário um aumento do número de gerações para produção de "boas" soluções.

Um ponto interessante foi quando testamos cada algoritmo distribuído executando-os na máquina virtual PVM composta de apenas um único processador. Os processos trabalhadores inicializados pelo processo mestre, em cada algoritmo, ocuparam um mesmo processador onde o processamento destes aconteceram em *time-sharing* e não em paralelo. Os processos trabalhadores junto com o processo mestre foram executados em um único processador compartilhando intervalos pequenos de tempo de execução entre eles. O tempo médio de execução de cada algoritmo distribuído, nestas condições, se tornaram maior que o tempo médio de execução do algoritmo seqüencial devido aos custos obtidos, especialmente de comunicação, entre os processos executados num mesmo processador.

Veja na tabela 6.3 os tempos médios de processamento dos algoritmos seqüencial e distribuídos, onde os algoritmos distribuídos foram executados na máquina virtual PVM composta de um único processador.

Problema 10/5 JSP			
Tempo Médio de Processamento			
Algoritmo Seqüencial	**Algoritmos Distribuídos**		
	MCPOP	MDPOP	MDPROC
281.38	303.38	294.61	289.03

Tabela 6.3 – Tempos médios de processamento nos testes computacionais

O desenvolvimento dos algoritmos distribuídos para o algoritmo seqüencial e híbrido além de obter computação mais rápida com o paralelismo alcançou um custo/benefício favorável. Ao invés de executarmos o programa seqüencial em uma única máquina de elevado potencial ao nível de processador e memória para evoluir uma população de soluções, a tarefa computacional do programa foi executada de forma paralela em um ambiente com três processadores de menor porte e com memórias de menor capacidade de armazenamento.

A escalabilidade também é um ponto caracterizado para os algoritmos distribuídos, ou seja, é certo que teremos maior desempenho de um programa quando executado em um maior número de máquinas a compor a máquina virtual. Quanto maior o número de máquinas, maior é o número proporcional de processos trabalhadores distribuídos entre as máquinas se tornando menor o número de indivíduos por processador. A carga computacional é distribuída proporcionalmente entre os processadores disponíveis onde a população de soluções é distribuída e processada. Um balanceamento entre o número de mensagens a transferir entre processos, sobrecarga de comunicação, e o aumento do número de processos deverá ser considerado para não prejudicar o desempenho dos algoritmos distribuídos.

7 Considerações Finais

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

7.1 Conclusão

A paralelização do algoritmo genético adotado como técnica de otimização base no algoritmo seqüencial para solucionar o problema *Job-Shop* apresentou resultados significativos. Foram obtidos reduzidos tempos de processamento dos algoritmos distribuídos em relação ao tempo de processamento do algoritmo seqüencial, demonstrando que é possível obter ganhos com a computação paralela, em alguns casos.

A independência dos processos concorrentes que permitiram a distribuição da carga computacional entre processadores de menor porte com menor quantidade de recursos, como memória, para execução do algoritmo genético foi de fato o fator que determinou o bom desempenho dos algoritmos distribuídos. Procuramos explorar a natureza paralela implícita nos algoritmos genéticos que pesquisam um conjunto de soluções por iteração e não uma única solução por iteração. Cada computador passou a processar independentemente uns dos outros as tarefas e/ou soluções que lhes foram encaminhadas fazendo uso dos seus próprios recursos de armazenamento disponíveis, ou seja, não houve um compartilhamento de memória entre os computadores. Os algoritmos distribuídos que criam uma população de indivíduos em cada computador da máquina virtual, armazenam essa população na memória do seu respectivo computador. Embora os processos são executados de forma independente, um tempo deverá ser dispensado para mantermos uma lógica de comunicação entre os processos sobre a rede de comunicação para garantir a funcionalidade do algoritmo genético e a obtenção do melhor indivíduo gerado após o processo evolutivo. A esse tempo dispensado damos o nome de custo de comunicação e ele não afetou o desempenho dos algoritmos distribuídos em relação ao algoritmo seqüencial.

