unesp

UNIVERSIDADE ESTADUAL PAULISTA

Programa de Pós-Graduação em Engenharia Elétrica

DESENVOLVIMENTO DE UM PROGRAMA PARA ROTEAMENTO E ALOCAÇÃO DE ELEMENTOS DE CIRCUITOS EM ARQUITETURAS PARALELAS UTILIZANDO ALGORITMOS GENÉTICOS

CECILIO MERLOTTI RODAS

DO

1210001428

Ilha Solteira - SP

DESENVOLVIMENTO DE UM PROGRAMA PARA ROTEAMENTO E ALOCAÇÃO DE ELEMENTOS DE CIRCUITO EM ARQUITETURAS PARALELAS UTILIZANDO ALGORITMOS GENÉTICOS

CECILIO MERLOTTI RODAS

Orientador: Prof. Dr. Norian Marranghello

Proc. 063/04-NAD 04

UNESP - "CAMPUS DE ILHA SOLTEIRA"
SERVIÇO TÉC. DE BIBLIOTECA E DOCUMENTAÇÃO

DATA DE CHEGADA	DATA DE TOMBO
18.03.04	31.03.04
REGISTRADO POR	TOMBO
Ailza	Te. 1428
AQUISIÇÃO	CLASSIFICAÇÃO
Doação Autor R$ 10,00	R685d

ex
sys 215267
sys 56130

Dissertação apresentada à Faculdade de Engenharia de Ilha Solteira - Universidade Estadual Paulista "Júlio de Mesquita Filho", como parte das exigências para a obtenção de título de Mestre em Engenharia Elétrica.

1210001428

ILHA SOLTEIRA

FEVEREIRO - 2004

BCpIS - FEIS • UNESP

FICHA CATALOGRÁFICA

Elaborada pela Seção Técnica de Aquisição e Tratamento da Informação/Serviço Técnico de Biblioteca e Documentação da FEIS/UNESP

Rodas, Cecilio Merlotti
R685d Desenvolvimento de um programa para roteamento e alocação de elementos de circuitos em arquiteturas paralelas utilizado algoritmos genéticos / Cecilio Merlotti Rodas. – Ilha Solteira : [s.n.], 2004
86 p. : il.

Dissertação (mestrado) – Universidade Estadual Paulista. Faculdade de Engenharia de Ilha Solteira, 2004

Orientador: Norian Marranghello
Bibliografia: p. 83-86

1. Algoritmos genéticos. 2. Simulação de circuitos. 3. Processamento paralelo.

30403014

Desenvolvimento de um programa para roteamento e alocação de elementos de circuitos em arquiteturas paralelas utilizando algoritmos genéticos

Cecilio Merlotti Rodas

DISSERTAÇÃO SUBMETIDA À FACULDADE DE ENGENHARIA - CAMPUS DE ILHA SOLTEIRA – UNESP – COMO PARTE DOS REQUISITOS NECESSÁRIOS PARA A OBTENÇÃO DO TÍTULO DE MESTRE EM ENGENHARIA ELÉTRICA

Prof. Dr. Aparecido Augusto de Carvalho
Coordenador do Programa de Pós–Graduação
em Engenharia Elétrica

COMISSÃO EXAMINADORA:

Prof. Dr. Norian Marranghello – orientador

Prof. Dr. Flurio Damiani

Prof. Dr. José Roberto Sanches Mantovani

Ilha Solteira/SP, fevereiro de 2004.

BCpIS - FEIS - UNESP

AGRADECIMENTO

Meus agradecimentos,

A Deus pela vida e todas as oportunidades que Ele tem me concedido.

A meus pais e irmãos pelo apoio e sacrifícios realizados para que eu pudesse conquistar novos horizontes.

A meu orientador, Norian Marranghello; pela paciência, apoio, amizade e por sempre estar disposto a compartilhar seus conhecimentos.

A meu sobrinho, Rodrigo e ao Prof. Sérgio pela ajuda técnica.

A minha namorada, Cristiane, por me amparar na busca de melhores caminhos.

Ao amigo Cristiano pela amizade, incentivo e companheirismo nas horas difíceis.

A todos meus queridos familiares e amigos que de uma maneira ou outra estiveram colaborando e apoiando.

Aos professores do curso de pós-graduação em Engenharia Elétrica da FEIS.

Ao Centro Universitário de Votuporanga e meus colegas de trabalho.

À ajuda financeira do CNPq, com o qual pude contar durante meus estudos.

BCpIS - FEIS - UNESP

RESUMO

O presente trabalho utiliza Algoritmos Genéticos Paralelos para o mapeamento de elementos de circuito numa arquitetura multiprocessada.

Para a paralelização do algoritmo genético (AG) foi utilizado o *software* PVM (*Parallel Virtual Machine)*. A implementação se baseou no modelo "*ilha*" de AG, com o padrão *mestre/escravo*.

Os resultados obtidos apontam para uma velocidade de processamento paralelo da ordem de 50% melhor que utilizando a implementação seqüencial no mesmo equipamento. Além disto, a qualidade dos resultados foi cerca de 6% melhor do que os obtidos com a implementação seqüencial correspondente.

cm 1 2 3 4 5 6 7 8 unesp 12 13 14 15 16 17 18 19

ABSTRACT

This work uses Parallel Genetic Algorithms for mapping circuit elements onto a three-dimensional architecture.

For the parallel processing, PVM (Parallel Virtual Machine) has been used. The "island" model of genetic algorithms, in a master/slave implementation has been used.

The results show that parallel processing is about 50% faster than the serial implementation in the same equipement. Furthermore, a 6% improvement in the quality of the best sample when compared to the corresponding serial program outcomes has been observed.

unesp

ÍNDICE

cm 1 2 3 4 5 6 7 8 unesp 12 13 14 15 16 17 18 19

LISTA DE FIGURAS

cm 1 2 3 4 5 6 7 8 unesp 12 13 14 15 16 17 18 19

INTRODUÇÃO

O algoritmo genético (AG) é um método de otimização baseado no processo de seleção natural. Possíveis soluções (indivíduos) são criadas e depois são submetidas a processos de *cruzamento* e *mutação*. Por meio desses processos ele seleciona os indivíduos mais aptos. O algoritmo genético paralelo (AGP) é implementado para que o processamento chegue a uma solução otimizada em menor espaço de tempo.

Durante o projeto de circuitos a simulação é importante porque permite testar o comportamento que ele terá após ser construído. Dessa maneira é possível prever seu funcionamento e, se necessário, fazer as alterações necessárias. A simulação procura otimizar a qualidade do circuito.

Uma metodologia foi desenvolvida [20] a fim de simular circuitos. Trata-se de uma arquitetura multiprocessada tri-dimensional (ABACUS – *hArdware BAsed CircUit* Simulation) capaz de simular, em seus processadores (*hardware*), elementos de circuito.

O presente trabalho visa a eficiência de algoritmos genéticos paralelos no mapeamento de elementos de circuito sobre tal arquitetura. O programa implementado neste trabalho não é capaz de simular um dado circuito. Ele faz o mapeamento dos elementos de um circuito sobre a arquitetura ABACUS.

No capítulo 1 deste trabalho são abordadas, resumidamente, as técnicas tradicionais de simulação de circuitos e a especificação da arquitetura ABACUS.

unesp

O capítulo 2 trata dos AGs (algoritmos genéticos). É mostrado o funcionamento padrão de um AG; a maneira como as soluções podem ser representadas e os procedimentos de cruzamento e mutação. Além disto, faz-se uma revisão bibliográfica.

O ambiente de programação paralela, seu objetivo e os métodos que possibilitam sua implementação são considerados no capítulo 3.

A metodologia utilizada para se chegar aos resultados alcançados é discutida no capítulo 4. No qual também são discutidas as variações necessárias nos processos de cruzamento e mutação do AG padrão para a resolução do problema proposto.

Alguns detalhes da implementação são abordados no capítulo 5. Nele também é mostrada a estrutura de dados utilizada para a implementação do programa desenvolvido.

No capítulo 6 são apresentados os resultados dos testes realizados assim como o comentário dos mesmos.

No capítulo 7 são feitas as considerações.

