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

Marco Antonio Silveira de Souza

Biblioteca de Aplicação Genérica de Algoritmos Genéticos Paralelos Distribuídos

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

Prof. Dr. José Mazzucco Júnior.

Florianópolis, Julho de 2003.

Biblioteca de Aplicação Genérica de Algoritmos Genéticos Paralelos Distribuídos

Marco Antonio Silveira de Souza

Esta dissertação foi julgada adequada para a obtenção do título de Mestre em Ciência da Computação Área de Concentração Sistemas de Computação e aprovada em sua forma final pelo Programa de Pós-Graduação em Ciência da Computação.

_____

Raul S. Wazlawick, Dr.
Coordenador do CPGCC

Banca Examinadora

_____

José Mazzucco Júnior, Dr.
Orientador

_____

Jovelino Falqueto, Dr.
Membro

_____

Luiz Alfredo Soares Garcindo, Dr.
Membro

"A coisa mais nobre que podemos experimentar é o mistério. Ele é a emoção fundamental, paralela ao berço da verdadeira ciência."
(Albert Einsten).

"A matemática é bonita!"
(João Batista Souza).

Ao professor José Mazzucco Júnior, pela sua orientação
e companheirismo.

Ao Corpo Docente da Pós-Graduação em Ciência da Computação da Universidade Federal de Santa Catarina, que contribuiu para o meu aperfeiçoamento pessoal e profissional.

À Unisul por seu apoio.

Em especial:
Aos meus pais e irmãos, pelos incentivos e empolgação.

À minha esposa e minhas filhas, Juliana e Cristine que pacientemente suportaram minhas ausências.

Por fim agradeço a Deus as oportunidades e por ter condições e capacidade de aproveitá-las.

Sumário

Lista de Figuras

Lista de Tabelas

Resumo

Neste trabalho apresentaremos o desenvolvimento de uma biblioteca de funções de Algoritmos Genéticos (AG) utilizando o paradigma da passagem de mensagens. Isto torna viável a sua utilização em sistemas distribuídos, mais especificamente em cluster de computadores. Algoritmos Genéticos são reconhecidos por sua eficiência na área de otimização e tem aplicações nas mais diversas áreas do conhecimento.

O protótipo apresentado define uma interface de acesso ao usuário onde ele pode implementar suas aplicações e depois executá-las sobre um ambiente paralelo de forma fácil e transparente.

Palavras-chave: Algoritmos Genéticos, computação paralela e distribuída, otimização.

Abstract

In this work we will present the development of a library of functions of genetic algorithms (AG) using the paradigm of messages transmition. This makes possible its use in distributed systems, more specifically in cluster of computers. Genetic algorithms are recognized by efficiency in the its optimization area and it has applications in several areas of knowledge.

The prototype presented defines an access interface to the user where he can implement its applications and then execute them on a parallel environment in an easy and transparent way.

Key-words: Genetic algorithms, parallel computing, optimization.

1 Introdução

Os Algoritmos Genéticos são uma das mais conhecidas e originais técnicas de resolução de problemas dentro do que se define como “Computação Evolucionária” (ou “Algoritmos Evolutivos”), termo que agrupa os Algoritmos Genéticos, as Estratégias Evolutivas e a Programação Evolutiva. A razão por trás deste fato é que este tipo de algoritmo é relativamente simples de ser implementado e eficiente em sua busca por melhoria. Sob a ótica da computação, os Algoritmos Genéticos modelam o fenômeno natural de herança genética e a teoria da evolução proposta por Darwin [CRU, 2002].

Este tipo de algoritmo parte de um conjunto de soluções possíveis de um dado problema, denominado de população inicial, que vão se reproduzir e competir pela sobrevivência. Os que conseguem transferem suas características para as novas gerações de soluções que conseqüentemente devem ser melhores que as anteriores.

A implementação paralela de um AG possibilita uma maior velocidade de execução dos AGs e, dependendo da forma como isto é feito, melhores resultados são obtidos quando comparados com um AG tradicional [PAZ, 1997]. A computação paralela possui a vantagem do alto desempenho, porém existem fatores, como alto custo de aquisição e manutenção, e dependência do fabricante, que diminuem esta vantagem. Uma forma de contornar este problema é através da utilização de sistemas distribuídos como plataforma de execução paralela através de troca (passagem) de mensagens [LIN, 1998].

Entre os diversos modelos de troca de mensagens existentes em sistemas distribuídos, o padrão MPI (*Message-Passing Interface*) é um esforço de padronização destes diversos paradigmas, que foi sugerida por um grupo de trabalho formado por representantes da indústria, governo e universidades. Ela é uma biblioteca com funções para troca de mensagens, responsável pela comunicação e sincronização de processos. Dessa forma, os processos de um programa paralelo podem ser escritos em uma linguagem de programação seqüencial, usualmente C ou Fortran.

Neste documento propomos a implementação de uma biblioteca de funções de Algoritmos Genéticos apta a trabalhar sobre um ambiente paralelo/distribuído utilizando o modelo de passagem de mensagens. Desta maneira os detalhes de implementação do algoritmo genético e a forma de sua paralelização são completamente transparentes ao

usuário. Assim, a biblioteca fornece uma interface fácil de operar, não exigindo do usuário final nenhum conhecimento sobre programação paralela para a sua utilização.

Os capítulos que se seguem apresentam os temas abordados sendo que inicialmente apresentamos os fundamentos de Computação Paralela e Distribuída (Capítulo 2), passando pela definição do padrão MPI (Capítulo 3) e dos Algoritmos Genéticos (Capítulo 4). A seguir, apresentamos nosso protótipo desenvolvido (Capítulo 5) dando os detalhes de sua implementação juntamente com um exemplo de aplicação. Por fim, apresentamos as conclusões e os projetos futuros (Capítulo 6).

2 Fundamentos de Computação Paralela e Distribuída

2.1 Considerações Iniciais

A literatura apresenta diversas definições para computação paralela (ou processamento paralelo), dentre as quais cabe citar a sugerida por Almasi [ALM, 1994]: "computação paralela constitui-se de uma coleção de elementos de processamento que se comunicam e cooperam entre si e com isso resolvem um problema de maneira mais rápida", e a apresentada por Quinn [QUI, 1987]: "computação paralela é o processamento de informações que enfatiza a manipulação concorrente dos dados, que pertencem a um ou mais processos que objetivam resolver um único problema".

As arquiteturas seqüenciais de von Neuman (tradicionalmente aceitas) conseguiram ao longo dos últimos anos um grande avanço tecnológico, sendo largamente aprimoradas no decorrer deste tempo, mas estas arquiteturas ainda demonstram deficiências quando utilizadas por aplicações que necessitam de maior poder computacional.

Dentre os principais motivos para o surgimento da computação paralela, destaca-se a necessidade de aumentar o poder de processamento em uma única máquina. Segundo [SOU, 1996], a computação paralela apresenta diversas vantagens em relação à seqüencial, tais como:

- alto desempenho para programas mais lentos;
- soluções mais naturais para programas intrinsecamente paralelos;
- maior tolerância a falhas;
- modularidade.

Apesar das vantagens apresentadas anteriormente, existem alguns fatores negativos na utilização da computação paralela que devem ser considerados:

- maior dificuldade na programação;
- necessidade de balanceamento de cargas para melhor distribuição dos processos nos vários processadores;

- intensa sobrecarga no sistema (por exemplo, na comunicação entre processos), que impede que se tenha um *speedup*[1] ideal;
- necessidade de sincronismo e comunicação entre processos, o que gera certa complexidade.

2.2 Arquiteturas de Máquinas Paralelas

Uma arquitetura paralela, conforme Duncan [DUN, 1990], fornece uma estrutura explícita e de alto nível para o desenvolvimento de soluções utilizando o processamento paralelo, através da existência de múltiplos processadores, estes simples ou complexos, que cooperam para resolver problemas através de execução concorrente.

As máquinas paralelas são formadas por nós processadores interligados fisicamente através de uma rede de interconexão, cujo objetivo é proporcionar grande poder computacional através da cooperação de processadores na execução de eventos concorrentes, oferecendo confiabilidade e uma boa relação custo/desempenho, suprindo a necessidade de um processamento intensivo.

Existem muitas maneiras de se organizar computadores paralelos. Para que se possa visualizar melhor o conjunto de possíveis opções de arquiteturas paralelas, é interessante classificá-las. Segundo Ben-Dyke [BEN, 1993], uma classificação ideal deve ser:

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

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

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

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

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

[1] Medidas de desempenho, que permitam a análise do ganho obtido com o aumento do total de processadores utilizados.

Muito já foi desenvolvido em termos de *hardware* paralelo, e várias classificações foram propostas [ALM, 1994] [BEN, 1993] [DUN, 1990]. A mais utilizada e aceita pela comunidade computacional é a classificação de Flynn [FLY, 1972].

2.2.1 Classificação de Flynn

A classificação proposta por Flynn [FLY, 1972], embora muito antiga, é amplamente adotada e baseia-se no fluxo de instruções e no fluxo de dados (*o fluxo de instruções eqüivale a uma seqüência de instruções executadas em um processador sobre um fluxo de dados aos quais estas instruções estão relacionadas*) para classificar um computador em quatro categorias:

- SISD (*Single Instruction Stream/Single Data Stream*) - Fluxo único de instruções/Fluxo único de dados: corresponde ao tradicional modelo von Neumann (*máquinas possuem uma unidade de processamento e uma unidade de controle*). Um processador executa seqüencialmente um conjunto de instruções sobre um conjunto de dados. Mais difundido em computadores comerciais e/ou convencionais.
- SIMD (*Single Instruction Stream/Multiple Data Stream*) - Fluxo único de instruções/Fluxo múltiplo de dados: envolve múltiplos processadores (escravos) sob o controle de uma única unidade de controle (mestre), executando simultaneamente a mesma instrução em diversos conjuntos de dados. Arquiteturas SIMD são utilizadas, por exemplo, para manipulação de matrizes e processamento de imagens, ou seja, aplicações que geralmente repetem o mesmo cálculo para diferentes conjuntos de dados.
- MISD (*Multiple Instruction Stream/Single Data Stream*) - Fluxo múltiplo de instruções/Fluxo único de dados: envolve múltiplos processadores executando diferentes instruções em um único conjunto de dados. Geralmente, nenhuma arquitetura é classificada como MISD, isto é, de difícil aplicação prática. Alguns autores consideram arquiteturas *pipeline* como exemplo deste tipo de organização.

- MIMD (*Multiple Instruction Stream/Multiple Data Stream*) - Fluxo múltiplo de instruções/Fluxo múltiplo de dados: envolve múltiplos processadores executando diferentes instruções em diferentes conjuntos de dados, de maneira independente. Este modelo abrange a maioria dos computadores paralelos e os sistemas distribuídos, ou seja, mais adequado para as aplicações de uso geral.

2.2.2 Classificação baseada no acesso a memória

A categoria MIMD engloba uma grande diversidade de máquinas paralelas, no qual, uma das formas mais usadas de classificá-las é o critério baseado no acesso à memória, isto é, a forma como a memória central está fisicamente organizada e o tipo de acesso que cada processador tem à totalidade da memória, classificando esta arquitetura em computadores MIMD de memória compartilhada e memória distribuída [ALV, 2002].

Na classe de arquitetura MIMD de memória compartilhada incluem-se as máquinas com múltiplos processadores que compartilham um espaço de endereços de memória comum (máquinas multiprocessadas). Os *multiprocessadores* caracterizam-se pela existência de uma memória global e única, a qual é utilizada por todos os processadores (hardware fortemente acoplados), os processos comunicam-se via memória compartilhada.

Contudo, na outra classe de arquitetura MIMD de memória distribuída incluem-se as máquinas formadas por várias unidades processadoras (nós), cada uma com a sua própria memória privativa (hardware fracamente acoplados), também conhecidos como *multicomputadores*. Em virtude de não haver compartilhamento de memória, os processos comunicam-se via troca de mensagens. No ambiente físico desse trabalho, os multicomputadores destacam-se como unidades alvo.

2.2.3 Multicomputadores

Um multicomputador é formado por vários computadores (nós autônomos), unidades constituídas por um processador, memória local, dispositivos de entrada/saída (não necessariamente) e de suporte físico para a comunicação com outros nós, ligados por uma rede de interconexão, conforme a figura abaixo [COR, 1999].

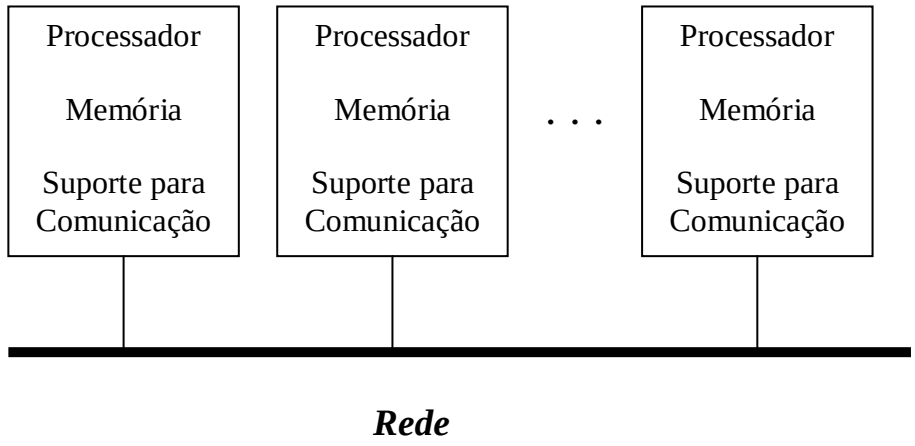

Figura 2.1 – Multicomputador

Neste tipo de arquitetura, cada computador executa seu próprio programa, e o mesmo acessa a memória local diretamente e pode enviar e receber mensagem sobre a rede de comunicação. As mensagens são usadas para a comunicação entre um processador e outro ou para ler e escrever na memória remota. O custo para acessar a memória local, geralmente é menor que para acessar a memória remota. Todavia, o custo de enviar uma mensagem é independente da localização do nó e de outros tráfegos na rede de comunicação, mas depende do tamanho da mensagem.

Neste ponto, cabe ressaltar um fator que pode prejudicar o desempenho desses sistemas (redes físicas dos multicomputadores e dos mecanismos de comunicação), o tempo de latência de rede, conseqüência da duração da troca de mensagens.

Entre algumas características dos multicomputadores, que servem para diferenciá-los de outras máquinas paralelas, tornando a sua utilização mais difundida, destacam-se:

- Grande número de nós homogêneos;
- Proximidade física dos nós;

- Comunicação entre os processos através da troca de mensagens;
- Alto grau de paralelismo real.