Com um aumento no número de processadores o desempenho deverá continuar melhorando. Um balanceamento entre o número de mensagens a transferir entre processos, sobrecarga de comunicação, e o aumento do número de processos deverá ser considerado para não prejudicar o desempenho dos algoritmos distribuídos.

O algoritmo genético paralelizado permite também aumentar o tamanho dos problemas resolvíveis pela heurística, trabalhando com populações maiores, sem prejuízo do tempo de processamento.

7.2 Propostas de Novas Pesquisas

Como proposta de futuras pesquisas, sugerimos a utilização de comunicação assíncrona nos algoritmos distribuídos para ambientes heterogêneos, devido às cargas computacionais não uniformes ou processadores diferentes.

Outros problemas de otimização combinatória que tenham como base a técnica de otimização heurística algoritmo genético também poderiam ser implementados sem nenhuma modificação na lógica computacional dos algoritmos distribuídos, que apenas paralelizam o algoritmo genético utilizado para solucionar o problema *Job-Shop*.

Outro ponto a ser avaliado é o balanceamento entre a quantidade de comunicação viável e o número de processadores a ser incluído na máquina virtual.

8 Referências Bibliográficas

ANSARI, N. e HOU, E. *Computational Intelligence for Optimization*. Kluwer Academic Publishers, 1997.

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

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

CCUEC, Unicamp e FINEP, MCT. *Treinamento: Introdução ao PVM*. Disponível na Internet via FTP: ftp://ftp.cenapad.unicamp.br/cursos/PVM/apostila.ps.gz, 12 de setembro 2002.

CHIPPERFIELD, A., FLEMING, P. Parallel Genetic Algorithms. IN: ZOMAYA, A. Y. H. *Parallel and Distributed Handbook*. New York: McGraw-Hill, 1996. p. 1118-1143.

COLOURIS, G., DOLLIMORE, J., KINDBERG, T., *Distributed Sytems – Concepts and Design*, Segunda Edição. Addison-Wesley, 1996.

COSTA, J. I. *Introdução aos Algoritmos Genéticos*. In: VII Escola de Informática da SBC Regional Sul. Anais... SBC, 1999.

FEO, T. A., RESENDE, M. G. C. *Greendy randomized adaptive search procedures.* Jornal of Global Optimization, 6:109-133, 1995.

GEIST, A., BEGUELIN, A. et. al. *PVM: Parallel Virtual Machine - A Users' Guide and Tutorial for Networked Parallel Computing.* MIT Press, Massachusetts, 1994. Disponível na Internet na versão on-line: http://www.netlib.org/pvm3/book/pvm-book.html em 12 de setembro de 2002.

HAX, A.; CANDEA, D.: ***Production and Inventory Management,*** Englewood Cliffs, N.J., Prentice-Hall, 1984

MORTON, T.E., PENTICO, D.W. ***Heuristic Scheduling Systems.*** John Wiley & Sons. N.Y., 1993.

NETO, J. F. B. *Os Métodos Heurísticos com Base em Algoritmos Genéticos.* Relatório Técnico, Departamento de Engenharia de Produção, COPPE/UFRJ, Rio de Janeiro, Maio/1997.

OCHI, S. L. *Conhecimentos Heurísticos: Aplicações em Problemas de Otimização*. XIII Jornada de Atualização em Informática, XIV Congresso da Sociedade Brasileira de Computação, DCC/Universidade Federal Fluminense, 1994, p. 16-34.

PACHECO, R.F. e SANTORO, M.C. *Proposta de Classificação Hierarquizada dos Modelos de Solução para o Problema de* ***Job Shop Scheduling.*** GESTÃO E PRODUÇÃO, Revista do Departamento de Engenharia de Produção, Universidade de São Carlos, Abril de 1999. p. 1-15.

PINEDO, M., CHAO, X. *Operations Scheduling With Applications in Manufacturing and Services.* Irwin McGrawn-Hill, 1999.

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.

ROUCARIOL, C. Parallel Branch and Bound Algorithms - an Overview. IN: *Parallel and Distributed Algorithms*. Amsterdam: Elsevier, 1989. p. 153-163.

SCOFIELD, W. C. L. *Aplicação de Algoritmos Genéticos ao Problema Job-Shop*. Relatório Técnico 2 / 2001, DECOM/ICEB/UFOP, Ouro Preto.