1 SIMULAÇÃO DE CIRCUITOS

Neste capítulo são comentadas algumas características do modelo tradicional de simulação de circuitos. É explicada a metodologia ABACUS.

Como a densidade dos circuitos integrados tem crescido muito, o uso de ferramentas de *software* para auxiliar nos seus projeto e fabricação tornou-se indispensável para um processo comercialmente competitivo. O seu objetivo é predizer o comportamento elétrico de um circuito com detalhes suficientes para prover uma idéia muito aproximada de seu comportamento real [15, 20].

Circuitos VLSI típicos possuem centenas de milhares de componentes, sendo necessária uma ferramenta poderosa para simulá-los. Em 1975 foi apresentado um programa de simulação analógica, o SPICE, na universidade da Califórnia, em Berkeley, com boa precisão, mas com alto custo computacional. Desde então novos simuladores foram criados aumentando consideravelmente a velocidade em relação ao SPICE. Apesar disso, tais simuladores ainda não são suficientemente rápidos para tratarem circuitos VLSI, já que gastam muito tempo para resolver o sistema de equações resultantes dos circuitos [20].

Nesse tipo de simuladores tidos como convencionais, quando se tratam circuitos VLSI é necessário montar um sistema de equações muito grande capaz de representá-los. O tratamento deste sistema ocupa grande parte do tempo. O *software* neste caso tem que ser capaz de identificar cada tipo de elemento, representá-lo através de sua fórmula matemática, verificar o tipo de relacionamento que ele terá

unesp

com os demais elementos aos quais ele está conectado e desta maneira montar a equação.

A metodologia ABACUS [20] foi proposta para superar as dificuldades encontradas nos simuladores padrão.

1.1 Arquitetura ABACUS

O ABACUS é uma arquitetura multiprocessada que, ao passar por um processo de mapeamento dos componentes do circuito a simular-se sobre seus processadores, assume as características comportamentais do circuito em questão [5]. A idéia fundamental é que se tenha um arranjo com vários processadores os quais são chamados de MPH (*model processing hardware-element*) – "elemento para processamento de modelos", cada um representando o comportamento de um dos elementos do circuito a ser simulado, ou seja, cada processador assume características comportamentais do elemento que simula. Todos os processadores são controlados por um processador hospedeiro, que é o responsável pela transferência das condições de contorno necessárias para cada processador. Eles são interconectados através de um conjunto de barramentos programáveis de acordo com a topologia do circuito [16].

O processador hospedeiro é mais complexo que os processadores MPH. Ele faz a leitura de uma *netlist*[1] buscando as informações do circuito a ser simulado e

[1] *Netlist* é a relação de componentes de um circuito, de suas conexões e outras informações relativas ao circuito.

unesp

depois gerencia o arranjo dos MPHs. Ele identifica os elementos de circuito, suas interconexões e as análises a serem feitas. Se o número de elementos do circuito é tal que não é possível mapeá-lo diretamente no arranjo, o gerente particiona o circuito num número adequado de subcircuitos e, então, mapeia sobre o arranjo cada subcircuito a seu turno, fazendo as conexões necessárias entre cada um deles [20].

A primeira etapa da simulação consiste num processo de associação (mapeamento) dos elementos do circuito aos processadores. Como resultado de um mapeamento específico, pode-se ter, por exemplo, um componente C1 com um dado comportamento sendo simulado por um processador P2. Neste caso, o processador P2 deverá interagir com outros processadores assumindo as características do componente C1. Este é o objetivo do processo de mapeamento, ou seja, definir qual processador ficará encarregado de simular um certo componente do circuito. Finalizada esta etapa, inicia-se o roteamento entre os processadores. Este roteamento consiste em estabelecer caminhos internos entre os processadores para comunicação destes, dois a dois. Estes caminhos são análogos às conexões que seriam estabelecidas no circuito real. Com o mapeamento e o roteamento definidos, obtém-se uma arquitetura que deve comportar-se de maneira equivalente ao circuito que se pretende simular, e que fornecerá informações importantes na fase de projeto de circuito [15, 16].

Um exemplo prático, onde um circuito de três componentes é mapeado sobre uma arquitetura com quatro processadores é ilustrado na figura 1.1. Observa-se ao lado esquerdo da figura o circuito a ser mapeado (acima) e a arquitetura do simulador (abaixo), com seus processadores ainda sem função específica. Ao lado

unesp

direito, vê-se a arquitetura com os componentes do circuito já mapeados sobre seus processadores [15, 16].

O mapeamento da figura 1.1 foi facilmente realizado devido ao número reduzido de componentes. Mas se houvesse uma quantidade tal de elementos de forma a inviabilizar o mapeamento, então seria necessário realizar um particionamento do circuito. O objetivo do particionamento é realizar uma separação do circuito alvo em diversas partes (módulos), isso geralmente ocorre quando o número de elementos é maior que 70% ou 80% do número de processadores disponíveis no arranjo. Para a realização dessa separação, leva-se em conta o volume de comunicação entre as partes propostas. Um baixo volume de comunicação é um indicador de bom particionamento. Então, as partes obtidas são mapeadas e roteadas separadamente na arquitetura [16].

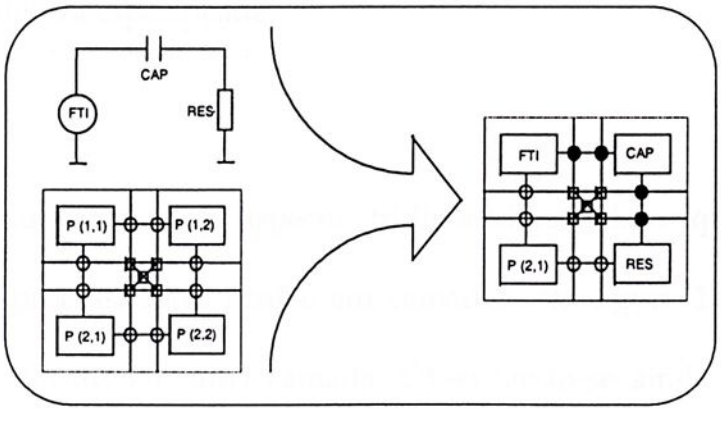


Figura 1.1: Exemplo de mapeamento de um filtro RC em uma matriz de processadores.

Na figura 1.2 aparecem representados os processadores por quadrados sem preenchimento, com a letra P no interior. Também são representadas as três classes de chaves: chave quadrada, representada por um círculo sem preenchimento; chave

pentagonal, representada por um círculo com preenchimento; e chave hexagonal, representada por um quadrado com preenchimento. As chaves hexagonais, também chamadas de inter-camadas, permitem a ligação entre os processadores das diversas camadas do arranjo [20]. As outras duas categorias de chaves servem para interligar os processadores do arranjo aos barramentos adjacentes e/ou a outros processadores, dentro de uma mesma camada.

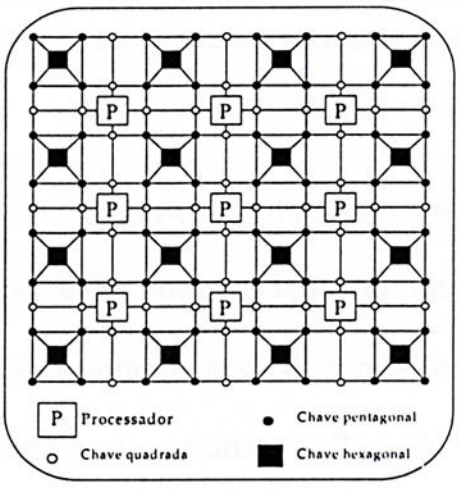


Figura 1.2: Arquitetura especificada

A arquitetura tem um aspecto tridimensional. Isso quer dizer que a distribuição dos processadores ocorre em camadas. Na figura 1.3 aparecem nove processadores dispostos em uma camada. Observando-se ainda a mesma figura, nota-se que as chaves hexagonais possuem apenas quatro pontos de conexão ao invés de seis. Isso ocorre devido aos dois pontos restantes que são responsáveis pela conexão inter-camadas, ou seja, estão num plano perpendicular ao papel. Portanto, estas duas conexões são responsáveis por rotear a comunicação entre processadores que estejam em camadas distintas. A visualização de todas as conexões dessa classe pode ser observada na figura 1.3.