Em um multicomputador [JUN, 1999], a interação entre os seus nós (processadores autônomos com memórias privadas, de onde são obtidas suas instruções e dados) é alcançada através do envio de mensagens que trafegam pelo sistema por meio de uma rede de interconexão de alta velocidade. As características funcionais dessas redes, representam fatores determinantes no desempenho de um sistema multicomputador, podendo ser classificadas de acordo com a natureza das redes:

- **Redes estáticas**: rede cuja topologia não pode ser alterada. Apresenta uma topologia fixa, onde as ligações entre os seus nós processadores são estabelecidas na construção do sistema e não podem ser reconfiguradas. Os canais são permanentes, podendo haver desperdício de banda passante[2], uma vez que muitas estações não vão enviar mensagens nos intervalos a ela destinados.
- **Redes dinâmicas**: rede cuja topologia pode ser alterada, através de programação de chaves. Não possuem uma topologia fixa, como as estáticas. As ligações entre os seus nós processadores são estabelecidas por meio de comutadores de conexões (circuitos de chaveamento eletrônico), cuja manipulação permite a atribuição de canais físicos temporários que podem ser criados de acordo com as características de comunicação de um determinado problema [JUN, 1999], ou seja, a alocação do canal é realizada por demanda de envio de mensagens.

As redes de interconexão dinâmicas fundamentais para o multicomputador alvo deste trabalho são o *barramento* e o *crossbar*.

Barramento

Este tipo de topologia (**Fig.** 2.3) é bastante semelhante ao conceito de arquitetura de barra em um sistema de computador, onde todas as estações (nós) se ligam ao mesmo meio de transmissão [SOA, 1997]. É um modelo bastante flexível, de conexão simples e de baixo custo, pois permite adição e subtração de nós, sem a necessidade de efetuar

[2] intervalo de freqüências que compõem um sinal elétrico.

profundas alterações no sistema. Uma das principais desvantagens deste modelo, está relacionada com a extensibilidade, ou seja, este tipo de rede não funciona muito bem com um grande número de nós devido a sua baixa capacidade de transmissão, e outra relacionada com o fator de tolerância a falhas e/ou faltas, isto é, uma interrupção da comunicação entre todos os nós, decorrente de uma eventual falha no *barramento* [JUN, 1999]. O desempenho do sistema pode aumentar significativamente, se as comunicações através do *barramento* não forem freqüentes.

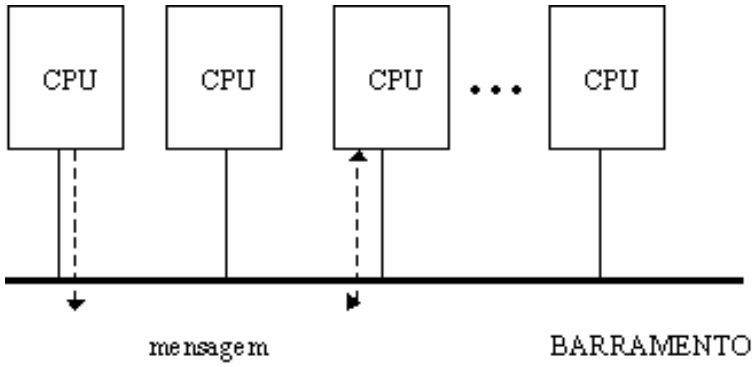

Figura 2.2 – Multicomputador baseado em barramento

Crossbar

Um *crossbar* N x M possui N entradas dispostas horizontalmente e M saídas dispostas verticalmente, construindo uma rede do tipo grelha, na qual a interseção de duas linhas representa um comutador, que controla a conexão entre a entrada e a saída correspondentes. Na figura 2.4, é destacada uma conexão entre a entrada *E4* e a saída *S3*, resultante do fechamento do comutador correspondente. Nessa situação, o estabelecimento de uma nova conexão, envolvendo a entrada *E4* e a saída *S3*, não seria possível [JUN, 1999]. O *Crossbar* permite a conexão de quaisquer entradas N a quaisquer saídas M, onde cada linha de entrada ou saída pode participar de apenas uma conexão por vez. Este modelo é bastante poderoso, sem contenda (*conflitos*) quando comparada ao *barramento*, porém uma desvantagem que merece destaque está

relacionada com o seu crescimento, isto é, o seu custo cresce consideravelmente quando aumenta muito o seu tamanho, tornando-se uma rede muito cara.

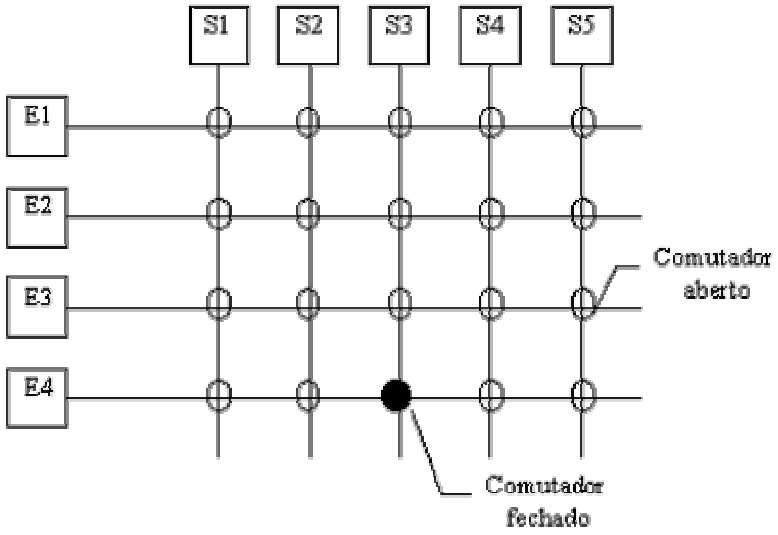

Figura 2.3 – Dispositivo crossbar

2.3 Sistemas Computacionais Distribuídos

No início da década de 70, com o aparecimento das tecnologias de redes de computadores com maior desempenho e maior confiabilidade, foi possível o desenvolvimento dos sistemas computacionais distribuídos. Com o avanço tecnológico das redes de comunicação e o crescente aumento da potência computacional dos computadores pessoais e das estações de trabalho, vários projetos foram desenvolvidos tornando os sistemas distribuídos mais eficazes e difundidos. Entre os exemplos desses projetos, destacam-se: Amoeba, Mach, Chrous e o TRICE [TAN, 1997] [COL, 1994] [MUL, 1993].

Um sistema computacional distribuído, conforme Tanenbaum [TAN, 1997] e Colouris [COL, 1994], é uma coleção de computadores autônomos, interligados por uma rede de comunicação e equipados com um sistema operacional distribuído, que permitem o compartilhamento transparente de recursos existentes no sistema, isto é, para o usuário final não é visível a existência de múltiplos recursos. Esta abordagem descreve o *sistema operacional distribuído* como o elemento responsável por manter as características necessárias utilizando como meio de comunicação a rede e o *software*

como elemento determinante, o que caracteriza um sistema distribuído. A proposta de um *sistema operacional distribuído* é fazer com que este conjunto de máquinas interligadas pareça para seus usuários como se fosse uma única máquina, onde os mesmos (usuários) não tomam conhecimento de onde seus programas estão sendo processados e de onde seus arquivos estão armazenados. Esta proposta difere de um *sistema operacional de rede*, onde o usuário sabe em que máquina ele está, sendo necessário saber a localização de um recurso disponível para poder utilizá-lo, ou seja, os mesmos sabem da existência das várias máquinas da rede, podem abrir sessões em máquinas remotas e transferir dados de uma máquina remota para a máquina local.

Outra definição feita por Mullender [MUL, 1993], é o fato de que um sistema distribuído não deve ter pontos críticos de falha, característica que faz com um sistema distribuído leve vantagem em relação a um sistema centralizado.

Os sistemas distribuídos apresentam inúmeras vantagens, que os tornam mais difundidos: compartilhamento de recursos (componentes de software e hardware que estão disponíveis para a utilização do usuário), flexibilidade (crescimento, expansão da capacidade computacional do sistema, sistema aberto), confiabilidade (disponibilidade do sistema), performance (poder de computação total maior), escalabilidade (sistema se comporta da mesma forma independente do número de máquinas) e transparência (visão de um sistema único), sendo esta última uma característica fundamental e compartilhada pela maioria dos autores [TAN, 1997] [COL, 1994] [MUL, 1993]. Tolerância a falhas (ou faltas) e concorrência são também abordados por [COL, 1994] [MUL, 1993].

Por outro lado, os sistemas distribuídos apresentam também algumas desvantagens, tais como: carência de software para sistemas distribuídos, surgimento de "gargalos" representados pelas redes de comunicação (*podem saturar*) e falta de segurança (*fácil acesso*) apresentada pelo compartilhamento de dados e grande número de usuários.

2.4 Considerações Finais

As diversas áreas na qual a computação se aplica demandam cada vez mais poder computacional, esta necessidade contribuiu consideravelmente para o surgimento da computação paralela. Como o avanço tecnológico e consequentemente o aumento do

desempenho nas arquiteturas de von Neumann (*filosofia seqüencial etc.*) é uma tarefa árdua e cara, imposta pela limitação tecnológica, a computação paralela compõe uma alternativa mais barata, elegante e aprimorada, principalmente para problemas essencialmente paralelos [SOU, 1996].

Classificações de arquiteturas surgiram em conseqüência do desenvolvimento de várias máquinas paralelas, sendo que a classificação proposta por Flynn ainda é bastante utilizada e aceita pela comunidade computacional. Esta Classificação considera o processo computacional como a execução de uma seqüência de instruções sobre um conjunto de dados. A execução seqüencial de instruções é visto como um fluxo único de instruções controlando um fluxo único de dados. A introdução de fluxos múltiplos de dados ou de fluxos múltiplos de instruções é que faz surgir o paralelismo. A classificação de Duncan têm por objetivo resolver os problemas da classificação de Flynn, utilizando termos mais genéricos, classificando as arquiteturas em síncronas e assíncronas [DUN, 1990]. Denominam-se arquiteturas síncronas aquelas arquiteturas paralelas que coordenam as operações por meio de relógios globais, unidade de controle central ou controladores de unidades vetoriais. As arquiteturas assíncronas são caracterizadas por um controle de *hardware* descentralizado, de maneira que os processadores são independentes entre si.

O desempenho dos computadores pessoais e das estações de trabalho vem aumentando significativamente nos últimos anos, sendo assim, quando interligados por uma rede de alta velocidade e alta confiabilidade, podem ser aplicados para solucionar uma variedade de aplicações que necessitam de alto poder computacional.

Os sistemas computacionais distribuídos tornaram-se muito populares e muitas linhas de pesquisa têm sido desenvolvidas com o objetivo de melhorar cada vez mais o compartilhamento de recursos com transparência, desempenho e confiabilidade.

Estas definições são suficientemente vagas para levantar a habitual questão do que é que distingue um sistema paralelo de um sistema distribuído. A fronteira é difícil de traçar, mas tem a ver com os objetivos que cada um deles pretende atingir. Num sistema paralelo, a ênfase é colocada no desempenho da execução, enquanto num sistema distribuído o objetivo é suportar um conjunto de serviços de forma transparente à sua localização, preocupando-se com questões como a tolerância a falhas, a proteção e autenticação de clientes e servidores. Esta maior funcionalidade dos modelos de

programação dos sistemas distribuídos penaliza-os, normalmente, em termos de desempenho.

A utilização de sistemas computacionais distribuídos, com conceitos utilizados pela computação paralela, fez com que trabalhos fossem desenvolvidos para explorar o grande potencial da computação paralela utilizando os sistemas distribuídos. Baseando-se nesses princípios, ambientes de passagem de mensagem foram aperfeiçoados e/ou criados, que serão descritos no próximo capítulo.

3 MPI

3.1 Considerações Iniciais

Message Passing Interface (MPI) é um sistema de passagem de mensagem padronizada e portável, projetado por um grupo de pesquisadores representando indústria e universidades, para funcionar em uma ampla variedade de computadores paralelos. O padrão define a sintaxe e a semântica de uma biblioteca de rotinas feitas para diversos usuários escreverem programas portáveis com passagem de mensagens em Fortran 77 ou C. Várias implementações bem testadas e eficientes de MPI já existem, incluindo algumas que são gratuitas e de domínio público [SNI, 1996].

3.2 Origem

Em abril de 1992 o Centro de Pesquisa em Computação Paralela no estado da Virginia (EUA) promoveu um workshop intitulado "Padrões para Passagem de Mensagens Em Um Ambiente de Memória Distribuída" [WAL, 1992] dando origem ao esforço da padronização MPI. As discussões continuaram a partir deste encontro, via correio eletrônico, formando assim o Fórum MPI, que durante os meses seguintes aprimoraram as idéias iniciais até ser lançado o primeiro esboço deste padrão em novembro de 1992 e uma revisão completa em fevereiro de 1993. Na conferência de Supercomputação de 1993 o padrão MPI foi apresentado e após um período de discussões públicas, que resultou em algumas modificações, a versão 1.0 foi realizada (MPI-1) em junho de 1994.

Em 1997 uma segunda versão do MPI foi lançada (MPI-2) e provê características adicionais não prevista na primeira implementação tais como ferramentas para I/O paralelo, extensões para Fortran 90 e C++ e gerenciamento dinâmico de processos. Até o presente momento, algumas implementações do MPI incluem parte do padrão MPI-2 mas uma implementação completa deste padrão ainda não existe [PAC, 2001].

Hoje o padrão MPI conta com a participação de mais de 80 colaboradores representando indústrias e universidades oriundos dos Estados Unidos e Europa. Uma das características iniciais do MPI era aproveitar algumas idéias já existentes e bem sucedidas em alguns sistemas de passagem de mensagem já existentes. Por este motivo MPI foi fortemente influenciado pelos trabalhos desenvolvido por empresas como Cray, Convex, IBM, Intel, NEC, nCUBE e Thinking Machine, bem como das idéias vindas de bilbiotecas como PVM, p4, Zipcode, PARMACS, Chamaleon, TCGMSG e Express.

3.3 Objetivos

Segundo [PAC, 1998] os objetivos do MPI é desenvolver um padrão amplo para escrita de programas usando passagem de mensagens. Desta forma a interface deverá estabelecer um padrão prático, portável, eficiente e flexível para passagem de mensagens. Uma Lista de objetivos aparecem abaixo:

- desenhar uma interface para programação de aplicações;
- permitir comunicação eficiente;
- permitir implementações que possam ser usadas em ambientes heterogêneos;
- ter uma interface apropriada para programação em C e Fortran;
- prover uma interface de comunicação confiável. O usuário não precisa se preocupar com falhas de comunicação;
- definir uma interface não muito diferente de uso corrente como PVM, NX, Express,p4,etc., e fornecer extensões que permitam uma maior flexibilidade;
- definir uma interface que possa ser implementada em diferentes plataformas com mudanças pouco significantes no sistema de comunicação básico; e
- a interface deverá ser desenhada para permitir comunicação segura.