SOUZA, M. J. F. *Tópicos Especiais em Inteligência Artificial.* Apostila 1/2001. DECOM/ICEB/UFOP, Ouro Preto.

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

WALTER, C. *Planejamento e Controle da Produção - PCP.* Apostila de aula.- Mestrado em Engenharia de Produção - UFRGS, 1999

Anexo A. Glossário Genético

Este glossário é voltado para a área de algoritmos genéticos e por isso faz uso de um considerável grau de liberdade na definição de termos com relação a seus correlatos na biologia.

Adaptação (ou aptidão) – no campo da biologia, o nível de adaptação mede o quanto um indivíduo está apto para sobreviver no meio em que reside. Em sua analogia com a natureza, os algoritmos genéticos fazem uso do grau da adaptação para representar quão bem um determinado indivíduo (solução) responde ao problema proposto.

Convergência – o grau de convergência é característica das populações dos algoritmos genéticos, e diz respeito à diferença entre a média de adaptação de sua geração atual e suas anteriores. A ascensão deste índice indica que o processo de evolução está efetivamente promovendo a melhora da média de adaptação da população, e sua estabilização em torno de um mesmo valor por muitas gerações normalmente indica que a população se estacionou em um determinado valor médio de adaptação, caso em que a continuação do processo de evolução se torna improdutiva.

Convergência Prematura – a convergência prematura é um problema bastante recorrente na técnica de algoritmos genéticos: ela ocorre quando a diversidade de uma população cai de tal forma que o processo de reprodução gera a cada geração filhos muito semelhantes aos pais, o que causa a convergência da população com média de adaptação em muitos casos pouco adequada e retarda ou até mesmo estanca por completo a evolução.

Cromossomo – um cromossomo consiste de uma cadeia de genes. Visto que tradicionalmente são utilizados genomas com uma cadeia simples de genes, muitas vezes este termo é utilizado, com certa liberdade, como sinônimo para genoma.

Cruzamento – durante o processo de cruzamento os indivíduos previamente selecionados são cruzados com seus pares para gerar filhos.

Diversidade (ou biodiversidade) – característica de uma população de indivíduos, a diversidade diz respeito ao grau de semelhança entre eles.

Evolução – o processo de evolução é o responsável pela busca efetuada pelos Algoritmos Genéticos. A cada passo os indivíduos da população a ele submetida passam pelos processos de seleção, reprodução e mutação, criando assim uma nova geração a partir de sua anterior.

Função de aptidão – a função objetivo recebe como entrada um indivíduo e retorna o grau de adaptação que este apresenta.

Gene – um gene serve como um *container*, um espaço para a alocação de um valor. As características dos indivíduos são definidas a partir dos valores contidos no conjunto de um ou mais genes que as codificam.

Geração – a cada passo do processo de evolução uma nova geração é criada a partir da população anterior, e esta última é atualizada. Para que o processo de evolução possa ser considerado eficiente é necessário que em cada nova geração tenda a ser melhor (mais adaptada) que suas anteriores.

Indivíduo – um indivíduo pertencente a um algoritmo genético representa uma possível solução para o problema a ser tratado.

Mutação – na natureza, falhas no processo de reprodução podem facilmente ocorrer, fazendo com que os filhos apresentem características que não seriam atingíveis através de uma combinação de características dos pais. A este fenômeno dá-se o nome de mutação, e nos algoritmos genéticos esta transformação ocorre sobre indivíduos provindos do processo de reprodução.

População – uma população consiste em um conjunto de indivíduos. Ela apresenta características não observáveis nos indivíduos, tais como grau de diversidade e de convergência.

Reprodução – o processo de reprodução consiste da alocação em pares dos indivíduos selecionados e do cruzamento entre estes. Ele é o responsável pela perpetuação de características ao longo do processo de evolução.

Seleção – nesta etapa onde indivíduos são escolhidos para a reprodução de acordo com seu grau de adaptação. Para isto cada indivíduo é avaliado e os mais aptos da população possuem maior probabilidade de serem selecionados.