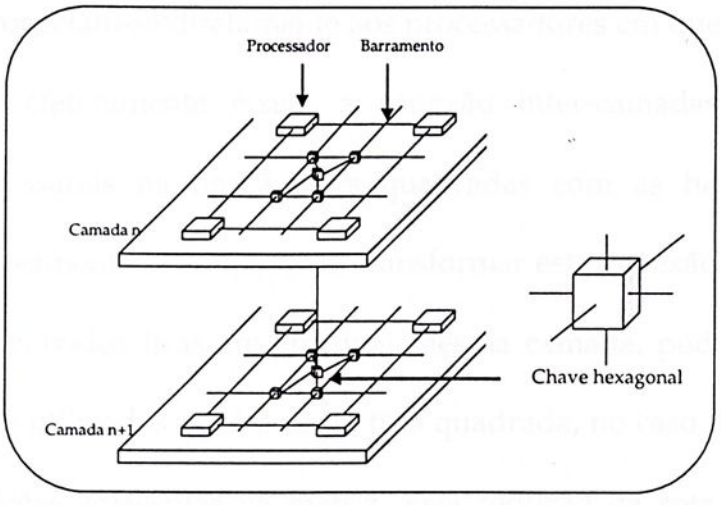


Figura 1.3: Esquema de interconexão em uma matriz de processadores.

A arquitetura mostrada na figura 1.3 apresenta uma grande simetria, ou seja, o número de processadores por linha pode ser igual ao número de processadores por coluna. Assim, utilizando-se a notação MxNxP (onde M é o número de linha de processadores, N é o número de colunas e P o de camadas), a arquitetura representada na figura 1.3 poderia ser representada pela forma 2x2x2 (formando uma matriz 2x2, com 2 linhas com 2 processadores cada e 2 camadas).

Uma peculiaridade importante da arquitetura é que os processadores comunicam-se de maneira direta somente com chaves quadradas. Dessa maneira, toda conexão entre um par de processadores, passa obrigatoriamente no mínimo por dois destes componentes; um no início da rota, na origem da conexão, outro conectando o final da rota ao processador destino.

Quando há necessidade de processadores se conectarem a processadores de uma camada adjacente há um aumento do custo operacional, já que para que essa conexão seja realizada o sinal tem que passar, no mínimo, por duas chaves

quadradas, que conectam-se diretamente aos processadores em questão, duas chaves hexagonais, que efetivamente fazem a conexão inter-camadas e duas chaves pentagonais, necessárias na ligação das quadradas com as hexagonais. Se, no momento do mapeamento, fosse possível transformar esta conexão, de forma que os processadores envolvidos ficassem em uma mesma camada, poder-se-ia reduzir o número de chaves utilizadas a até dois do tipo quadrada, no caso dos processadores estarem em posições adjacentes na matriz. Esta redução da rota a ser percorrida refletir-se-ia de maneira positiva na qualidade final do simulador.

É importante notar que, no escopo deste trabalho, quando se fala em otimização de custos operacionais e desempenho é o mesmo que minimização das rotas de conexão entre os processadores. Ou seja, o número de processadores que fazem parte da simulação de um certo circuito é sempre o mesmo, portanto, uma solução com boa qualidade será medida em função da quantidade e do tipo de chaves. Quanto menos chaves forem necessárias, maior será o desempenho e menor o custo [15, 16].

Feito o mapeamento, envolve-se também nos preparativos para a simulação a especificação de caminhos internos à arquitetura, estabelecidos para a comunicação entre processadores. Desse modo criam-se conexões análogas às existentes no circuito real. Terminados o mapeamento e o roteamento, o *software* de simulação assume o controle do *hardware* de forma a executar a simulação [15].

cm 1 2 3 4 5 6 7 unesp 1 12 13 14 15 16 17 18

2 ALGORITMOS GENÉTICOS

Neste capítulo são mostrados os processos padrões de um algoritmo genético, sua aplicação na otimização de problemas e uma revisão bibliográfica.

Os algoritmos genéticos (AGs) têm despertado o interesse de muitos pesquisadores como um método para resolver problemas de otimização. A quantidade de problemas para os quais os AGs têm sido utilizados é bem ampla. As aplicações práticas de AG podem ser achadas em várias áreas da engenharia, como em sistemas de controle, otimização de funções, processamento de sinais, problemas combinatórios, etc. [8, 13, 14].

Os AGs constituem uma família de modelos computacionais inspirados na teoria da evolução Darwiniana, mais especificamente baseados em mecanismos de seleção natural, dentro do conceito de "sobrevivência das espécies" [2, 13]: os indivíduos mais "fortes", com boas características genéticas, sobrevivem e o cruzamento entre eles pode gerar indivíduos melhores; desta maneira, a seleção elimina características inadequadas à população [9].

AG se refere ao modelo introduzido por John Holland em meados da década de 1970 e investigado por seus alunos na universidade de Michigan [8, 13, 14]. O interesse de suas pesquisas era atingir a mesma robustez encontrada em sistemas naturais, adaptados a muitos ambientes. Quatro diferenças principais podem ser

observadas ao comparar AGs com a maioria das outras técnicas de otimização convencionais [13]:

1. a manipulação de uma codificação ao invés do problema em si;

2. busca em paralelo a partir de uma população, não de um simples ponto;

3. busca via amostragem; e

4. busca usando operadores estocásticos, sem regras determinísticas.

É necessário que possíveis soluções para o problema sejam representadas através de *bits*. Cada um destes representa um determinado parâmetro do modelo. O conjunto de todos estes *bits* representativos deve definir uma solução para o problema. Essas soluções podem ser geradas aleatoriamente. Depois, várias operações são realizadas para que as soluções geradas aleatoriamente sejam aperfeiçoadas, buscando-se assim a melhor solução [1].

Os operadores genéticos para seqüência de *bits* atraíram a atenção da maioria dos pesquisadores, em comparação com outros tipos de representação. Quando se vai da teoria para a prática, às vezes é preciso usar operadores genéticos diferentes e mais adaptados aos campos de aplicação.

A terminologia utilizada no tratamento de AGs é baseada na biologia; assim, o termo *população* significa um conjunto de possíveis soluções, *aptidão* é usado para designar a qualidade de uma solução e cada uma das soluções é chamada de *indivíduo* [1, 2, 6, 8, 9, 13].

cm 1 2 3 4 5 6 7 unesp 1 12 13 14 15 16 17 18

Um problema popular bem conhecido nos meios matemáticos, que pode ser usado para exemplificar o funcionamento dos AGs, é o do caixeiro viajante *(travelling salesman problem – TSP)*, ilustrado na figura 2.2. Esse problema é de natureza combinatória e aparece em diversas aplicações, de projeto de circuitos integrados a minimização de rotas. Basicamente, o caixeiro deve visitar cada cidade em um dado território somente uma vez e depois retornar à cidade de origem. O trajeto entre cada par de cidades tem um determinado custo associado. Dado o custo da viagem entre cada uma das cidades, deve-se escolher o itinerário que resultará no menor custo global, sem que o caixeiro passe duas vezes pelo mesmo trajeto.

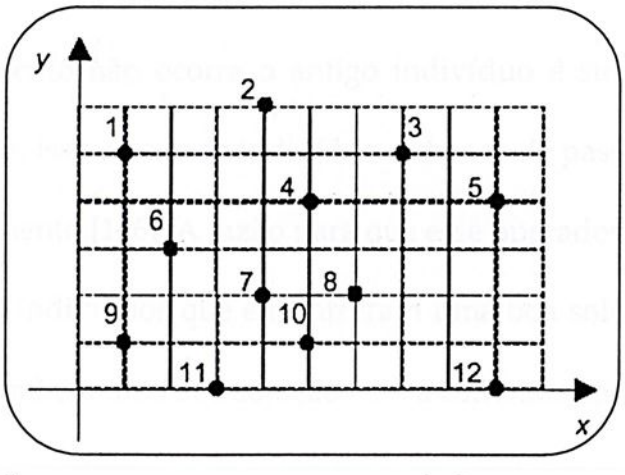


Figura 2.2: Problema do caixeiro viajante com 12 cidades.