3.4 Características

Muito dos conceitos MPI não são novos em programação paralela, foram apenas padronizados. Cada implementação MPI fornece uma biblioteca de funções (ou subrotinas no caso de Fortran) que uma aplicação pode usar. O padrão inclui [DAV, 2002]:

- comunicação ponto a ponto;
- operações coletivas;
- grupos de processos e contextos de comunicação. Uma maneira de formar e manipular grupos de processos e definir contextos de comunicação únicos (útil para o desenvolvimento e manutenção de bibliotecas);
- topologia de processos - funções que permitem o mapeamento de processos em algumas topologias (reais ou virtuais) como em uma pipe-line, grid cartesiano, etc;
- ligações para Fortran e C; e
- gerenciamento e pesquisa de ambiente que permite processos a iniciarem, questionarem e mudarem seus contextos de comunicação, estrutura, etc., enquanto executam. Eles são essenciais para programas com passagem de mensagens altamente portáveis.

Ausências do padrão (até o momento) são:

- operações explícitas para memória compartilhada;
- especificações para sistemas operacionais;
- ferramentas para construção de programas;
- gacilidades para debuger;
- suporte explícito a threads;
- suporte ao gerenciamento de tarefas; e
- funções para I/O.

3.5 Conceitos Básicos sobre MPI

MPI define um conjunto de 129 rotinas que juntas oferecem os serviços de Comunicação ponto a ponto, comunicação coletiva, suporte a grupos de processos, contextos de comunicação e topologia de processos.

Nesta seção iremos discutir mais a fundo os principais conceitos do padrão MPI partindo da idéia de Grupos, Contextos e Comunicadores. Estas definições são vistas primeiro pois elas são a base de funcionamento para as rotinas de comunicação Ponto a

Ponto e rotinas de comunicação Coletivas, bem como para a aplicação de Topologia de Processos.

3.5.1 Grupos, Contextos e Comunicadores

3.5.1.1 Gupos de processos

Um grupo de processos em MPI é uma coleção de processos onde cada um possui seu próprio identificador que na nomenclatura oficial é referenciada como rank. Sendo n o total de processos que compõem um grupo, então os ranks deste grupo variam de 0 a n-1.

Segundo [LUS, 2003] grupos de processos podem ser usadas em duas importantes formas. Primeiro, eles podem ser usados para especificar quais processos estão envolvidos em uma operação de comunicação coletiva, como um broadcast. Segundo, eles podem ser usados para introduzir paralelismo de tarefas em uma aplicação, assim diferentes grupos fazem diferentes tarefas. Se isto é feito carregando diferentes códigos executáveis dentro de cada grupo, então referencia-se isto como paralelismo de tarefas MIMD. Alternativamente, se cada grupo executa seções diferentes dentro do mesmo código executável, então se referencia isto como paralelismo de tarefas SPMD (também conhecido como paralelismo de controle).

3.5.1.2 Contextos de Comunicação

Os contextos de comunicação foram inicialmente propostos para permitir a comunicação de processos através de canais de fluxo de dados distintos. Contextos de comunicação asseguram que uma mensagem enviada por um processo não seja incorretamente recebida por outro processo. Assim forma-se famílias de mensagens umas separadas das outras.

Uma situação onde este fato pode ocorrer é quando, por exemplo, uma mesma aplicação faz referência externa a outras bibliotecas de funções. Se esta aplicação compartilha o mesmo ambiente (contexto) para a comunicação de todas a as suas

funções, incluindo ai as fornecidas pela biblioteca externa, então este ambiente deixa de ser seguro para a troca de mensagens.

O usuário nunca faz operações explícitas sobre contextos (não há tipo de dados "contexto" visível ao usuário), entretanto contextos são mantidos dentro de comunicadores do lado do usuário, de forma que mensagens enviadas através de um comunicador conhecido podem somente ser recebidas através do comunicador corretamente adequado. MPI fornece uma rotina coletiva em um comunicador para pré-alocar um número de contextos para uso dentro do escopo do comunicador, este pode então ser usado pelo sistema MPI, sem uma sincronização a mais, quando o usuário cria duplicação ou sub-grupos usando o comunicador. O programa está correto, desde que estas operações ocorrem na mesma ordem em todos os processos que possuem o comunicador [LUS, 2003].

Uma vez que se propõe aqui a implementação de uma biblioteca de funções para Algoritmos Genéticos, esta característica de contextos de comunicação fornecida pelo MPI é particularmente importante.

3.5.1.3 Comunicadores

A noção de contextos e grupos são combinados em um simples objeto chamado comunicador [GRO, 1994]. Em uma comunicação coletiva ou em uma comunicação ponto a ponto entre membros de um mesmo grupo, somente um grupo precisa ser especificado, e os processos de origem e destino são fornecidos pelos seus números de identificação (rank) dentro deste grupo.

Em uma comunicação ponto a ponto entre processos de diferentes grupos, dois grupos devem se especificados. Neste caso os processos fonte e destino devem fornecer suas identificações dentro dos respectivos grupos [FOR, 2003].

3.5.2 Topologia de Processos

Grupos em MPI não possuem hierarquia. Porém podem ser estruturados conforme uma topologia de interconexão qualquer. A necessidade deste tipo de estruturação surge

porque vários programas paralelos, especialmente aqueles baseados em algoritmos matemáticos, torna-se mais simples e fáceis de se expressar quando os processos podem ser interligados de acordo como a topologia especificada pelo algoritmo. Alem disso, a especificação de topologias em software (chamadas também de topologias virtuais) podem servir como informação para o mapeamento correto dos processos aos processadores do sistema e aumentar, desta forma, o desempenho do programa paralelo.

Topologias para grupos de processos são definidas através de um grafo no qual os vértices correspondem aos membros do grupo e os arcos representam as ligações entre cada um dos membros. Além da topologia de grafos, que é a mais genérica, MPI também possui uma série de funções para tratar da topologia do tipo Grid Cartesiano, uma das mais utilizadas em programas paralelos. Em MPI, grupos podem ter ou não uma topologia associada. Para aqueles grupos que possuem topologia, cada membro tem associado, além do rank, uma localização específica dentro da topologia do grupo. MPI mantém esta correspondência entre identificadores de tal forma que, a partir do rank de um processo em um grupo, seja possível se determinar qual o lugar que este membro ocupa dentro da topologia do grupo e vice-versa.

3.5.3 Comunicação Ponto-a-Ponto

Segundo [SNI, 1996] a comunicação Ponto-a-Ponto é definida como a transmissão de dados entre dois processos, um enviando e outro recebendo. Este é o mecanismo básico de comunicação do MPI. A seguir serão discutidos os principais fundamentos da comunicação ponto-a-ponto.

3.5.3.1 Comunicação Bloqueante e Não Bloqueante

Uma função de envio bloqueante somente retorna quando o buffer da mensagem (fornecido como parâmetro a função) pode ser reutilizado pelo processo. Quando uma mensagem é enviada de forma não bloqueante, a função de envio pode retornar antes que o buffer possa ser alterado novamente. Deve-se garantir que qualquer alteração no

conteúdo do buffer somente seja realizada quando há certeza de que não se irá mais afetar os dados da mensagem.

Na recepção bloqueante de mensagens, o processo destinatário é bloqueado até que a mensagem desejada esteja armazenada no buffer fornecido pelo processo. A função de recepção não-bloquante, no entanto, pode retornar antes que a mensagem tenha sido recebida. Neste caso, uma função de verificação terá de ser executada para indicar quando a mensagem estará, realmente, disponível ao receptor.

3.5.3.2 Modos de Comunicação

MPI estabelece quatro modos de comunicação, que indicam como o envio de uma mensagem deve ser realizado (para cada um dos modos de comunicação existe associada uma versão bloqueante e não bloqueante da rotina de envio da mensagem).

No modo pronto (ready) a função de envio só ocorre com sucesso caso a recepção da mensagem já tenha sido iniciada pelo processo destinatário.

No modo padrão (standard) mensagens podem ser enviadas independentemente da função de recepção ter sido chamada ou não. Caso o envio da mensagem ocorra antes que a função de recepção tenha sido iniciada, a mensagem poderá ser armazenada internamente pelo sistema ou a função de envio poderá bloquear o processo remetente até que a mensagem seja recebida pelo processo destino.

O modo de envio buferizado (buffered) é similar ao modo padrão, porém a mensagem deve obrigatoriamente ser armazenada pelo sistema caso a função de recepção não tenha sido iniciada no momento de envio da mensagem.

Por fim, no modo síncrono a função de envio somente retorna no momento em que a recepção da mensagem já tenha sido iniciada pelo processo destinatário [MAZ, 1996]. O receptor deve enviar uma confirmação de recebimento da mensagem, de maneira que o transmissor possa ter certeza de que a mensagem foi recebida. Este modo penaliza o desempenho na troca de mensagens, porém consegue-se maior confiabilidade na entrega da mensagem e evita-se sobrecarga da rede por mensagens não recebidas. Além disso, possibilita que o comportamento de um programa paralelo seja melhor controlado, facilitando a sua depuração.

De acordo com o Fórum MPI [MPI, 2003], a tabela abaixo descreve um resumo das rotinas para comunicação ponto-a-ponto disponíveis no MPI.

		Padrão	**Síncrono**	***Bufferizado***	***Ready***
Bloqueantes		MPI_Send MPI_Recv	MPI_Ssend	MPI_Bsend	MPI_Rsend
	Comunicação em dois sentidos	MPI_Sendrecv MPI_Sendrecv_replace			
Não Bloqueantes		MPI_Isend MPI_Irecv	MPI_Issend	MPI_Ibsend	MPI_Irsend
	Requisições Persistentes	MPI_Send_init MPI_Recv_init	MPI_Ssend_init	MPI_Bsend_init	MPI_Rsend_init

Tabela 3.1 – Rotinas de Comunicação Ponto a Ponto do MPI

3.5.4 Mensagem

Em uma operação genérica de send e receive, uma mensagem consiste de um envelope, indicando os processos de origem e destino, e um corpo, contendo o dado atual a ser enviado.

MPI usa três informações para caracterizar o corpo de uma mensagem em uma forma flexível [PAC, 2001]:

1. Buffer - O endereço inicial da memória onde o dado de saída se encontra (para uma operação de send) ou onde o dado de entrada será armazenado (operação receive).
2. Datatype - O tipo de dado a ser enviado. Em casos simples isto é um tipo elementar como um inteiro ou um real. Em aplicações mais avançadas, este pode ser um tipo definido pelo usuário construído a partir de tipos básicos. Isto pode ser imaginado a grosso modo como uma estrutura em C, e pode conter dados em qualquer lugar, ou seja, não necessariamente em locais contíguos da memória.

Esta habilidade de fazer uso de tipos definidos pelo usuário permite completa flexibilidade na definição do conteúdo de uma mensagem.

3. Count - O número de itens do tipo Datatype a ser enviado.

Por exemplo (A,300,MPI_REAL) descreve um vetor A de 300 números reais, sem se preocupar com o tamanho ou formato de representação do número de ponto flutuante definida para uma determinada arquitetura. Uma implemetanção MPI para redes heterogêneas garante que os mesmos 300 números reais serão recebidos, mesmo se diferentes máquinas tenham uma representação distintas para ponto flutuante [GRO, 1994].

3.5.5 Tipo de dados MPI

Todas as mensagens MPI são tipadas, sendo que o tipo de conteúdo deve ser especificado tanto no envio quanto no recebimento. O usuário pode usar tanto os tipos de dado pré-definidos pelo padrão ou definir seus próprios tipos de acordo com suas necessidades.

3.4.5.1 Tipo de dados Básico

Os tipos de dados básicos em MPI correspondem aos tipos básicos de C e Fortran. A tabela abaixo descreve os tipos em MPI em relação ao C:

Definição Do MPI	Definição em C
MPI_CHAR	signed char
MPI_INT	signed int
MPI_FLOAT	float
MPI_DOUBLE	Double
MPI_UNSIGNED_SHORT	Unsigned short int
MPI_UNSIGNED	Unsigned int
MPI_UNSIGNED_LONG	Unsigned long int
MPI_LONG_DOUBLE	long double
MPI_LONG	Signed long int
MPI_UNSIGNED_CHAR	Unsigned char
MPI_SHORT	Signed short int
MPI_BYTE	-
MPI_PACKED	-

Tabela 3.2 – Tipos de dados básicos em C

3.4.5.2 Tipos Definidos Pelo Usuário

Os mecanismos de comunicação MPI introduzidos anteriormente permite-nos enviar e receber uma seqüência de elementos idênticos que são contíguos na memória. Muitas vezes é desejável que se envie dados que não sejam homogêneos, como uma estrutura, ou que não seja contígua na memória, como uma parte de um array. Isto permite amortizar o overhead causado pelo envio e recebimento de muitos elementos sob um canal de comunicação. MPI fornece dois mecanismos para realizar isto [SNI, 1996].

- O usuário pode definir tipos de dados derivados, que especificam arranjos (layouts) de dados mais gerais. Tipos de dados definidos por usuários podem ser usados em funções de comunicação MPI, em lugar dos tipo de dados pré-definidos.
- Um processo emissor pode explicitamente compactar dados não-contíguos em um buffer contíguo, e então enviá-lo. Um processo receptor pode explicitamente descompactar dados recebidos em um buffer contíguo e armazená-lo em locais não contíguos.

3.6 MPICH

O MPICH [GRO, 2003] é uma implementação do padrão MPI distribuída gratuitamente na Internet. É um projeto liderado por pesquisadores da Argonne National Laboratory e Mississippi State University com contribuições de pesquisadores da IBM e estudantes voluntários.

A estrutura do MPICH se divide em dois níveis. No primeiro concentra-se todo o código fonte reaproveitável (independente de hardware), que pode ser transportado diretamente e onde as funções do MPI são implementadas. No segundo nível situa-se o código dependente de plataforma e estes dois níveis comunicam-se através de uma camada denominada ADI (Abstract Device Interface). Todas as funções do MPI são escritas utilizando-se rotinas e macros oferecidas pela ADI. É importante ressaltar que a ADI é projetada para permitir que qualquer biblioteca de passagens de mensagens possa ser implementada sobre ela.

Denomina-se dispositivo (device), à organização de software que se encontra no segundo nível do MPICH, e através de uma especificação precisa para a ADI, possibilita-se que os fabricantes de hardware, por exemplo, forneçam implementações eficientes de dispositivos próprios. Pode-se também implementá-los a partir de rotinas de bibliotecas de passagem de mensagens nativas (da própria plataforma de hardware) ou plataformas de passagens de mensagens já existentes.

Em virtude da facilidade de transporte do código do MPICH para novas plataformas de hardware, esta implementação vem se difundido de maneira significativa e rápida, e já executa com eficiência em uma grande gama de máquinas paralelas e redes de computadores.

A figura a seguir apresenta um modelo resumido da estrutura do MPICH (versão 1.0.12) quando instalado sobre uma rede LINUX. O dispositivo utilizado é o ch_p4 (implementação mista baseada nas rotinas Chameleon e p4) [LIN, 1998].