A seguir serão discutidos os aspectos fundamentais dos AGs, seus princípios básicos, tais como representação, função de aptidão e operadores de reprodução.

2.1 Operadores de Reprodução

2.1.1 Cruzamento

No processo de cruzamento (*crossover*) ou recombinação, dois indivíduos da população são combinados de maneira a se obter um novo indivíduo com

características dos dois selecionados. Esse método permite que características de um determinado indivíduo sejam transferidas para seus descendentes. A estratégia para este operador pode ser variada, mas a mais comum é a que seleciona o primeiro indivíduo pela sua aptidão e o segundo aleatoriamente. Então uma máscara, M_j, é gerada aleatoriamente, j = 1...L, onde L é o tamanho do cromossomo.

O cruzamento não é aplicado normalmente em todos os pares de indivíduos selecionados para acasalar (fig. 2.3). Uma escolha aleatória é feita, onde a probabilidade de cruzamento a ser aplicada depende das características físicas do problema em estudo.

Caso o cruzamento não ocorra o antigo indivíduo é simplesmente copiado para a nova população, isso dá a cada indivíduo a chance de passar seus genes sem a interrupção do cruzamento [1, 6]. A razão para que esse operador seja utilizado é que a combinação de dois indivíduos que encontraram uma boa solução em subespaços diferentes, encontre também uma boa solução com a combinação dos dois [1].

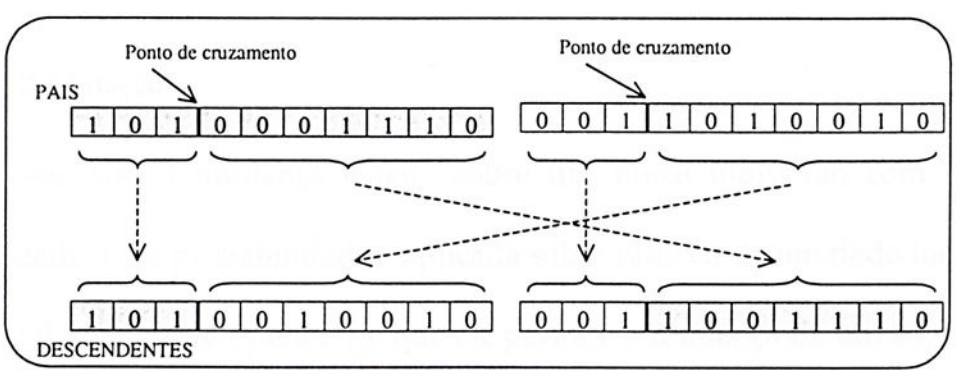


Figura 2.3: Cruzamento ponto-simples.

Diferentes algoritmos de cruzamento foram criados, freqüentemente envolvendo mais de um corte no cromossomo. Também foi observado que, apesar do

uso de dois pontos de corte (*2-point*) incrementar a capacidade de combinações do algoritmo, esse ponto adicional aumenta o tempo de execução necessário para que se atinja o resultado [7].

Um operador de cruzamento que pode manipular seqüências *não-binárias* é o PMX (*Partially – Matched Crossover*) – "cruzamento parcialmente casado". Dados os cromossomos pais, o operador copia uma subcadeia de um dos pais diretamente nas mesmas posições no filho. As posições restantes são preenchidas com os valores que ainda não foram utilizados na ordem em que aparecem no outro pai (Fig. 2.4) [14].

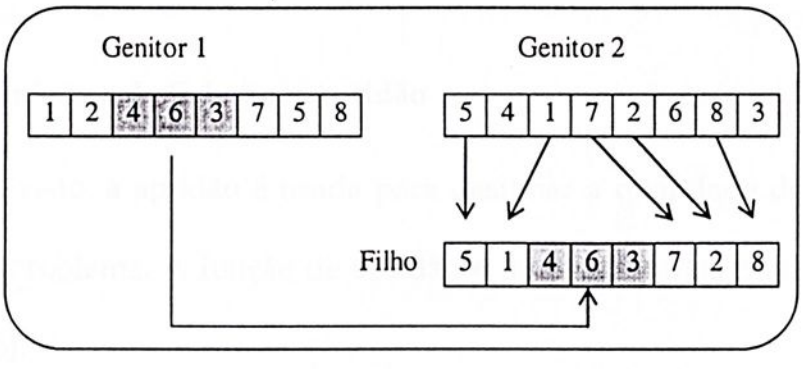


Figura 2.4: Exemplo de aplicação do operador PMX.

2.1.2 Mutação

Nesse caso a mudança é feita sobre um único indivíduo com uma certa probabilidade. Essa probabilidade é aplicada sobre cada *bit* de um dado indivíduo. A vantagem do uso desse operador é que ele permite a mudança de um *bit*, caso todos os indivíduos de uma população possuam um *k-ésimo bit* com o mesmo valor. Nesse caso, mesmo que ocorra o cruzamento esse *bit* permanecerá imutável. O operador de mutação possibilita que esse *k-ésimo bit* seja mudado, oferecendo assim a oportunidade do aumento do espaço de busca [6, 7]. Dessa maneira, as características

que poderiam não ser exploradas pelo cruzamento podem então ser capturadas. A figura 2.5 ilustra a atuação deste operador.

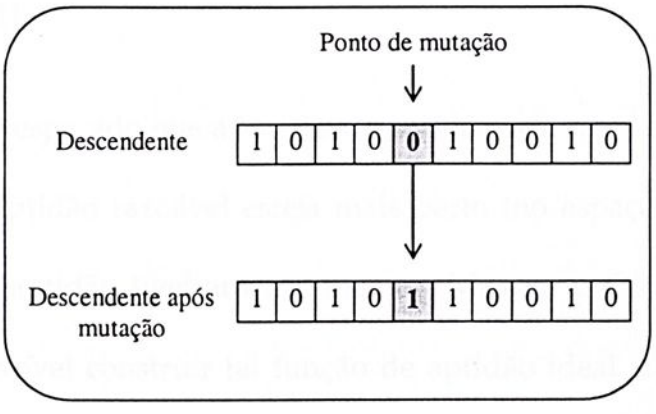


Figura 2.5: Mutação de um gene.

2.2 Mecanismos de Seleção e Aptidão

Como já visto, a aptidão é usada para designar a qualidade de uma solução para um dado problema. A função de aptidão é uma das partes mais cruciais para qualquer AG [6].

Na seleção a aptidão de cada indivíduo é analisada em relação à média de todos os indivíduos da população. A freqüência com que um indivíduo aparecerá na próxima geração é dada pela equação (2.1):

$$hi \alpha \frac{fi}{f} \qquad \text{(Equação 2.1)}$$

onde *hi* é a freqüência, *fi* é a aptidão de um indivíduo e *f* é o valor médio de aptidão de toda a população.

O efeito desse procedimento é possibilitar que um indivíduo com aptidão maior que a média seja mantido e indivíduos com aptidão abaixo da média tenham freqüência menor [1].

Idealmente é esperado que a função de aptidão seja regular, de maneira que o cromossomo com aptidão razoável esteja mais perto (no espaço de parâmetros) de cromossomos com aptidão ligeiramente melhor. Mas na prática isso nem sempre acontece, não é possível construir tal função de aptidão ideal. Todavia, se AGs (ou qualquer técnica de busca) forem bem executados, será possível achar um meio para a construção de funções de aptidão que não tenham *máximos locais*, ou um *máximo global* isolado [6].

Para muitos problemas, a construção da função de aptidão pode ser uma tarefa óbvia. A regra geral na construção da função de aptidão é que ela deve refletir o valor do cromossomo em algum caminho "real". Calcular essa possibilidade não é tão fácil, mas existem situações em que, ao menos, sabe-se o que precisa ser calculado.

Infelizmente o valor de um cromossomo não é sempre uma quantidade facilmente calculável para encaminhar a busca genética. Em problemas de otimização combinatória, tem-se muitas restrições e freqüentemente muitos pontos no espaço de busca representam cromossomos inválidos.

unesp

2.3 O Problema de Convergência Prematura

Quando a população converge para uma solução ótima a gama de aptidões se reduz. Pode ocorrer que durante o processo o problema chegue a uma *convergência prematura* [6]. Essa convergência prematura pode ser entendida da seguinte maneira, o problema atinge uma solução rápida pelo fato do operador de seleção sempre buscar indivíduos com melhor aptidão. Se por um lado, este procedimento permite que se encontre uma solução com maior rapidez, por outro lado, pode levar à desconsideração de características importantes, presentes em indivíduos com baixa aptidão.

Ao se aproximar da convergência o AG perde sua capacidade de buscar soluções melhores: o cruzamento de cromossomos quase idênticos produz pouca diferença nos novos indivíduos. Uma boa solução nesse caso seria o uso do operador de mutação para produzir novas mudanças, mas o processo fica lento quando passa a depender principalmente deste operador [6].

Outro problema que ainda pode ocorrer é o chamado convergência-lenta (*slow finishing*), onde após várias gerações a população evolui mas não consegue atingir um *máximo global*. A média de aptidão pode ser alta e fazer com que haja pouca diferença entre o melhor indivíduo e a média global.

2.4 Escolha dos Indivíduos para o Cruzamento

Consiste em não limitar a escolha de indivíduos a serem usados na geração de novas populações somente aos de melhor qualidade, mas sim a um conjunto de

indivíduos com aptidão "acima da média". Isso possibilitaria que boas características fossem selecionadas e ainda permitiria que algumas características, talvez menos interessantes, também fossem passadas para as gerações futuras. Desse modo evitar-se-ia a convergência prematura, já que haveria uma maior interação entre as diversas soluções, mesmo as menos ajustadas.

Muitos critérios, cada qual com uma razão plausível, existem para selecionar a seqüência para ser trocada. Normalmente a melhor seqüência de cada subpopulação é selecionada para a troca, com a expectativa de aumentar a média de aptidão de cada subpopulação. Mas, como já visto, tal procedimento pode levar a uma *convergência prematura* [2, 6].

Na maioria dos estudos, foi observado que a seleção de seqüências escolhidas aleatoriamente de um subconjunto acima-da-média minimiza o problema de convergência prematura. Esta técnica alternativa ajuda a preservar mais o material genético para as futuras gerações, todavia reduz a probabilidade de seqüências menos ajustadas serem envolvidas no acasalamento das subpopulações.

É importante considerar ainda que, para prevenir convergências prematuras, a operação de mutação também é uma boa solução. Ela tem a habilidade específica para alterar a estrutura do gene de uma maneira aleatória, com um valor probabilístico pequeno. Isso pode ser aplicado em uma seqüência binária (mutação de *bit*) assim como para genes com números-reais (mutação aleatória) [8].

unesp

2.5 Algoritmos Genéticos Paralelos (AGPs)

Apesar das operações genéticas apresentarem melhores resultados quando comparadas com outras técnicas de otimização, muitas vezes leva-se muito tempo para atingir o resultado esperado. Em sistemas complexos, onde são necessários grande número de cálculos, a velocidade se torna muito baixa, o que poderia prejudicar aplicações práticas [1, 3, 8, 9]. Uma das razões para a lentidão das operações genéticas é o cálculo dos valores de aptidão de cada cromossomo no conjunto da população [8].

No intuito de buscar um ganho na velocidade de AGs, considerou-se a possibilidade de paralelizar sua execução. Muitas implementações de AGs foram possíveis em uma variedade de sistemas multiprocessadores, como máquinas MIMD com memória compartilhada, ambientes de troca-de-mensagens como o hipercubo, e ainda arquiteturas SIMD (como o *Connection Machine*) [1].

A seguir, apresenta-se uma proposta de AG paralelo [1]:

1- **Avaliação da função de aptidão:** a aptidão de cada indivíduo pode ser calculada independentemente. Dessa maneira o cálculo da aptidão pode ser distribuído entre os diversos processadores possibilitando o aumento da velocidade do processamento. Pode-se pensar que, aparentemente, a velocidade máxima de processamento será alcançada quando o número de processadores for igual ao número de indivíduos da população. O que nem sempre acontece devido ao aumento de comunicação entre eles.

2- **Cruzamento**: se for escolhido criar cada indivíduo da próxima geração pela aplicação do operador de cruzamento, pode-se fazer isso em paralelo para cada novo indivíduo. A alternativa seria aplicar o cruzamento e colocar o resultado individual na população existente onde seria substituído por um indivíduo com baixa aptidão. Para que isso ocorra deverá existir uma condição para que vários indivíduos se mantenham com boa aptidão. Da mesma maneira que na avaliação de aptidão, quanto maior for o número de processadores, maior será a velocidade obtida.

3- **Mutação**: como a operação de mutação é aplicada a cada *bit* de um dado indivíduo independentemente dos demais, isso permite sua paralelização.

No entanto, nem sempre há um grande ganho de velocidade nos itens acima citados, já que quanto maior o número de processadores, mais intensa será a troca de informações entre eles, o que pode reduzir a velocidade de processamento. Além do que poderia haver dependência de dados, ou seja, um processo poderia ser atrasado pela espera de dados a serem calculados e fornecidos por um outro processo [1].

Infelizmente, um sistema multiprocessador com memória compartilhada usando, por exemplo, sistemas de comunicação através de barramentos, não é capaz de expandir muito a quantidade de processadores elementares (PEs). Além disso, os barramentos podem causar um congestionamento das informações que por eles transitam. Por essa razão, muitos dos sistemas com vários processadores usam memória local e implementam a comunicação via troca-de-mensagens (esse item será abordado mais adiante).

unesp

A abordagem mais comum consiste num algoritmo distribuído onde a população é particionada em várias subpopulações que evoluem em paralelo e onde periodicamente há troca de indivíduos. Este modelo é chamado de *ilha* [3]. Nele, cada população é manipulada independentemente. Para que a proliferação de bom material genético ocorra, pode ser feita uma interação entre as diversas subpopulações de tempos em tempos, possibilitando a troca de características entre os diversos conjuntos populacionais [1, 2, 3, 8]. Esse método pode evidentemente reduzir a interação entre alguns PEs.

Alguns autores [1, 2, 3, 8] argumentam que este método aumenta o desempenho em relação ao AG serial, pelo fato de que cada subpopulação evolui para uma otimização sem causar interferência nas demais. A combinação dessas várias populações já com bons resultados pode melhorar ainda mais o resultado final [1].

3 AMBIENTES DE PROGRAMAÇÃO PARALELA

Neste capítulo são discutidos o significado de processamento paralelo e alguns ambientes de troca-de-mensagens.

Processamento paralelo é um termo usado para denotar técnicas de processamento concorrente de dados, permitindo que vários processos possam ser executados ao mesmo tempo.

Uma vantagem importante do processamento paralelo é que em presença de uma falha em um dos processadores, os demais ainda podem continuar trabalhando, mesmo que isso cause um menor desempenho para o sistema.

Computadores maciçamente paralelos se tornaram um meio eficaz para se obter um grande poder computacional, necessário para atender às demandas de aplicações que requerem muitos cálculos.

Nos últimos anos houve um aumento de atividades em diferentes aspectos da produção de *software* para computadores maciçamente paralelos. Novos paradigmas de programação foram criados, bibliotecas para programação paralela foram desenvolvidas. Essas atividades têm convencido muitos pesquisadores a migrarem das arquiteturas seqüenciais para as paralelas [18].

unesp

3.1 Métodos de Computação Paralela

Um projeto de computação paralela começa com a escolha de modelos ou métodos de paralelismo que podem ser usados. Os modelos de computação paralela podem ser bem diferentes. Existem diferenças em sua flexibilidade, mecanismos de interação de tarefas, granularidade de tarefas, suporte para localidade, escalabilidade (verifica até onde o sistema pode crescer sem trazer prejuízos), e modularidade. Esta última reduz a complexidade do programa pelo desenvolvimento de vários componentes separadamente. A seguir são apresentados os modelos principais.