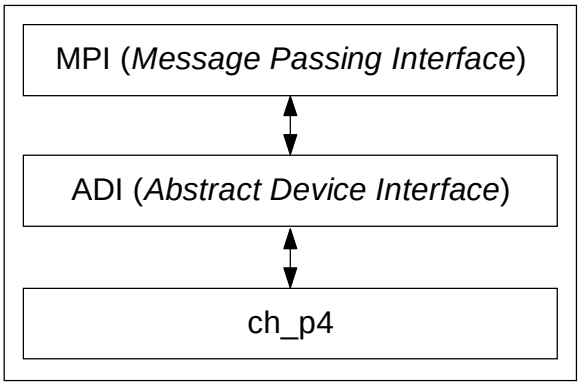

Figura 3.1 – Estrutura do MPI sobre o sistema Linux

Fonte [LIN, 1998]

3.7 Considerações finais

Neste capítulo apresentamos um padrão utilizado em programação paralela utilizando passagem de mensagens denominada MPI (Message Passing Interface). Este padrão surgiu a partir da mobilização de entidades representando empresas e universidades que identificaram a importância de unificar o esforços isolados cada qual com seus bons resultados. Hoje este padrão é uma referência de fato.

MPICH é uma implementação do padrão MPI distribuída livremente pela Internet e é a implementação escolhida para este trabalho. Os motivos para isto são a sua grande popularidade, robustez e baixo custo.

4 Algoritmos Genéticos

4.1 Considerações Iniciais

Neste capítulo introduzimos o conceito de Algoritmos Genéticos (AG), partindo de uma breve história, em seguida descrevendo suas diversas partes, considerando um AG tradicional. Ao final do capitulo, apresentamos os Algoritmos Genéticos Paralelos (AGP), uma variação do modelo tradicional que explora o potencial deste tipo de algoritmo em ambientes paralelos ou distribuídos.

4.2 Algoritmos Genéticos

AG são algoritmos aproximativos baseados nas idéias de Charles Darwin, biólogo e naturalista britânico autor do Livro "A Origem das Espécies" [DAR, 2003]. Nesta obra, o autor apresenta duas teorias: mediante a evolução biológica todas as plantas e animais descendem de formas anteriores mais primitivas. A segunda, afirma que a evolução se deve à seleção natural. A teoria da evolução diz que a vida se desenvolveu ao acaso a partir de matéria inorgânica e logo cresceu gradualmente e continuamente em complexidade e variedade, transferindo em seus genes seus conhecimentos preenchendo a terra com todas as espécies que hoje existem. As forças que impulsionam e evolução são a mutação e a seleção natural. As mutações fornecem novas informações genéticas e a seleção natural permite que aqueles organismos que sobrevivem passem a nova informação genética melhorada a seus descendentes, que, se sobreviverem, passarão a seus filhos e assim sucessivamente [CRU, 2002].

Hoje, os AGs constituem uma parte da Computação Evolutiva, a qual está rapidamente crescendo como uma subdivisão da Inteligência Artificial. Segundo [LOU, 2002] os AGs surgiram da mistura entre sistemas naturais e artificiais e da observação dos sistemas biológicos.

A tabela 4.1 apresenta um resumo das correspondências entre as terminologias empregadas em AGs:

Natural	Algoritmo Genético
Cromossomo	String
Gene	Característica
Alelo	Valor da característica
Lócus	Posição na string
Genótipo	Estrutura ou população
Fenótipo	Conjunto de parâmetros, uma estrutura decodificada

Tabela 4.1 – Correspondência entre os termos natural e artificial

4.3 História

Os AGs foram desenvolvidos na Universidade de Michigan, Estados Unidos, pelo professor e matemático John H. Holland nos anos 70. O primeiro livro sobre o tema surgiu em 1975 com o título "Adaptation in Natural and Artificial Systems". Diversos artigos e dissertações estabeleceram a validade da técnica em otimização de funções e controle de aplicações, tendo se expandido hoje para diversas áreas do conhecimento tais como em aplicações comerciais, ciência e engenharia. A razão por trás deste fato é que este tipo de algoritmo é relativamente simples de ser implementado e poderoso em sua busca por melhora.

Sob a ótica da computação, os AGs modelam o fenômeno natural de herança genética e a teoria da evolução. Os indivíduos de determinada espécie representam possíveis soluções que são melhoradas através de operadores genéticos a cada iteração, análogo ao processo de evolução biológica. Tais operadores são definidos de acordo com o problema em questão.

O objetivo principal de um AG então é o de evoluir a partir de um conjunto de soluções possíveis de um dado problema, denominado de população inicial, tentando produzir novas gerações de soluções que sejam melhores que as anteriores.

4.4 Algoritmo Genético Tradicional

Um AG tradicional funciona da seguinte maneira: uma população de cromossomos (os pais em potencial de uma nova população) se mantém em todo o processo evolutivo. A cada um deles se associa um valor de "fitness" (aptidão) que está associada com o valor da função objetivo a otimizar. Cada cromossomo então está codificando um ponto no espaço de configurações (o espaço de busca) do problema.

Dois cromossomos são selecionados para serem os pais de uma nova solução mediante um mecanismo de crossover (cruzamento). Nos algoritmos originais de Holland, um deles era eleito de acordo com o seu valor de "fitness" (quanto maior este valor maior a probabilidade de ser eleito), enquanto que o outro era eleito aleatoriamente. Este processo se repete tantas vezes quanto necessário. No decorrer deste capítulo serão vistos com maiores detalhes os operadores genéticos. Um esboço de um AG tradicional está descrito na figura 4.1.

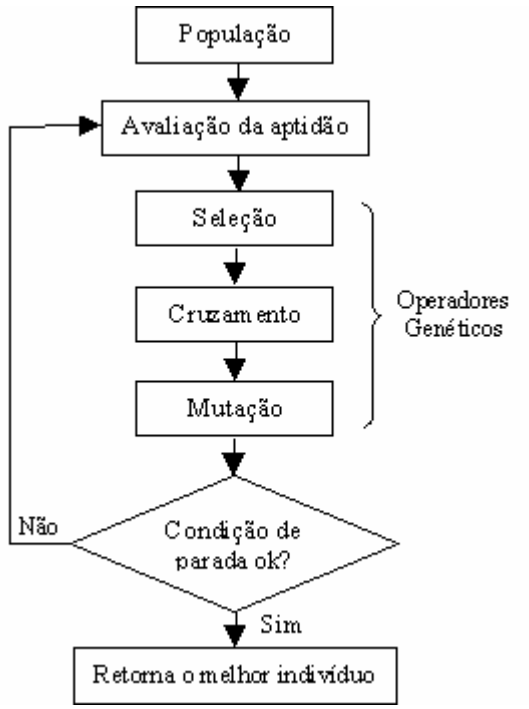

Figura 4.1 – Fluxograma de um AG

Múltiplas variações deste esquema básico foram desenvolvidas desde que surgiram os AGs. Algumas das mudanças mais significativas dizem respeito ao tipo de codificação, que inicialmente eram feitas apenas por strings binárias. Em [LAG, 1995] o autor afirma que para determinados tipos de aplicações, como por exemplo, o problema do Caixeiro Viajante, uma representação por números inteiros seria a mais indicada. Neste caso, cada número dentro da string representaria o custo (a distância) entre duas

cidades. Outra modificação importante foi a inclusão de outras metaheurísticas[3] como *Simulated Annealing*, *Tabu Search* e *Scatter Search*. Desta reunião de tecnologias surgiu o termo "Algoritmos Genéticos Híbridos". Modelos paralelos e distribuídos também foram desenvolvidos [NOW, 1992]. Os AGs paralelos terão uma abordagem maior ainda neste capítulo.

Independente de qual tipo seja o AG, existem cinco componentes que devem ser incluídas em uma implementação [LAG, 1995]:

- uma representação em termos de cromossomo, das configurações de um problema;
- uma maneira de criar as configurações da população inicial;
- uma função de avaliação que permita ordenar os cromossomos de acordo com sua função objetivo;
- operadores genéticos que permitam alterar a composição dos novos cromossomos gerados pelos pais durante a reprodução; e
- valores dos parâmetros que o AG usa (tamanho da população, probabilidades associadas com a aplicação dos operadores genéticos, etc.).

Descreveremos em seguida os detalhes quanto à representação e operadores genéticos, levando em consideração os algoritmos genéticos tradicionais.

4.5 Avaliação de Aptidão

Avaliar a aptidão de um indivíduo (cromossomo) significa fornecer uma medida qualitativa, indicando se este indivíduo está ou não próximo da solução desejada. Quanto mais próximo do valor objetivo (máximo ou mínimo), mais adaptado ele está e maiores as suas chances de sobrevivência, ou seja, de passar para as próximas gerações. Este componente tem uma atuação chave dentro de qualquer AG. O ideal seria que esta função diferenciasse corretamente as boas das más soluções, evitando que ótimas soluções sejam descartadas e que as más soluções sejam trabalhadas.

[3] Metaheurísticas são procedimentos destinados a encontrar uma boa solução, eventualmente a ótima, consistindo na aplicação, em cada passo, de uma heurística subordinada, a qual tem que ser modelada

FALQUETO (2002) nos chama a tenção para um erro de interpretação comum envolvendo os termos "fu nção de adaptabilidade" e "função de aptidão". O autor esclarece que "Uma função de adaptabilidade ou de adaptação mede a flexibilidade de um organismo a se manter viável em diferentes ambientes. A função de aptidão, mede o quão bem equipado está um organismo para viver e evoluir em seu habitat."

4.6 Representação Cromossômica

A primeira providência a ser tomada quando se pretende resolver determinado problema através de AG é definir de que forma o problema em questão vai ser representado (codificado). A importância vem do fato de que os AGs operam em um espaço de busca codificado. O espaço de busca consiste em um conjunto de pontos que representam as soluções viáveis de um problema. A codificação é feita como uma string de tamanho fixo sobre um alfabeto finito. Sobre string de tamanho fixo entende-se que esse não varia seu tamanho durante a execução da simulação.

Nos trabalhos iniciais de Holland, os cromossomos eram representados por strings binárias de dados, feitas a partir de um alfabeto composto apenas pelos valores 0 e 1, porém este se mostrou insuficiente com o passar dos anos, para certos tipos de problemas. Mesmo assim esta representação tem sido largamente utilizado, sendo, portanto, bastante analisada e depurada. Além disso, uma representação binária é uma codificação natural para um grande número de problemas como, por exemplo, o Problema da Mochila. Neste problema a string (10011) pode representar a configuração (x_1=1, x_2=0, x_3=0, x_4=1, x_5=1) enquanto que considerando o Problema do Caixeiro Viajante, esta codificação não é tão natural. Neste caso, a representação por números inteiros é a mais adequada, assim a solução *(123451)* poderia estar associada a uma configuração, em que o número associado a cada gene do cromossomo representa uma cidade [LAG, 1995].

MICHALEWICZ (1996) argumenta que a representação binária apresenta desempenho inferior quando aplicada a problemas numéricos com alta dimensionalidade e onde alta precisão é requerida. Supondo um exemplo, onde temos um problema com 100 variáveis, com domínio no intervalo [-500, 500] e que

---

para cada problema específico. Fonte [SOU, 2003].

precisamos de 6 dígitos de precisão após a casa decimal. Neste caso, precisaríamos de um cromossomo de comprimento 3000, e teríamos um espaço de busca de dimensão aproximadamente 10^{1000}. Neste tipo de problema, o AG clássico apresenta desempenho pobre. O mesmo autor apresenta também simulações computacionais comparando o desempenho de AGs com codificação binária e com ponto flutuante, aplicados a um problema de controle. Os resultados apresentados mostram uma clara superioridade da codificação em ponto flutuante.

Independentemente do tipo de representação selecionada, devemos sempre verificar se a representação está corretamente associada com as soluções do problema analisado. Ou seja, que toda solução tenha um cromossomo associado e reciprocamente que todo cromossomo gerado pelo AG esteja associado a uma solução válida do problema analisado.

Para [CAS, 2002] fica claro, portanto, que a codificação é uma das etapas mais críticas na definição de um AG. A definição inadequada da codificação pode levar a problemas de convergência prematura do AG. A estrutura de um cromossomo deve representar uma solução como um todo e deve ser o mais simples possível.

4.6.1 Esquemas

Esquemas são um conceito importante nos estudos dos AGs. Enquanto uma codificação que representa um cromossomo é uma seqüência definida sobre um alfabeto A, que pode ser formado pelos elementos A={0,1}, por exemplo, um esquema é uma seqüência de mesmo comprimento mas definida sobre um alfabeto com um caracter adicional, como um asterisco (*). Tal elemento atua como um coringa que, quando presente numa posição de um esquema, significa que nessa posição qualquer símbolo do alfabeto A pode ser admitido. Por exemplo, o esquema E=(01*0*) pode ser usado para representar os seguintes elementos, considerando o alfabeto A={0,1,*}:

01*0*	01101
	01100
	01001
	01000

Tabela 4.2 – Possíveis valores de string para um esquema simples

Os esquemas são considerados blocos construtivos e foram desenvolvidos por Holland para explicar como os AGs funcionam. Segundo [FUR, 1998], na teoria tradicional, um AG age descobrindo, enfatizando e recombinando bons blocos de construção nas soluções. A idéia é que boas soluções são produzidas por segmentos de bons blocos. Estes blocos são representados pelos esquemas.

4.7 Operadores Genéticos

Os operadores genéticos são os responsáveis pelas modificações sofridas pelos indivíduos de uma população. O objetivo básico de um operador genético é produzir novos cromossomos que possuam propriedades genéticas superiores às encontradas nos pais. Há três operações básicas presentes em todos os AGs [COS, 2000]: Seleção, Crossover e Mutação. Segundo [LOU, 2002] alguns algoritmos não utilizam a operação crossover e são conhecidos como algoritmos evolutivos e não AG.

4.7.1 Seleção

O operador de seleção tem a função de aprimorar os indivíduos de uma população. Ele é responsável por determinar quais indivíduos farão parte da nova geração, através da ação de outros operadores genéticos. Para [COS, 2000], o operador de seleção determina o repasse do código das melhores soluções de geração em geração, contribuindo diretamente para o sucesso do algoritmo. A seleção baseia-se em valores dados pela função objetivo, denominada função de fitness ou de avaliação, que mede a adequação do indivíduo ao meio ambiente. Quanto mais adaptado um indivíduo, maior a chance de contribuir para as gerações futuras. Algumas técnicas que implementam a seleção serão descritas a seguir.

4.7.1.1 Roleta Giradora (Roulette Wheel)

O Método da Roleta, (Roulette Wheel), simula o giro de uma roleta e a escolha aleatória de uma de suas divisões, onde cada divisão corresponde ao valor de aptidão, em porcentagem, de determinado indivíduo. Quanto maior este índice, mais chances um indivíduo tem de ser selecionado, conforme pode ser visualizado na figura 4.2.