Modelo *tarefa/canal*: é freqüentemente usado para descrever algoritmos. Um processo consiste de um conjunto de tarefas (representadas por círculos) e conectadas por canais (setas), como representado na figura 3.1. Uma tarefa contém um programa e memória local, define ainda um conjunto de portas pelas quais se dá o interfaceamento para o seu ambiente. Um canal é uma fila de mensagem na qual um remetente pode colocar mensagens e da qual um receptor pode remover mensagens, "bloqueando" se estas não estão disponíveis [21].

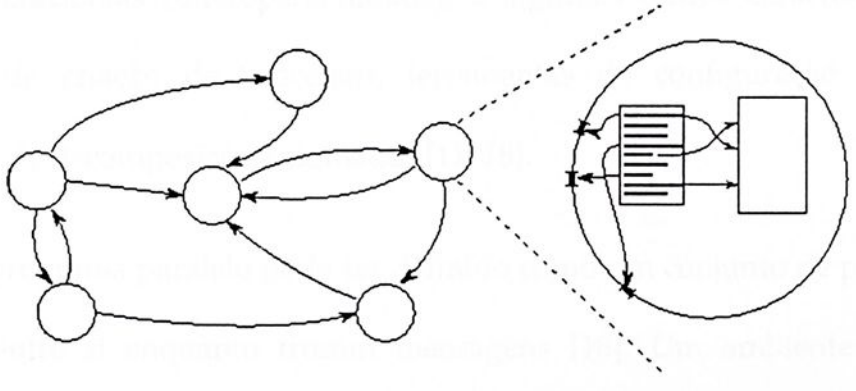


Figura 3.1: A figura mostra um instante do estado de um processo e o detalhe de uma tarefa simples. tarefas são representadas por círculos e canais por setas.

unesp

Ambiente de *troca-de-mensagens*: é provavelmente o mais utilizado usado na programação paralela. Cada tarefa é identificada por um único nome, e as tarefas interagem pelo envio e recebimento de mensagens; as quais transitam entre os processadores [21].

Um dos fatores que estimula a utilização do ambiente de *troca-de-mensagens* é o fato de poder usufruir dos equipamentos existentes para a construção de um ambiente paralelo. Não é necessariamente preciso comprar uma máquina que tenha uma estrutura paralela inerente.

3.2 Ambiente de Troca-de-Mensagens

Com o termo de sistemas de *troca-de-mensagens* identifica-se um ambiente de comunicação que é baseado no uso de primitivas dedicadas à troca de mensagens. Além dessas primitivas de comunicação básicas, esse ambiente é normalmente caracterizado por modelos de programação que são suportados por diferentes sistemas operacionais (interoperabilidade), e algumas outras características como facilidades de criação de processos, ferramentas de configuração de redes, e ferramentas de decomposição automática [11, 18].

Um programa paralelo pode ser definido como um conjunto de processos que interagem entre si enquanto trocam mensagens [18]. Um ambiente de *troca-de-mensagens* é um conjunto de funções e sub-rotinas que fornece meios para a divisão de uma aplicação para ser executada paralelamente. Os dados são divididos e passados para outros processadores como mensagens (fig. 2.2).

unesp

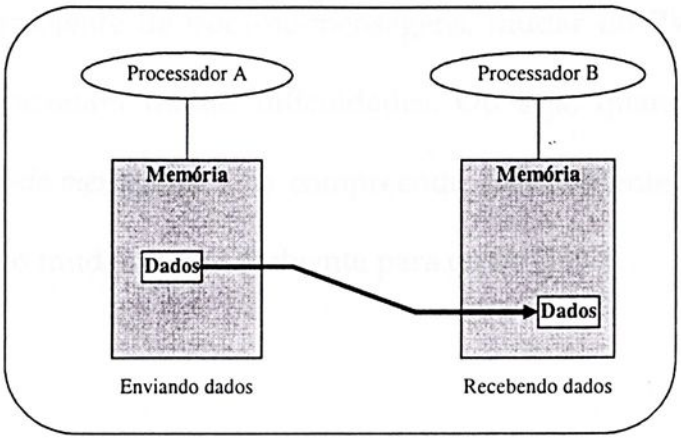


Figura 3.2: Enviando dados da tarefa 'A' para a 'B'.

Os processadores recebem as mensagens e as desempacotam, realizam alguma tarefa sobre os dados recebidos, e enviam os resultados de volta ao processador de origem, ou para outros processadores do sistema [11].

Normalmente, o ambiente de troca-de-mensagens permanece o meio mais seguro para se obter alto desempenho para a maioria de aplicações distribuídas. Esta flexibilidade permite paralelizar vários tipos de aplicações (cliente-servidor, dados-paralelos, sistemas de tempo-real). Tal ambiente é ainda portável e interoperacional [21], porque ele pode ser executado em diferentes tipos de sistemas operacionais, desde que haja o arquivo executável do programa a ser processado. Portável porque pode trabalhar em diferentes tipos de arquiteturas, por exemplo, uma máquina pode ter um processador Intel da família Pentium, outra pode possuir um processador AMD Atlon e assim por diante.

Dois ambientes conhecidos de *troca-de-mensagens* são o *Parallel Virtual Machine* (PVM) e o *Message Passing Interface* (MPI). Muitas das características mais importantes estão disponíveis em ambos. Uma vez conhecidos os detalhes para se

BCMS - FEIS - UNESP

programar num ambiente de troca-de-mensagens, mudar do PVM para o MPI, ou vice-versa, não acarretam muitas dificuldades. Ou seja, quando se entendem os conceitos de *troca-de-mensagens* e se compreende corretamente sua aplicação, não causa muito esforço mudar de um ambiente para outro [11].

3.3 *Parallel Virtual Machine* (PVM)

O PVM é uma biblioteca de funções de comunicação para troca de mensagens permitindo assim o processamento paralelo. Esta permite que os cálculos sejam divididos e distribuídos. Dessa maneira, quando distribuídos em várias máquinas, serão executados paralelamente, quando distribuídos numa única máquina, serão executados concorrentemente.

Uma parte importante do suporte lógico é dedicada à gestão de processos, das tarefas e de sinais na máquina virtual. O PVM permite que uma coleção heterogênea de computadores esteja interconectada por diversos tipos de rede. Por esse motivo, o conjunto das máquinas pode ser visto como se fosse uma máquina virtual paralela.

O fato de ser interoperacional e portável, permite que seus usuários explorem as máquinas já existentes, dessa maneira pode-se minimizar os custos relativos à aquisição de novos equipamentos [21]. Ou seja, o PVM permite que se utilizem diversos tipos de máquinas, de diferentes arquiteturas integrando-as de tal maneira que possam ser utilizadas para integrar a máquina virtual [11].

unesp

A idéia por trás do PVM é reunir um conjunto variado de recursos computacionais utilizando redes para formar uma "máquina virtual". O trabalho em PVM começou no início dos anos 90 no *Oak Ridge National Laboratories* e teve muito sucesso durante um certo tempo entre os cientistas de computação.

A partir da versão 3 do PVM, a máquina virtual pode consistir de processadores simples, multiprocessadores com memória compartilhada e multiprocessadores escalonáveis.

O PVM é uma ferramenta amplamente utilizada porque proporciona portabilidade. Quando um código de programação for bem elaborado para um ambiente de troca-de-mensagens, ele tende a ser executado com sucesso em vários tipos de arquiteturas [11].

3.4 *Message-Passing Interface* (MPI)

O MPI foi desenvolvido por um grupo de fabricantes de computadores, desenvolvedores de programas e cientistas de computação. A idéia era propor um ambiente que fosse adicionar novas propriedades aos ambientes de troca-de-mensagens já existentes, aproveitando-se as características disponíveis de tais ambientes.

unesp

3.4..1 Características do MPI

Os modelos de programação de *troca-de-mensagens* são não-determinísticos: a ordem de chegada das mensagens enviadas por dois processos, A e B, para um terceiro processo, C, não é definida. Entretanto, o MPI garante que duas mensagens enviadas de um processo, A, para outro processo, B, chegarão na ordem enviada. É responsabilidade do programador assegurar que a ordem seja determinística quando for necessário.