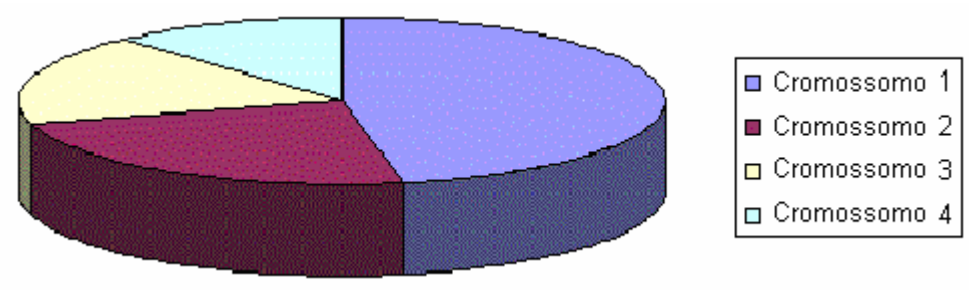

Figura 4.2 – Distribuição de aptidões de diversos cromossomos

Imaginando o disco acima girando, seria fácil de perceber que o indivíduo 1 tem mais chances de ser selecionado em relação aos demais.

4.7.1.2 Seleção por Classificação

O método descrito anteriormente, apresentará problemas se os valores de adaptação forem muito diferentes. Por exemplo, se o valor de adaptação do melhor cromossomo for 90% de toda a roleta, os demais cromossomos terão poucas chances de serem selecionados.

O método de Seleção por Classificação primeiramente classifica a população, depois cada cromossomo recebe um valor de acordo com esta classificação. O pior terá valor 1, o segundo pior terá valor 2 e assim sucessivamente. O melhor terá valor N igual ao número de indivíduos da população. Nas próximas figuras temos a situação após ocorrer a classificação dos cromossomos [OBI, 1998]:

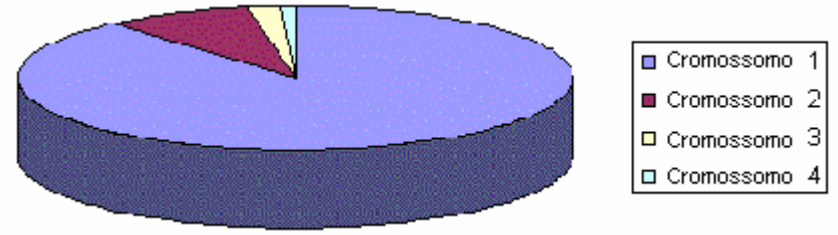

Figura 4.3 – Situação antes da classificação

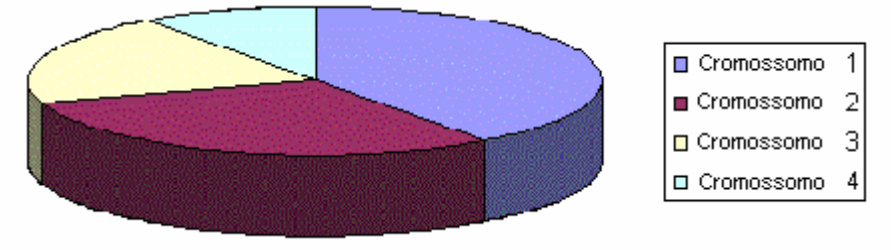

Figura 4.4 – Situação após a classificação

Fonte [OBI, 1998]

4.7.1.3 Seleção por Estado Fixo (Steady-State)

Neste método, a estratégia é manter a população original com a exceção de alguns indivíduos menos adaptados. O AG seleciona os cromossomos com altos valores de adaptação para gerar os filhos. Os cromossomos menos aptos, com baixos valores de adaptação, são retirados da população e os cromossomos filhos gerados são colocados em seus lugares. O restante da população permanece inalterada na composição da nova geração.

4.7.1.4 Seleção por Elitismo ou Torneio

Ao se aplicar os operadores de crossover e mutação para gerar novas populações, há uma grande chance de perder o melhor cromossomo. No método de seleção por Elitismo, primeiramente copia-se o melhor cromossomo (ou os melhores cromossomos) para a nova população. O restante é realizado do modo normal. O Elitismo pode aumentar rapidamente o desempenho do AG porque evita a perda da

melhor solução. Mas também pode provocar rápido empobrecimento das características genéticas da população.

4.7.2 Crossover

O operador Crossover simula o processo de reprodução sexuada: dois seres vivos (os pais) trocam parte de seu material genético para criar um novo organismo. A idéia por trás da operação de crossover é a troca de informações entre diferentes soluções candidatas. Originalmente esta operação era do tipo "Crossover de um ponto" (one-point crossover) [LAG, 1995]. Neste caso, o operador elege aleatoriamente um ponto de divisão de tal maneira que a informação genética das partes divididas são trocados entre os pais para formar um novo indivíduo. O exemplo abaixo ilustra melhor esta operação.

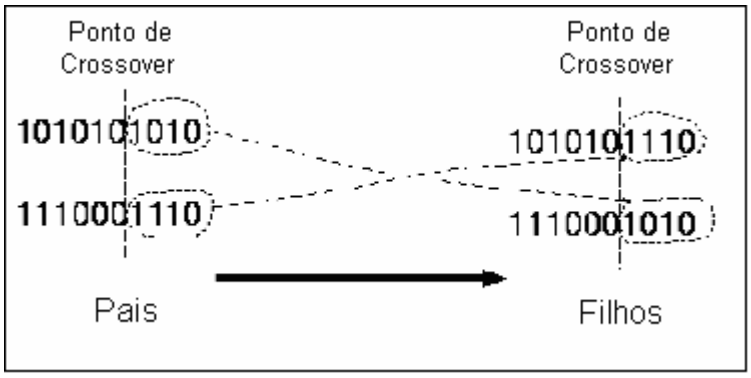

Figura 4.5 – Operação de crossover sobre um ponto

Fonte: [POL, 1996]

A operação de crossover possui um forte impacto nos blocos de construção dos AGs, pois a cada ocorrência deste, há a possibilidade de um esquema ser destruindo. Esta é uma das principais razões pelas quais o crossover não deve ser executado a todo instante.

Outras variantes da operação de crossover estão disponíveis em [OBI, 1998] [LOU, 2002].

4.7.3 Mutação

O operador mutação altera um ou mais genes de um cromossomo de acordo com uma probabilidade denominada Taxa de Mutação. Em cromossomos binários isto consiste em inverter um bit por outro. A idéia do operador mutação é introduzir informação nova em uma população, porém com a preocupação de não prejudicar o progresso já adquirido no processo de busca. Por esta razão é dada uma probabilidade pequena de atuação deste operador.

Os operadores de crossover e mutação descritos até aqui foram enfocados com base em uma codificação binária, entretanto de acordo com [ZUB, 2002], existem alternativas específicas para uma codificação baseada em ponto flutuante. No caso do operador crossover, podemos citar crossover aritmético e para mutação os tipos uniforme e gaussiana [MIC, 1996].

4.8 Algoritmos Genéticos Paralelos

Um AG seqüencial, como os considerados até aqui tem se mostrado eficiente em muitas aplicações. Entretanto algumas considerações quanto ao seu desempenho serviram de incentivo para o desenvolvimento de Algoritmos Genéticos Paralelos (AGP). Nowostawski e Poli [NOW, 1996] relacionam algumas características:

- para certos tipos de problemas, a população precisa ser muito grande e a memória requerida para armazenar cada indivíduo pode ser considerável. Neste caso seria inviável executar uma aplicação eficientemente usando uma única máquina;
- a avaliação de fitness geralmente consome uma considerável quantia de tempo; e
- AGs seqüenciais podem limitar-se a regiões sub-ótimas do espaço de busca incapaz de encontrar soluções de boa qualidade. Algoritmos Genéticos Paralelos podem pesquisar em paralelo diferentes sub-espaços do espaço de busca, tornando menos provável o encontro de soluções com baixa qualidade.

Para Mühlenbein [MUH, 2002], os Algoritmos Genéticos Paralelos usam duas grandes modificações comparadas aos AGs tradicionais. Primeiro, a seleção para

reprodução é feita de forma distribuída. A seleção de um parceiro é feita por cada indivíduo independentemente de sua vizinhança. Segundo, cada indivíduo pode melhorar seu valor de aptidão (fitness) durante seu tempo de vida através de por exemplo uma busca local como hill-climbing[1].

Os Algoritmos Genéticos Paralelos não são somente uma extensão do modelo tradicional, visando o aumento de performance. Segundo [ALB, 2001], eles representam uma nova classe de algoritmos que fazem um trabalho de pesquisa em diferentes regiões do espaço de busca.

Existem diferentes métodos de se implementar um Algoritmo Genético Paralelo, porém não há uma taxonomia eleita como referência que as descreva.

Uma delas, descrita em [NOW, 1992], divide as diferentes técnicas de paralelização em oito classes:

1. Paralelização Master-Slave (avaliação de fitness distribuída)
    (a) Síncrona
    (b) Asíncrona
2. Sub-populações estáticas (com migração)
3. Sub-populações estáticas (sem migração)
4. Algoritmos Genéticos Paralelos massivos
5. Diversas populações com migração (Dynamic demes)
6. AG de estado seguros paralelos
7. Algoritmos Genéticos Paralelos desordenados (Parallel messy genetic algorithms)
8. Métodos híbridos (ex. Sub-populações estáticas com migração, com avaliação de fitness distribuída dentro de cada sub-população).

Para este estudo, utilizaremos o termos descritos por PAZ [PAZ, 1995], que classifica os AG baseados no modelo de paralelização utilizados:

- paralelização global;
- paralelização com granularidade[4] grossa; e
- paralelização com granularidade fina.

[1] Técnica com origem na matemática clássica desenvolvida nos séculos 18 e 19. Esta classe de métodos de busca encontra um ponto ótimo seguindo o gradiente local da função (eles são, as vezes, chamados de métodos dos gradientes). Suas pesquisas são determinísticas e geram resultados sucessivos baseados somente nos resultados anteriores. Fonte [LOU, 2002].

4.8.1 Paralelização Global

Neste modelo há uma única população onde a avaliação de cada indivíduo é dividida entre vários processadores. Esta classe de algoritmos é chamada "global" porque a seleção e o acasalamento são realizados considerando-se todos os indivíduos de uma população. Algoritmos Genéticos Globais são freqüentemente implementados como programas Mestre-Escravo (denominado Master-Slave na literatura inglesa especializada) e requerem uma comunicação constante entre processadores (Figura 4.6).

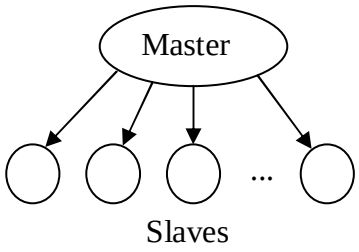

Figura 4.6 – Esquema de um AGP Global

PAZ [PAZ, 1997] reforça que os Algoritmos Genéticos Paralelos Globais exploram o espaço de busca da mesma maneira que sua versão serial e são relativamente fáceis de serem implementados.

A avaliação de indivíduos é a operação mais comum a ser paralelizada no modelo Global. Isto porque o valor de fitness é particular a cada indivíduo em relação ao restante da população e não há a necessidade de comunicação durante esta etapa. A avaliação de indivíduos é paralelizada associando-se uma fração da população a cada um dos processadores usados. A comunicação ocorre somente a medida que os processadores recebem um subconjunto de indivíduos para avaliar e quando os processadores retornam os valores de fitness.

Denomina-se Algoritmo Genético Global Síncrono quando o algoritmo pára e espera o recebimento dos valores de fitness para toda a população antes de prosseguir para a próxima geração. Os algoritmos síncronos tem as mesmas características que um algoritmo seqüencial tradicional, tendo a possibilidade no aumento da performance como única diferença.

---

[4] O termo "Granularidade" em computação paralela define o nível de paralelismo, e está relacionado com o tamanho dos processos executados pelos processadores.

O método de paralelização global não requer uma arquitetura de hardware em particular, e ela pode ser implementada tanto em computadores com memória compartilhada como em memória distribuída. Considerando-se uma arquitetura de memória distribuída, a população é armazenada em um processador. Este processador, denominado Mestre é responsável por enviar os indivíduos aos outros processadores ("Escravos") para avaliação, coletar os resultados, e aplicar os operadores genéticos produzindo a próxima geração.

4.8.2 Paralelização com Granularidade Grossa

Este modelo consiste em um conjunto de subpopulações atribuídas a processadores distintos que evoluem independentemente umas das outras e periodicamente trocam indivíduos entre si. Este modelo também é conhecido como Modelo em Ilha ou Modelo Distribuído uma vez que eles são geralmente implementados em computadores MIMD de memória distribuída. É usual também cada subpopulação ser chamada de Deme[5].

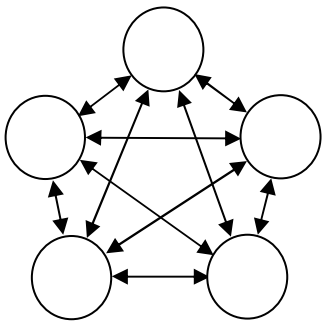

Figura 4.7 – AG Distribuído

As características importantes desta classe de algoritmos são o uso de subpopulações relativamente grandes e a introdução do operador de migração, responsável por enviar certos indivíduos de uma subpopulação à outra. O propósito da operação de migração é dar a possibilidade de renovação a uma subpopulação guiando a busca genética global a regiões promissoras do espaço de busca. Desta forma os melhores resultados correntes são compartilhandos entre as subpopulações. São três os

[5] Termo usado na Grécia antiga para delimitar uma subdivisão territorial. Semelhante a vila ou distrito.

fatores que influenciam no processo de comunicação gerado pelo operador de migração: topologia, intervalo de migração e taxa de migração.

A topologia de conexão entre populações é um dos fatores que influenciam no processo de comunicação gerado pelo operador de migração e é determinada pela arquitetura de hardware. Este fator tem relação direta com ao desempenho do AG ao controlar a velocidade de propagação das boas soluções entre as sub populações. Quanto maior a conectividade existente, maior a chance de indivíduos bem adaptados escoarem influenciando assim as subpopulações. Topologias comuns incluem anéis (rings), grade (grid), Hipercubo entre outros.

Um outro fator que têm influência no processo de comunicação é o intervalo de migração. Este valor descreve a freqüência com que a migração acontece, e deve ser definido em função do tempo necessário para que cada subpopulação encontre novas boas soluções. Migrações em excesso constituem um desperdício de recursos e migrações tardias podem deixar de trazer benefícios ao algoritmo.

A taxa de migração controla o número de indivíduos que irão migrar em cada intervalo. Baixos níveis de comunicação podem prejudicar a capacidade de exploração do algoritmo enquanto que o excesso de comunicação pode reduzir o processo de refinamento das soluções.