O MPI permite vários estilos de comunicação, incluindo bloqueante e não-bloqueante. A comunicação bloqueante é aquela em que uma tarefa é interrompida para esperar dados que dependem de outro processo, há sincronismo. Já na não-bloqueante os processos podem continuar a execução mesmo que os dados não estejam disponíveis, não há sincronismo. O não-bloqueamento permite a sobreposição da comunicação e cálculo. O MPI pode suportar um modelo no qual não exista espaço de memória disponível para registradores, onde os dados devam ser copiados diretamente do espaço de endereço dos processos de envio para o espaço de memória dos processos recebidos.

3.5 Comparando MPI e PVM

O MPI tem um conjunto muito mais rico de funções de comunicação e conseqüentemente apresenta um melhor desempenho de comunicação, o PVM tem a vantagem da heterogeneidade e maior tolerância a falhas. O MPI também suporta melhor a portabilidade, enquanto o PVM suporta melhor a interoperabilidade, ou

seja, a máquina virtual pode possuir diversas máquinas com sistemas operacionais diferentes [21].

Na tabela 2.1 são mostradas as principais características do PVM e do MPI. Destas características pode-se ressaltar que o PVM possibilita o uso do conceito de máquinas virtuais paralelas, enquanto o MPI, tem mecanismos de comunicação melhor desenvolvidos.

Tabela 2.1: Comparação entre PVM e MPI.

PVM	MPI
Máquina Virtual	
Permite que uma coleção de recursos computacionais (heterogêneos) possam ser vistos com uma única máquina paralela; Controle de processos: capacidade de iniciar, interromper e controlar processos, em tempo de execução; Topologia: exige que o programador arranje manualmente tarefas em grupos, segundo a organização desejada.	Não possui o conceito da máquina virtual paralela, centralizando-se no conceito de *message-passing;* Controle de processos restrito; Topologia: Ainda que não possua o conceito de PVM, o MPI provê um alto nível de abstração, em termos de topologia.
Portabilidade versus Interoperabilidade	
Além da portabilidade (como o MPI), os programas permitem nível de interoperabilidade, permitindo execuções em arquiteturas diferentes.	Programas escritos para uma arquitetura podem ser compilados para uma outra arquitetura.
Tolerância a falhas	
Existem esquemas básicos de notificação de falhas, para alguns casos. Permite flexibilidade, de forma que, ainda em certas situações onde não existe resposta de uma máquina, uma aplicação receba resultados das outras.	As versões iniciais, não incluíam esquema de tolerância, a partir das ultimas especificações, esquemas similares ao de PVM são providos.
Segurança na Comunicação	
Possui um *daemon* mantendo a máquina virtual. Processos podem se comunicar com grupos, permitindo recuperação de falhas. As novas versões pretendem fazer uso de Comunicadores, como os do MPI.	O conceito de *communicators*, permite prover um elevado nível de segurança na comunicação, permitindo diferenciar mensagens de bibliotecas de mensagens de usuários.

Fonte: Manual de MPI – CENAPAD (Centro Nacional de Processamento de Alto Desempenho)

Recentemente a Universidade do Tennesse e o *Oak Ridge National Laboratories* começaram investigar a possibilidade de fundir as características do MPI e do PVM

BCo/IS - Feis - UNESP

[21]. O projeto é chamado PVMPI e envolve a criação de um paradigma de programação que permita o acesso às características de máquina virtual do PVM e as características de comunicação do MPI.

Apesar de muitos dos envolvidos no desenvolvimento do PVM terem participado também no desenvolvimento do MPI, o MPI não é simplesmente uma seqüência do PVM. O PVM foi desenvolvido em um laboratório de pesquisa universitário e evoluiu conforme novas características eram necessárias. O MPI é capaz de fornecer um número maior de funções em relação ao PVM e, apesar disso, manter um nível de simplicidade comparável ao PVM. O MPI não especifica os detalhes do manejo do sistema como no PVM; o MPI não especifica como uma máquina virtual é criada, operada e usada.

4 DISCUSSÃO DOS MODELOS UTILIZADOS NESTE TRABALHO

Neste capítulo são discutidos os principais aspectos da metodologia que foi desenvolvida por *Marranghello* e como ela foi implementada para poder se fazer o mapeamento. Algumas funções padrão de AG tiveram que ser modificadas para resolver o problema deste trabalho, já que as respostas do modelo tradicional não são adequadas neste caso.

4.1 Descrição do Trabalho

Este trabalho faz parte de um projeto mais amplo cuja finalidade é buscar um método alternativo com melhores resultados e menores custos para síntese de circuitos eletrônicos. Tem como objetivo o melhoramento dos módulos de mapeamento e roteamento de circuito, a ser realizado sobre a arquitetura ABACUS. A principal contribuição para a metodologia estudada nesse trabalho é a paralelização do algoritmo genético, responsável pela busca por melhores soluções no mapeamento de um dado circuito na arquitetura.

Em síntese, o trabalho visa controlar a distribuição de componentes de um dado circuito num arranjo tridimensional de processadores. Tal arquitetura é reconfigurável, e os processadores de tal arquitetura podem ser escolhidos para assumir o papel de um componente de circuito a ser simulado. Dessa maneira, um dado processador representaria um resistor ou um capacitor ou qualquer outro componente que faça parte do circuito a ser mapeado.

É importante salientar que um processador representará apenas um tipo de elemento para cada mapeamento. Ele poderá ainda não assumir nenhum comportamento, caso não seja escolhido pelo processador hospedeiro.

Após o mapeamento do circuito no arranjo de processadores tem início o roteamento. Este último é implementado através de um algoritmo heurístico e de tal modo que um dado mapeamento sempre terá o mesmo roteamento. Esse fato tem a vantagem de não precisar armazenar em memória o roteamento feito para cada mapeamento, já que o roteamento sempre seguirá os mesmos caminhos.

A cada mapeamento e roteamento a função de custo avalia a qualidade da distribuição feita sobre o arranjo. Essa função avalia o custo referente ao número de chaves utilizadas para que os elementos estejam conectados. Quanto menor o número de chaves utilizadas, melhor é a avaliação para um dado mapeamento.

A princípio é possível criar aleatoriamente indivíduos (ou mapeamentos) para fazer parte de uma população inicial e, através de um algoritmo de roteamento, pode-se verificar a factibilidade de cada indivíduo.

Após a geração de uma população inicial, é realizada uma interação entre elas. Esta, através dos operadores de cruzamento e mutação, busca soluções melhores do que as já encontradas visando reduzir os custos, de maneira a encontrar o melhor mapeamento no arranjo. Lembrando que os custos nesse caso se referem à utilização das chaves que interligam os processadores. Cada chave tem um custo sendo que a mais custosa delas é a chave hexagonal, que realiza a interconexão entre as camadas da matriz de processadores.

4.2 Metodologia

4.2.1 Máquina Paralela

Como já visto, para que se possa obter a paralelização devem-se utilizar recursos de ambientes que permitam tal procedimento. A paralelização neste caso será implementada com a utilização de recursos do PVM.

O PVM possui bom desempenho em diferentes tipos de arquiteturas e pode apresentar resultados satisfatórios utilizando uma ou mais máquinas.

4.2.2 Modelo de Algoritmo Genético

Neste trabalho é utilizado o modelo de AGs paralelos denominado "ilha" [3]. No início é gerada uma população com uma quantidade de indivíduos representando soluções para o problema. Sendo que estas podem ou não funcionar, já que elas foram criadas aleatoriamente, cada vez que uma nova solução é criada verifica-se se é factível ou não. Se for considerada factível, então a solução é inserida na população, caso contrário é descartada e uma nova solução é gerada. Isso ocorre até que se tenha alcançado o número especificado para o total de indivíduos da população. No caso específico deste trabalho, serão criados vários indivíduos que representarão, individualmente, mapeamentos de um determinado circuito sobre a arquitetura (ABACUS); esse procedimento será explicado mais adiante.

A partir da população inicial é feita a distribuição dos indivíduos que a compõem em outras subpopulações (que são recebidas nos computadores *escravos*).

unesp

Elas são submetidas a mutações e cruzamentos e depois são selecionados os melhores indivíduos através da função de aptidão (que nesse trabalho é verificada pelo roteamento). Após um número determinado de iterações, o melhor indivíduo de cada subpopulação é selecionado e retornado ao computador mestre.