4.8.3 Paralelização com Granularidade Fina

Neste caso a população é dividida e mantida por diversos processadores que se conectam por uma topologia em Grid (grade) cada qual processando sua subpopulação (Figura 4.8). Este modelo também é conhecido como Modelo de Vizinhança (Neighbourhood Model) ou Modelo Celular (Cellular Model). A principal diferença em relação ao modelo de granularidade grossa é o tamanho da subpopulação. No modelo de granularidade fina uma subpopulação pode conter um único indivíduo.

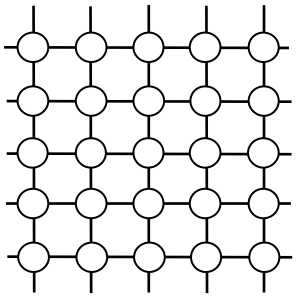

Figura 4.8 – AG de Granularidade Fina.

Neste modelo, o processo de reprodução é aplicado somente entre indivíduos vizinhos (de acordo com a topologia definida).

É possível combinar os diversos modelos de Algoritmos Genéticos Paralelos descritos anteriormente dando origem ao que se denomina Modelos Híbridos [ALB, 1995]. PAZ [PAZ,1997] descreve um modelo híbrido combinando um Algoritmos Genético Global com um Algoritmos de Granularidade Grossa (Fig. 4.4).

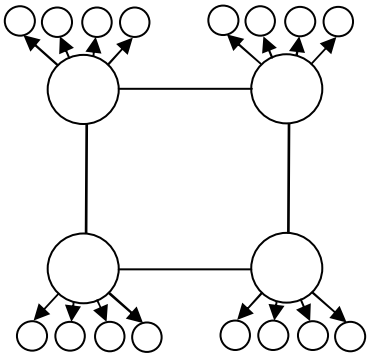

Figura 4.9 – Um modelo Híbrido.

Nesta configuração os dois métodos formam uma hierarquia sendo que o algoritmo de granularidade grossa assume o nível mais alto.

PAZ [PAZ, 1997], enfatiza ainda que enquanto o método de paralelização global não afeta o comportamento do algoritmo, os outros métodos introduzem mudanças fundamentais na forma como um AG trabalha. Por exemplo, no modelo global o mecanismo de seleção considera a população como um todo, mas em outros métodos a seleção é feita sobre um subconjunto da população (os Demes). Da mesma forma no

modelo global pode haver relacionamento entre quaisquer dos membros da população, mas nos métodos que dividem a população este relacionamento está restrito aos membros desta subpopulação.

4.9 Considerações finais

Este capítulo descreveu de forma sucinta os AG, uma importante técnica de otimização com raízes na computação evolutiva e que simula o desenvolvimento biológico ao longo do tempo de uma população de indivíduos.

Este tipo de algoritmo codifica possíveis soluções de um determinado problema em strings de dados (indivíduos). Um conjunto destas strings forma uma população que permanece durante todo o processo de simulação e que representa o espaço de busca que se pretende pesquisar.

A cada iteração é medida o quanto cada indivíduo é capaz de solucionar o problema proposto e, mediante este valor, algumas soluções em potencial são selecionadas, através de operações específicas de seleção e produzem uma nova população. Esta nova geração tende a ser mais aprimorada em relação às anteriores pelo fato de serem produzidas a partir de boas soluções, colhidas na etapa de seleção descrita anteriormente.

Os Algoritmos Genéticos Paralelos são uma variante do modelo tradicional e têm características que os tornam atraentes, pois além de aumentar a performance, trabalham em espaços de busca distintos, resultando em soluções mais precisas.

5 Implementação e resultados

5.1 Considerações Iniciais

Este capítulo apresenta o protótipo desenvolvido, fornecendo os detalhes de sua implementação, demonstra de que forma o AG é paralelizado juntamente com sua representação interna e descreve a interface com o usuário. Ao final, é dado um exemplo prático em que uma aplicação otimiza uma função até achar seu ponto máximo.

Para a execução dos testes, foi utilizado um pequeno cluster no estilo Beowulf, composto por 3 máquinas. Não serão dados os detalhes da montagem deste cluster, pois fogem ao escopo deste trabalho. Uma abordagem sobre este tema pode ser vista em [MAZ, 1996].

5.2 Modelo do Algoritmo Genético Paralelo

O método utilizado para implementar o AGP consiste basicamente em ter uma cópia de um AG seqüencial em cada nodo participante do cluster, semelhante ao modelo em Ilha descrito em [PAZ, 1998]. Nesta forma de implementação, cada processador armazena sua própria população de indivíduos, como se fossem ilhas isoladas, que serão então evoluídas de forma independente umas das outras. A figura 5.1 ilustra como é feito o processamento, considerando-se três processadores (`P0`, `P1` e `P2`). A interface com o usuário define alguns procedimentos, que serão descritos mais adiante e é composta por um conjunto de funções definidas no arquivo sga.h. A partir daí, cada processo gera sua população inicial e executa um AG tradicional com base nos valores definidos pelo usuário. A linha pontilhada indica que o loop principal dos N processos são feitos de forma paralela.

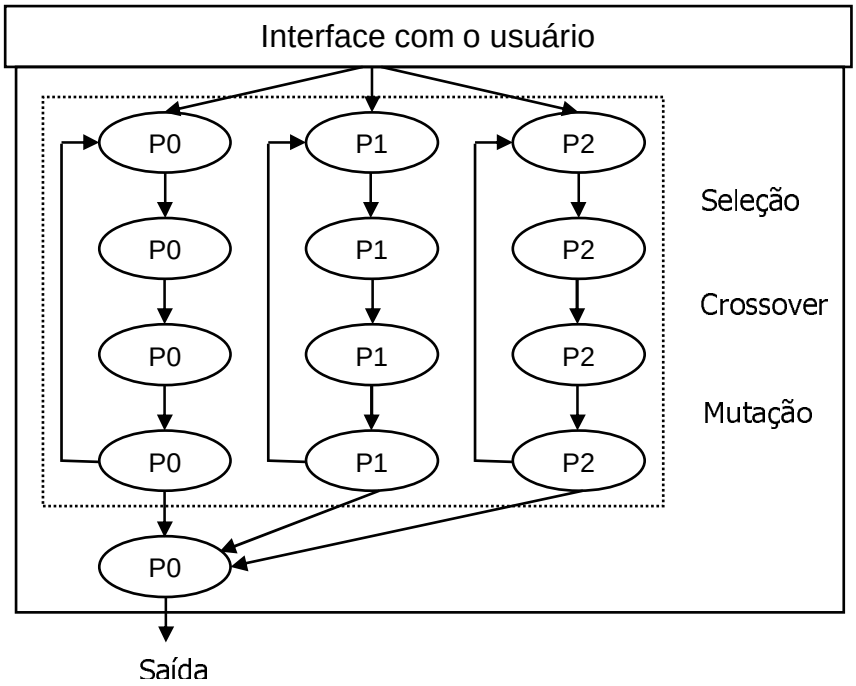

Figura 5.1 – Funcionamento interno da biblioteca

Como critério de parada é considerado apenas um determinado número de gerações, que pode ser ou não fornecida pelo usuário. Ao final do processamento, haverá em cada nodo participante do cluster uma possível solução referente à sua análise seqüencial. Cabe a um dos processos, o de número 0, escolher a melhor solução entre estas disponíveis.

5.3 Implementação da biblioteca

O protótipo foi desenvolvido em ANSI C sobre o Sistema Operacional Conectiva Linux. Para a exploração do paralelismo foi utilizado o pacote MPICH [GRO, 2003], uma implementação de domínio público do padrão MPI [PAC, 1998].

Como ferramentas foram utilizadas:

- Motor – uma IDE (Integrated Development Environment) baseada em console para o Linux [KLY, 2003]. Esta ferramenta dispõe de um ‘template” para desenvolvimento de bibliotecas estáticas e a facilidade de gerar pacotes de distribuição de software em vários formatos (tar.gz, rpm). Estes foram os principais motivos de sua escolha;

- TOMPI – designado para rodar programas MPI em computadores simples [DEM, 2003]. Com sua utilização, é possível executar programas paralelos em um computador normal, simulando a existência de um cluster;
- xxgdb – ferramenta de depuração para o ambiente gráfico do Linux [HOR, 2003].

A biblioteca é do tipo estática por imposição do próprio MPICH uma vez que, segundo GROPP et al. (2003), o suporte a bibliotecas compartilhadas até o presente momento é um projeto futuro. Uma discussão mais detalhada sobre bibliotecas estáticas e compartilhadas pode ser encontrada em [KER, 2003].

Como característica desta biblioteca podemos citar:

- codificação binária;
- crossover de um ponto;
- seleção pelo método da roleta;
- chamada a partir de programas escritos em C; e
- total de 16 funções visíveis ao usuário.

5.3.1 Representação Interna

Como sabemos, em AG, uma população é composta por um certo número de indivíduos que representam (codificam) possíveis soluções de um problema. É sobre esta população que os operadores genéticos atuam. As informações que definem um membro de uma população estão declaradas em uma estrutura de dados chamada Individuo. Esta estrutura está definida no arquivo de cabeçalho `sga.h` que está parcialmente representada na listagem seguinte.

sga.h

```
struct individuo
{
    unsigned *cromossomo;
    double   fitness;
    int      ptcros;
    int      pais[2];
};

struct melhoratual
{
    unsigned *cromossomo;
    double   fitness;
    int      geracao;
};

typedef struct Parametros Parametros;

struct Parametros
{
    int   tampop;
    int   tamcromo;
    int   maxgeracoes;
    float probcross;
    float probmut;
    float semente;
};
```

Listagem 5.1 – Parte do arquivo `sga.h`

O ponteiro `cromossomo` armazena a string binária que representa um membro da população. A variável `fitness` guarda a aptidão do indivíduo retornado pela função objetivo a cada rodada. `ptcross` e `pais[2]` armazenam o ponto de crossover e os pais para a nova geração respectivamente. Este arquivo define, ainda, outras duas importantes estruturas, que também estão descritos acima. A primeira se chama `melhoratual` e sua função é armazenar os dados do melhor indivíduo em cada geração. É sobre os dados definidos nesta estrutura que, ao final do processamento, o algoritmo paralelo baseia-se para colher a melhor solução.

A outra estrutura, denominada `Parametros`, define um tipo de dado que armazena os parâmetros utilizados pelo AG a saber:

- `tamanho` da população;
- `tamanho` do cromossomo;
- `número` máximo de gerações;

- `probabilidade` de crossover;
- `probabilidade` de mutação; e
- `semente` número da semente para o Gerador de Números Aleatórios.

Inicialmente estes parâmetros terão seus valores ajustados por valores propostos por De Jong, em seus estudos de otimização de funções, utilizando AG. Estes estudos estão expostos em [DEJ, 1975] e sugerem que, para uma boa performance, um AG requer a escolha de uma alta probabilidade de crossover, uma baixa probabilidade de mutação (inversamente proporcional ao tamanho da população) e um tamanho da população moderada. Sendo assim, fica definido os seguintes valores:

- tamanho da população: 30;
- tamanho do cromossomo: 8;
- número máximo de gerações: 100;
- probabilidade de crossover: 0.6;
- probabilidade de mutação: 0.0333; e
- semente para o Gerador de Números Aleatórios: 0.123456.

Estes valores, porém, nem sempre são as melhores opções quando se aplica AG em outras áreas de pesquisa que não otimização de funções.

Eventualmente o usuário terá a possibilidade de fazer alterações destes dados conforme suas necessidades. Esta opção ficará mais clara logo adiante, ao final do capítulo, quando do exemplo de aplicação.

Como foi mencionado, uma versão serial de um AG é executado em cada participante do cluster. Algumas modificações foram feitas para adequar a utilização sobre o ambiente paralelo. Uma delas foi quanto à geração da população inicial. Em [DAV, 2002], o autor alerta que por ser este procedimento randômico e baseado em um parâmetro fornecido pelo usuário (o valor da semente aleatória) que é visível a todos os processos, corre-se o risco no momento de gerar a população inicial de ter os mesmos valores em todos os nodos. Para eliminar este problema, foi associado à variável que armazena o valor da semente aleatória o número de cada processo. Desta forma, cada processador terá um valor diferente para a inicialização de seus indivíduos, garantindo, assim, uma diversidade da população.

5.3.2 Representação Binária

A motivação para o uso de codificação binária vem da teoria dos esquemas. COSTA e ZUBEN (2002) argumentam que seria benéfico para o desempenho do algoritmo maximizar o paralelismo implícito inerente ao AG e comprova que um alfabeto binário maximiza o paralelismo implícito.

A representação de números inteiros por strings binárias é a mais natural. Para a transformação de números reais é preciso considerar limites mínimo e máximo. O mapeamento de uma string binária S em um número real x é feito então em dois passos:

- Converter a string S da base 2 para a base 10:

$$b_{10} = \sum_{i=0}^{l} b_i \cdot 2^i \qquad (1)$$

Onde:

l = Tamanho do cromossomo

b_i = um bit da string S na posição i

- determinar o número real correspondente a x:

$$x = \min + (\max - \min)\ \frac{b_{10}}{2^l - 1} \qquad (2)$$

Onde:

min – limite inferior;

max – limite superior;

b – valor convertido em inteiro representado pela string S;

l – tamanho do cromossomo em bits.

Por exemplo a string (00011) no intervalo de –2 a 2,5 representa –1,565 pois segundo a equação (2):

$$x = -2 + (2{,}5 + 2)\frac{3}{2^5 - 1}$$

A biblioteca dispõe de duas funções, `BinpInteiro` e `BinpReal`, que manipulam a conversão de strings binárias em números inteiros e reais respectivamente. O protótipo de ambas é dado abaixo:

```
BinpInteiro(cromossomo, min, max);
BinpReal(cromossomo, min, max);
```

Onde:

`cromossomo` - o membro da população a ser convertido;

`min` - Valor inteiro representando o limite inferior;

`max` - Valor inteiro representado o limite superir;

Por exemplo, considerando uma variável de nome crom com o valor do exemplo anterior (00011), faríamos a conversão da seguinte forma:

```
x = BinpInteiro(crom, -2.5, 2);
```

5.3.3 Interface com o usuário

A interface define a maneira como o usuário deve proceder para implementar uma aplicação. Basicamente ela é composta por um conjunto de funções que o usuário invoca em uma determinada ordem que vão definir os parâmetros do AG como tamanho da população, número de iterações (gerações) etc, sendo que algumas destas funções são obrigatórias e outras opcionais. O principal objetivo deste módulo é fornecer ao usuário uma interface limpa e fácil de usar, ocultando os detalhes da implementação.

Um parâmetro importante que não é definido na interface com o usuário é o número de processos que farão parte do processamento. Este valor é passado através da linha de comando do Linux como um parâmetro do comando `mpirun` que é parte

integrante do pacote MPICH e responsável pela inicialização do ambiente paralelo. O que este comando faz é distribuir uma cópia do programa a cada processador que compõe o cluster. Sua sintaxe (resumida) é:

mpirun –np <n> <programa>

Onde:

n - número de processos;

programa - nome do programa executável.

Portanto para poder utilizar os recursos da biblioteca e implementar sua aplicação o usuário precisa seguir o modelo descrito abaixo que nada mais é do que a definição da interface com o usuário:

1. importar o arquivo de cabeçalho `agmpi.h;`
2. implementar uma função de avaliação;
3. declarar uma variável do tipo `Parametros`;
4. iniciar o ambiente paralelo;
5. executar o AG; e
6. encerrar o ambiente paralelo.

O arquivo de cabeçalho `agmpi.h` define todas as estruturas de dados, variáveis globais e os protótipos de funções a que o programador tem acesso.

A função de avaliação deve ser fornecida pelo usuário e também segue um modelo pré definido. Esta função tem a forma descrita no protótipo abaixo:

```
double nome_funcao(unsigned *cromossomo);
```

Desta forma, ela deve sempre receber como parâmetro uma string, representando um membro da população. Esta função retorna um tipo `double` e seu parâmetro é um ponteiro do tipo `unsigned`.

A variável do tipo `Parametros`, descrita anteriormente neste capítulo, serve como argumento para algumas funções. É por meio dela que passamos valores ao AG.

Em seguida, é preciso iniciar o ambiente paralelo, o que significa adicionar à aplicação a instrução `MPI_Init()`, uma função da biblioteca MPICH. É importante lembrar que neste ponto "iniciar o ambiente paralelo" tem um contexto diferente do descrito acima, quando da descrição do comando `mpirun`. Neste caso, a função `MPI_Init()` sincroniza todos os processos no início de uma aplicação paralela MPI.

O próximo passo é a execução do AG propriamente dito. Para tanto é preciso somente três funções, descritas a seguir por seus protótipos:

```
Inicializa(&ambiente);
Modifica(&ambiente);
Calcula(&ambiente,f);
```

Onde:

`ambiente` - uma variável do tipo `Parametros`;

`f` – o nome da função de avaliação definida pelo usuário.

Juntas, elas encapsulam o implementação do AG e devem ser chamadas nesta seqüência. Ao usuário, é opcional setar parâmetros como tamanho de cromossomo, tamanho da população, probabilidade de crossover etc, pois as funções `Inicializa` e `Modifica` assumem os valores definidos pela estrutura `Parametros` através de seus argumentos. A biblioteca fornece outras funções que se prestam a modificar estes valores e serão vistas mais adiante, no exemplo de aplicação.

A função `Calcula` detém o loop principal do programa, oculta os detalhes do paralelismo e tem um parâmetro a mais que é o nome da função de avaliação fornecida pelo usuário. No processo de número zero, esta função tem ainda outra responsabilidade antes do término do programa que é colher de cada participante do processamento a melhor solução encontrada por eles. Para tanto, a função `Calcula`, faz uso da rotina `MPI_Reduce`. Esta rotina faz com que todos os processos executem uma operação, em que o resultado parcial de cada processo é combinado e retorna para um processo específico. Em nosso caso, o "resultado parcial" citado refere -se ao melhor indivíduo (o

de maior fitness) da população local de cada processo e o “processo específico” é o processo de número zero que então compara estes resultados e seleciona o de maior valor. É este valor que será apresentado ao usuário. O padrão MPI define isto como “operação de redução” [SNI, 1996].

Por fim, o usuário deve encerrar o ambiente paralelo com uma chamada MPI_Finalize(), outra função definida pelo padrão MPI que tem o papel de sincronizar todos os processos antes do encerramento de qualquer programa.

5.4 Exemplo de Aplicação

Na seção anterior, apresentamos de que forma o programador tem acesso aos recursos da biblioteca, dando detalhes da interface com o usuário. A seguir, estes conceitos serão usados para implementar um exemplo completo.

O exemplo que demonstra o uso da biblioteca paralela consiste em achar o máximo da equação (3)

$$f(x) = 1 + x\,\mathrm{sen}(10\pi x) \qquad (3)$$

restrita ao intervalo $-1 \le x \le 2$. Tal função encontra-se graficamente exposta na figura 5.2.

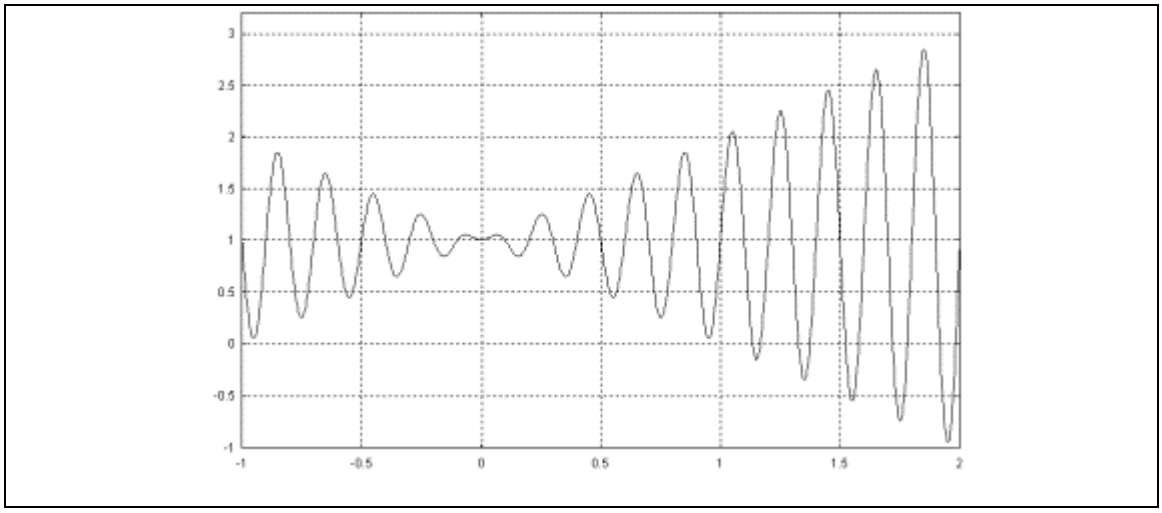

Figura 5.2 – Gráfico da equação (3)

O ponto máximo da função ocorre quando

$$f(x) = f(1.85047) = 2.850274 \qquad (4)$$

São características deste problema :

- função multimodal com vários pontos de máximo;
- é um problema de otimização global (encontrar o máximo global); e
- não pode ser resolvido pela maioria dos métodos de otimização convencional.

Como visto, para que possamos utilizar a biblioteca de Algoritmos Genéticos Paralelos, devemos primeiramente implementar a função de avaliação. Uma possível implementação é dada abaixo:

```
double avalia(unsigned *crom)
{
 double fitness=0.0,
        x=0.0,
        PI=3.14159265358979323846264 3;

 x=BinpReal(crom, -1, 2);
 fitness = x*sin(10*PI*x)+1.0;

 return fitness;
}
```

Listagem 5.2 – Função fornecida pelo usuário

Neste exemplo, a função `avalia` recebe um membro da população como parâmetro e decodifica esta string binária para seu valor correspondente ao intervalo -1 a 2 através da função `BinpReal`, o valor é armazenado na variável `x`. Em seguida é calculado o valor de aptidão deste indivíduo propriamente dito.

A função da biblioteca paralela `BinpReal` mapeia uma string binária em um correspondente valor real delimitado por dois extremos. Tendo a função de avaliação definida, basta usar as demais chamadas da biblioteca para a sua otimização.

Uma aplicação completa, segue o modelo descrito na listagem 5.2. Este é o protótipo mínimo requerido e é composto das seguintes partes:

- inclusão da biblioteca de cabeçalho `agmpi.h`;
- uma função de avaliação (semelhante à listagem 1); e
- uma função principal (main).

O arquivo de cabeçalho `agmpi.h` define todas as estruturas e protótipos de funções visíveis aos usuários. A função de avaliação será sempre particular ao tipo de problema que se deseja analisar e deve ser fornecida pelo usuário.

A função principal `main` inicializa o ambiente paralelo através da chamada MPI_Init (requerida) e evolui uma população de indivíduos durante um certo número de gerações. Na definição de `main`, há a necessidade de se declarar uma variável do tipo `Parametros`.

A listagem 5.2 completa a aplicação com a utilização das demais rotinas requeridas.

```
include "agmpi.h"

double avalia(unsigned *crom);

double avalia(unsigned *crom) {
  double fitness=0.0,x=0.0,
         PI=3.14159265358979238462643;

  x=BinpReal(crom, -1, 2);
  fitness = x*sin(10*PI*x)+1.0;

  return fitness;
}

int main(int argc, char **argv) {
  Parametros ambiente;
  MPI_Init(&argc, &argv);

  Inicializa(&ambiente);
  Modifica(&ambiente);

  Calcula(&ambiente,avalia);

  Printf("Melhor Fitness = %f\n",  FitnessMelhorIndividuo());

  MPI_Finalize();
  Return 0;
}
```

Listagem 5.3 – Implementação do problema proposto

A variável `ambiente,` do tipo `Parametros` é utilizada pelas demais funções da biblioteca e declara os parâmetros do AG que se baseia nos seguintes valores:

Tamanho da população	30
Tamanho do cromossomo	8
Número máximo de gerações	100
Probabilidade de crossover	0.6
Probabilidade de mutação	0.0333
Valor da semente para o GNA	0.123456

Tabela 5.1 – Valores padrão do AG.

Estes parâmetros são efetivamente definidos pelas funções `Inicializa()` e `Modifica()` que são requeridas em todas as aplicações e devem respeitar esta seqüência de chamadas.

A função seguinte, `Calcula`, é que realmente executa o código do AG (avaliação, seleção, crossover e mutação), além de abstrair o paralelismo. Esta exige, além da variável do tipo `Parametros` presentes nas demais funções, o nome da função de avaliação definida previamente pelo usuário. O algoritmo considera como critério de parada somente o número de gerações.

Ao final, será impresso o valor do melhor individuo através da chamada `FitnessMelhorIndividuo`(). Outras opções que retornam dados do problema estão na tabela 5.2. O próprio nome sugere sua utilização.

	Função
1	`TotalCrossover()`
2	`TotalMutacoes()`
3	`GeracaoMelhorIndividuo()`
4	`StringMelhorIndividuo()`

Tabela 5.2 – Conjunto de funções de retorno

O protótipo destas funções não define nenhuma passagem de parâmetro. Estas chamadas bem como `FitnessMelhorIndividuo`() são funções que retornam valores úteis a uma análise estatística e devem ser utilizadas somente depois da função

`Calcula` e antes de `MPI_Finalize`(). As funções 1 e 2 acumulam durante o processamento o número de operações de crossover e o número mutações efetuadas respectivamente e as funções 3 e 4 são dados referentes à melhor solução encontrada.

A chamada `MPI_Finalize` encerra o ambiente paralelo e também é requerida na função principal.

A forma como o programa foi elaborado permite que o problema seja computado de acordo com os parâmetros da tabela 5.1. Entretanto isto não limita o usuário a aceitar somente tais valores, a biblioteca fornece meios para que se possa alterar estes padrões através de chamadas específicas, dando uma maior liberdade ao programador. Estas funções estão descritas na tabela 5.3:

Nome da função	**Função**
`PopulacaoMax`	Altera o tamanho da população
`GeracaoMax`	Altera o número de gerações
`CromossomoMax`	Define o tamanho da string binária
`CrossoverMax`	Taxa de crossover
`MutacaoMax`	Taxa de Mutação
`SementeMax`	Altera o valor da semente para o GNA

Tabela 5.3 – Funções de alteração do ambiente

Os protótipos destas chamadas são semelhantes e estão descritas abaixo:

```
FuncaoMax(&var, valor);
```

Onde:

`FuncaoMax` – uma das funções da tabela 5.3;

`var` – variável do tipo Parametros;

`valor` – valor a ser alterado.

A listagem 5.3 apresenta a função `main`, agora com modificações dos parâmetros genéticos. Vale ressaltar que a alteração destes valores é opcional e elas podem ser feitas em uma ordem diferente do exemplo bem como é opcional a escolha

de qual função usar. A única exigência para o uso destas funções é que elas estejam entre as chamada `Inicializa` e `Modifica`.

```
int main(int argc, char **argv) {
  Parametros ambiente;

  MPI_Init(&argc, &argv);

  Inicializa(&ambiente);
     PopulacaoMax    (&ambiente, 60);
     GeracaoMax      (&ambiente, 150);
     CromossomoMax   (&ambiente, 20);
     CrossoverMax    (&ambiente, 0.2);
     MutacaoMax      (&ambiente, 0.01);
     SementeMax      (&ambiente, 0.0269);
  Modifica(&ambiente);

  Calcula(&ambiente,avalia);

  Printf("Melhor Fitness = %f\n", FitnessMelhorIndividuo());
  Printf("Ocorreu na geração %i\n",GeracaoMelhorIndividuo());

  MPI_Finalize();

  Return 0;
}
```

Listagem 5.4 – Alteração dos parâmetros genéticos

5.5 Considerações finais

Este capítulo detalhou a implementação do protótipo. Vale ressaltar que do modelo proposto até a confecção da biblioteca propriamente dita, um longo caminho de estudos foi realizado para que suas funcionalidades tivessem as características da transparência e facilidade de uso.

Para a realização dos testes, foi utilizado um pequeno cluster no estilo Beowulf [BEO, 2002], composto por três máquinas rodando RedHat Linux 7.3 e a biblioteca de passagem de mensagem MPICH versão 1.2.5.

O tipo de problema em questão, por ser de baixa complexidade, convergiu para um bom resultado em um curto espaço de tempo. Por este motivo, o teste consistiu em

comparar a qualidade das respostas geradas pelas implementações da listagem 5.3 e 5.4, desconsiderando as tomadas de tempo para a sua realização.
Os resultados estão listados na tabela 5.4.

	Listagem 5.2	Listagem 5.3
Resultado	1.847058	1.850468

Tabela 5.4 – Melhor fitness de cada implementação

Nota-se que a listagem 4 tem um valor bem próximo ao descrito na equação (4). Lembramos que os parâmetros genéticos desta implementação foram alterados explicitamente pelo usuário.

6 Conclusão

Os AGs são uma técnica de busca amplamente difundida e aceita, e os parâmetros que regem seu funcionamento são numerosos. Empregar esta técnica em um ambiente paralelo tende a ampliar o grau de dificuldade de uma aplicação prática.

Por outro lado, o uso de uma biblioteca de funções como a descrita aqui, onde em um só local é possível fazer uso de uma robusta técnica heurística sobre um sistema paralelo, oculta do usuário final detalhes que muitas vezes não seriam necessários para a resolução de seu problema e que teria de se preocupar se assim não fosse feito.

O exemplo utilizado para descrever o funcionamento da biblioteca, evidentemente não requer um alto poder de processamento para sua resolução, mas serviu de forma didática para demonstrar a facilidade de uso por parte do usuário, que foi um dos objetivos do projeto pensado inicialmente.

6.1 Trabalhos futuros

Ainda que a biblioteca atenda pelo nome de "genérica", ela pode ter sua utilização restrita a algumas áreas ou até a impossibilidade de uso, devido à forma como foi feita a sua implementação. Isto se deve principalmente pela escolha da codificação binária para representação das soluções. Para um grande número de aplicações, como descritos em [MIC, 1996], uma forma de representação por números reais ou inteiros seria mais indicada. Como trabalho futuro, então, seria interessante ampliar as formas de codificação de soluções. Isto implica, também, em novos operadores de seleção, reprodução e mutação específicos para cada tipo.

Uma maior atenção também deve ser dispensada para o tratamento de problemas que envolvam restrições ou que manipulem múltiplas variáveis.

Quanto à parte paralela do algoritmo, novas formas podem ser exploradas como, por exemplo, a utilização de modelos híbridos [PAZ, 1997] e a implementação do operador de Migração [PAZ, 1998a].

Referências Bibliográficas

[ALB, 1995] ALBA, Enrique; TROYA, José M. A Survey of Parallel Distributed Genetic Algorithms. Disponível em: <http://polaris.lcc.uma.es/~eat/publi.html>. Acesso em: 6 agosto 2003.

[ALB, 2001] ALBA, Enrique; COTTA, Carlos. Tutorial on Evolutionary Optimization Disponível em: <http://neo.lcc.uma.es/TutorialEA/semEC/main.html>. Acesso em: 10 fevereiro 2003.

[ALM, 1994] ALMASI, G. S; GOTTLIEB A. High Parallel Computing. 2ª ed, The Benjamin Cummings Publishing Company, Inc., 1994.

[ALV, 2002] ALVES, Marcos J. P. Construindo Supercomputadores com Linux. Rio de Janeiro: Brasport, 2002.

[BEN, 1993] BEN-DYKE, A. D. Architectural Taxonomy: A brief review. University of Birmingham, 1993.

[BEO, 2002] BEOWULF. The Beowulf FAQ: Beowulf mailing list FAQ, version 2 Disponível em: <http://www.beowulf.org>. Acesso em: 12 março 2003.

[CAS, 2002] CASTRO, Leandro N. de; ZUBEN, Fernando J. Von. Computação Evolutiva: Uma Abordagem Pragmática. Disponível em: <http://www.dca.fee.unicamp.br/~vonzuben/courses/ia707.html>. Acesso em: 06 março 2003.

[COS, 2000] COSTA, Umberto Souza da. Ordenação de Variáveis de BDDs Utilizando Algoritmos Genéticos Paralelos. Disponível em: <http://www.dimap.ufrn.br/Mestrado/Producao/dissertacoes.html>. Acesso em: 23 setembro 2003.

[COL, 1994] COLOURIS, G; DOLLIMORE, J; KINDBERG, T. Distributed System Concepts and Design. 2ª ed, Addison-Wesley Publishing Company, 1994.

[COR, 1999] CORSO, Thadeu B. Crux: Ambiente Multicomputador Configurado por Demanda. Tese de Doutorado. CPGEE. Universidade Federal de Santa Catarina. Florianópolis. 1999.

[CRU, 2002] CRUZ, Pedro; et al. Algoritmos Genéticos. Disponível em: <http://mayaweb.upr.clu.edu/~fvega/Artificial_Intelligence/GA1.pdf>.

Acesso em: 13 novembro 2002.

[DAR, 2003] DARWIN, Charles. The origin of Species. Disponível em: <http://www.literature.org/authors/darwin-charles/the-origin-of-species/>. Acesso em: 22/06/2003.

[DAV, 2002] DAVIES, Robies. An Introduction to MPI and Parallel Genetic Algorithm. Cardiff HPC Training & Education Centre.

[DEM, 2003] DEMAINE, Erik. TOMPI. Disponível em: <http://theory.lcs.mit.edu/~edemaine/>. Acesso em: 16 abril 2003.

[DUN, 1990] DUNCAN, R. A Survey of Parallel Computer Architectures, IEEE Computer, pp.5-16, Fevereiro, 1990.

[FAL, 2002] FALQUETO, Jovelino. Inspiração Biológica em IA. Tese de Doutorado. Universidade Federal de Santa Catarina, Centro Tecnológico. Programa de Pós -Graduação em Computação. Disponível via www em: <http://www.inf.ufsc.br/~falqueto/aGraduacao/INE5633Sist_Intel/Page_INE5633Sist_Intel.html>. Acesso em> Janeiro de 2004.

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

[FOR, 2003] MPI-FORUM. MPI: A Message-Passing Interface Standard Disponível em: <http://www.mpi-forum.org/docs/mpi-11-html/mpi-report.html>. Acesso em: 14/1/2003.

[FUR, 1998] FURTADO, João Carlos. Algoritmo Genético Construtivo Na Otimização De Problemas Combinatoriais De Agrupamentos. Disponível em: <www.lac.inpe.br/~lorena/sbpo98/agc-clust.pdf>. Acesso em: 15 janeiro 2003.

[GOL, 1989] GOLDBERG. David. E. (1989), Genetic Algorithms in Search, Optimization, and Machine Learning, Reading, MA: Addison-Wesley. 1989

[GRO, 2003] GROPP, William; LUSK, Ewing. Installation and User's Guide to mpich, a Portable Implementation of MPIVersion 1.2.5The chp4 device for Workstation Networks. Disponível em: <http://www-unix.mcs.anl.gov/mpi/mpich/>. Acesso em: 15 outubro 2003.

[HOR, 2003] HORTON, Thomas B. Using the gdb and xxgdb Debuggers under UNIX

Dept. of Computer Science and Engineering. Florida Atlantic University. Disponível em:< http://www.cse.fau.edu/~cot3002l/debug.html>. Acesso em: 15 outubro 2003.

[JUN, 1999] JUNIOR, José Mazzuco. Uma abordagem híbrida do problema da pro dução através dos algoritmos Simulated Annealing e Genético. Tese de Doutorado. Programa de Pós-Graduação em Engenharia de Produção. Universidade Federal de Santa Catarina. Florianópolis. 1999. Disponível em: <http://www.eps.ufsc.br/teses99/mazzuco/>. Acessado em: fevereiro de 2003.

[KLY, 2003] KLYAGIN, Konstantin, Introducing Motor: a Programming IDE for Linux. Disponível em: <http://konst.org.ua/en/writings>. Acesso em 06/03/2003.

[LAG, 1995] LAGUNA, Manuel; MOSCATO, Pablo. Algoritmos Genéticos. Disponível em: <http://www.ing.unlp.edu.ar/cetad/mos/publications.html>. Acesso em: 27/02/2003.

[LIN, 1998] LINHALIS, Flávia; FIATS, Mayb Iara. PVM e MPI. Instituto de Ciências Matemáticas de São Carlos. Universidade de São Paulo. 1998

[LOU, 2002] LOURENÇO, Carlos Eduardo Botelho. Algoritmos Genéticos. Disponível em: <http://black.rc.unesp.br/ccomp/algoritmo/direito.html>. Acesso em: 13 novembro 2002.

[LUS, 2003] LUSK, Ewing. MPI: A Message Passing Interface Disponível em: <http://www-fp.mcs.anl.gov/~lusk/papers/mpi-worksho/paper.html>. Acesso em: 10 abril 2003.

[MAZ, 1996] MAZZONETO, Giovanni Davi. Configuração e Montagem de um cluster para Máquinas Distribuídas. Disponível em: <http://www.bibliovirtual.fw.uri.br/bvinf/tc2001/tcgiovanni.html>. Acesso em: 15 janeiro 2003.

[MIC, 1996] MICHALEWICZ, Z. & SCHOENAUER, M. "Evolutionary Algorithms for Constrained Parameter Optimization Problems", Evolutionary Computation, vol. 4, no. 1, pp. 1-32, 1996.

[MIC, 1996b] MICHALEWICZ, Z., Genetic Algorithms + Data Structures = Evolution Programs. Third edition. New York: Springer-Verlag. 1996.

[MUH, 2002] MÜHLENBEIN, Heinz. Evolution in Time and Space - The Parallel Genetic Algorithm. Disponível em: <http://neo.lcc.uma.es/ParallelEA/papers.html>. Acesso em: 12 junho 2003.

[MUL, 1993] MULLENDER. Distributed System. 2ª ed, ACM PRESS Frontier Series Addison-Wesley Publishing Company, 1993.

[NOW, 1992] NOWOSTAWSKI, Mariusz; POLI, Riccardo. Parallel Genetic Algorithm Taxonomy. Disponível em: <http://cswww.essex.ac.uk/staff/poli/papers/publications.html>. Acesso em: 26 agosto 2003.

[OBI, 1998] OBITKO, Marek. Genetic Algorithm. Disponível em: <http://cs.felk.cvut.cz/~xobitko/ga/>. Acesso em: 12 março 2003.

[OCH, 2000] OCHI, Luiz Satoru. Algoritmos Genéticos: Origem e Evolução. Disponível em: <http://pub2.lncc.br/sbmac/com-fig/public/bol/BOL-2/artigos/satoru>. Acesso em: 11 março 2003.

[PAC, 1998] PACHECO, Peter. A User Guide to MPI. Department of Mathematics. University of San Francisco. San Francisco. March, 1998.

[PAC, 2001] PACS TRAINING GROUP. Introduction to MPI. University of Illinois. 2001. Disponível em: <http://webct.ncsa.uiuc.edu:8900/webct/public/home.pl>. Acesso em: 2 janeiro 2003.

[PAZ, 1995] PAZ; Erick Cantú. A Summary of Research on Parallel Genetic Algorithms. Disponível em: <http://citeseer.nj.nec.com/27505.html>. Acesso em: 6 fevereiro 2003.

[PAZ, 1997] PAZ; Erick Cantú. Designing Efficient Master-Slave Parallel Genetic Algorithms. Disponível em: <http://gal4.ge.uiuc.edu/technreprts.html>. Acesso em: 6 fevereiro 2003.

[PAZ, 1998] PAZ; Erick Cantú. A Survey of Parallel Genetic Algorithms. Disponível em: <http://www.llnl.gov/CASC/research.shtml>. Acesso em: 6 fevereiro 2003.

[PAZ, 1998a] PAZ; Erick Cantú. Designing Scalable Multi-Population Parallel Genetic Algorithm. Disponível em: <http://gal4.ge.uiuc.edu/technreprts.html>. Acesso em: 6 fevereiro 2003.

[POL, 1996] POLI, Riccardo. Introduction to Evolutionary Computation Disponível em: <http://www.cs.bham.ac.uk/~rmp/slide_book/slide_book.html>. Acesso em: 10 fevereiro 2003.

[QUI, 1987] QUINN, M. J. Designing Efficient Algorithms for Parallel Computers McGraw Hill, 1987.

[SNI, 1996] SNIR, Marc; et. al. MPI: The Complete Reference. Ed. Mit Press. Cambridge.1996

[SOA, 1997] SOARES, Luiz Fernando G; LEMOS, G; COLCHER, S. Redes de Computadores: Das LANs, MANs e WANs às redes ATM. 2ª ed. Rio de Janeiro: Campus, 1997.

[SOU, 1996] SOUZA, Paulo S. Lopes de. Máquina Paralela Virtual em Ambiente Windows. Dissertação de Mestrado. Instituto de Ciências Matemáticas de São Carlos. Universidade de São Paulo. Maio de 1996.

[SOU, 2003] SOUZA, Marcone Jamilson Freitas. Problema do Caixeiro Viajante com Coleta de Prêmios – Descrição do Problema. Disponível em: <http://www.decom.ufop.br/prof/marcone/index.htm>. Acesso em: 20 outubro 2003.

[TAN, 1997] TANENBAUM, Andrew S. Sistemas Operacionais Modernos. Ed: Prentice Hall do Brasil Ltda. Rio de Janeiro. 1997.

[WAL, 1992] WALKER, David. Standards for Message Passing in a Distributed MemoryEnvronment. Technical report, Oak Ridge National Laboratory,August 1992.

Anexos

Anexo A – Exemplo que otimiza a função x^2 descrita por Goldberg

```
include "agmpi.h"

double xQuadrado(unsigned *crom);

double xQuadrado (unsigned *crom) {
  double fitness=0.0,
        x=0.0,

  x=BinpReal(crom, 0, 1);
  fitness = x*x;

  return fitness;
}

int main(int argc, char **argv) {
  Parametros ambiente;
  MPI_Init(&argc, &argv);

  Inicializa(&ambiente);
  Modifica(&ambiente);

  Calcula(&ambiente, xQuadrado);

  printf("\n");
  printf("**** Dados Usados pelo Programa ****\n");
  printf("Tamanho da população = %i\n",TamanhoPopulacao(&ambiente));
  printf("Probabilidade Crossover = %f\n",ProbCrossover(&ambiente));
  printf("Mutacao = %f\n",ProbMutacao(&ambiente));
  printf("Melhor Fitness = %f\n",  FitnessMelhorIndividuo());

  printf("\n");
  printf("**** Estatisticas ao final do processamento  ****\n");
  Printf("Melhor Fitness = %f\n", FitnessMelhorIndividuo());
  Printf("Ocorreu na geração %i\n",GeracaoMelhorIndividuo());
  Printf("Operacoes de Crossover = %f\n", TotalCrossover());
  Printf("Total de Mutacoes = %f\n", TotalMutacoes());
  Printf("String do Melhor Individuo = %f\n",StringMelhorIndividuo());
  MPI_Finalize();

return 0;
}
```

Anexo B – Exemplo de Maximização

Este exemplo acha o máximo da função -x^2+8x+15 no intervalo 0 a 25.

```
include "agmpi.h"

double f(unsigned *crom);

double f(unsigned *crom) {
  double fitness=0.0,
         x=0.0;

  x=BinpReal(crom, 0, 25);
  fitness = -1*(x*x) + 8*x + 15;

  return fitness;
}

int main(int argc, char **argv) {
  Parametros ambiente;
  MPI_Init(&argc, &argv);

  Inicializa(&ambiente);
      PopulacaoMax(&ambiente, 30);
      GeracaoMax(&ambiente, 6);
      CromossomoMax(&ambiente, 5);
      CrossoverMax(&ambiente, 0.6);
      MutacaoMax(&ambiente, 0.0333);
      SementeMax(&ambiente, 0.4269);
  Modifica(&ambiente);

  Calcula(&ambiente, f);

  printf("\n");
  printf("**** Dados Usados pelo Programa ****\n");
  printf("Tamanho da população = %i\n",TamanhoPopulacao(&ambiente));
  printf("Probabilidade Crossover = %f\n",ProbCrossover(&ambiente));
  printf("Mutacao = %f\n",ProbMutacao(&ambiente));
  printf("Melhor Fitness = %f\n",  FitnessMelhorIndividuo());
  MPI_Finalize();

return 0;
}
```