O modelo funciona como descrito a seguir e ilustrado na figura 4.1:

1. Uma população inicial é gerada com genótipos aleatoriamente e de número fixo no computador *mestre;*
2. A população inicial é dividida em diversas subpopulações nos computadores *escravos;*
3. São realizados cruzamentos e mutações em cada uma das subpopulações.
4. Os melhores indivíduos são selecionados através da função de aptidão.
5. O melhor de todos os indivíduos de cada subpopulação é selecionado e retornado ao computador mestre.
6. É escolhido o indivíduo com o melhor resultado.

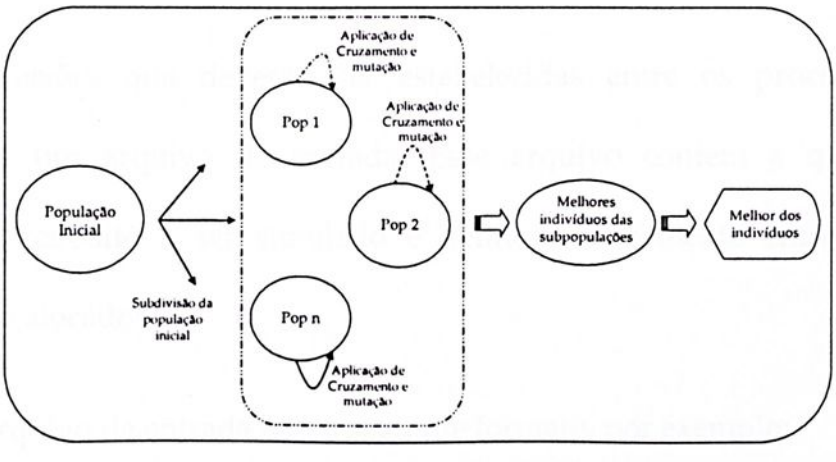


Figura 4.1: Esquema do algoritmo genético paralelo, modelo "*ilha*".

4.3 Desenvolvimento

A arquitetura de processadores foi representada em uma matriz com chaves quadradas, pentagonais e processadores. As chaves hexagonais foram representadas em um vetor de apontadores, de forma que, as chaves pentagonais em torno de uma determinada chave hexagonal sempre apontarão para a chave hexagonal pertinente. Esta mesma sempre apontará para as chaves hexagonais que estão à sua volta. Pois, para cada chave hexagonal do vetor de chaves hexagonais, estará definido o endereço das chaves pentagonais com as quais ela faz conexão. Para cada camada o número de chaves hexagonais é igual à raiz quadrada do número de elementos por camada mais 1, ou seja:

$$N = \left(\sqrt{z} + 1\right)^2,$$

Onde N é o número de chaves hexagonais e z é o número total de processadores por camada. Dessa maneira, para se ter o número total de elementos no vetor de chaves hexagonais, basta multiplicar z pelo número de camadas. Dessa maneira é possível obter-se o número de todas as chaves hexagonais representadas para a dada arquitetura.

As conexões que deverão ser estabelecidas entre os processadores são definidas em um arquivo de entrada. Esse arquivo conterá a quantidade de elementos do circuito a ser simulado e define o roteamento entre os diversos processadores alocados.

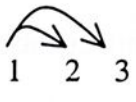
Esse arquivo de entrada tem o seguinte formato, por exemplo:

unesp

```
6
1 2 3
2 4 6
3 5 7
4 1 2
5 6 1
```

A primeira linha deste arquivo de entrada representa o número de elementos que terá o circuito a ser simulado, neste caso seriam seis elementos. Com esse valor é possível mapear aleatoriamente os elementos na arquitetura que foi definida com número de processadores e camada.

As demais linhas representam as ligações que cada elemento deve fazer. A cada linha, o primeiro caractere representa o elemento de origem das conexões, ou seja, no caso da linha dois do arquivo de entrada, o primeiro elemento é o 1 e ele deverá fazer conexões com o elemento 2 e o elemento 3. Numa mesma linha, sempre as conexões serão do primeiro elemento para os demais:



Como foi dito, com os dados de entrada é possível gerar os indivíduos que farão parte da população inicial. Eles são submetidos aos operadores de cruzamento e mutação na busca de soluções melhores para o problema.

A geração dos indivíduos da população é feita da seguinte forma, tendo o número de elementos do circuito a ser simulado, que é dado no arquivo de entrada, é possível saber quantos processadores têm que ser mapeados para representar um único elemento. Cada elemento é alocado uma única vez sobre a arquitetura e uma

vez um processador alocado, não pode assumir um outro valor. O processador onde o elemento será alocado é sorteado.

Esses elementos são representados por números. Foi definida uma estrutura para os processadores e, no campo especificado, um processador (P_n) assume o valor do elemento que está sendo alocado.

Uma vez mapeado, o processador é "marcado" com uma *flag*, que passa a receber valor 1. Dessa maneira somente os processadores que possuem *flags* iguais a zero poderão ser alocados. Esse procedimento mantém a integridade de cada indivíduo.

Esse processo de alocação de elementos é repetido até que todos os elementos tenham sido mapeados sobre a arquitetura. No exemplo dado no arquivo de entrada, esse processo teria que se repetir 6 vezes, a não ser que encontrasse algum processador já alocado, aumentando o número de repetições para encontrar outros processadores não alocados.

Esse processo de mapeamento é repetido até que se tenha alcançado o número total de indivíduos para a população inicial. Esse número é definido previamente dependendo da situação de cada caso a ser simulado na estrutura.

É importante notar que, dentro dessa população inicial provavelmente encontram-se indivíduos com melhor e pior qualidade, já que todos foram criados aleatoriamente. Não há um critério para a criação desses indivíduos. Os elementos são alocados aleatoriamente sobre a arquitetura. Mas dentro da população inicial

todos os indivíduos serão factíveis. Poderão possuir má qualidade, mas serão todos factíveis.

São considerados factíveis, neste caso, os mapeamentos que permitem a interconexão entre todos os componentes alocados sobre a arquitetura.

O que determina se os indivíduos são factíveis ou não é o algoritmo de roteamento do circuito. Esse algoritmo percorre a arquitetura de modo a fazer as ligações entre os processadores. Cada vez que uma chave é utilizada, ela é marcada e já não poderá ser mais utilizada no mapeamento. Isso quer dizer que os caminhos não podem se cruzar.

4.4 O algoritmo de roteamento

Como já comentado no item anterior, para que as ligações entre os processadores sejam realizadas, o algoritmo utiliza o arquivo de entrada, onde estão especificadas as ligações entre os componentes que compõem o circuito a ser mapeado. Como nesse momento os componentes já estão alocados, é possível saber sobre qual processador ele (componente) se encontra e, dessa maneira, é possível realizar o roteamento.

Para tal procedimento o algoritmo observa o elemento de origem e o de destino e localiza em que posição ele se encontra. Essa análise é feita pela verificação dos índices da posição em que se encontram, um e outro. A figura 4.2 dá uma visão

unesp

espacial no eixo *XYZ* de uma arquitetura com 9 processadores por camada e três camadas.

As direções que o algoritmo de roteamento pode seguir são representadas a seguir:

$x_i < x_d$ - direção Oeste → Leste;
$x_i > x_d$ - direção Leste → Oeste;
$y_i < y_d$ - direção Sul → Norte;
$y_i > y_d$ - direção Norte → Sul.

Caso o componente destino esteja à leste ($x_d > x_i$) em relação ao elemento de origem, ele tentará ir para a chave à leste, se esta já estiver alocada por uma outra ligação entre outros dois elementos, então ele tentará ir para a direção sul, se esta também estiver alocada, ele tentará ir para o norte, e em último caso tentará ir para o oeste. Caso todas as chaves estejam ocupadas, retorna um valor "infactível", já que não foi possível estabelecer a conexão.

Existe ainda a possibilidade do elemento destino não estar na mesma camada. Quando isso acontece, o processo de roteamento acessa as chaves hexagonais que podem fazer conexões entre camadas. Dessa maneira, durante o processo de roteamento, assim que se encontra uma chave pentagonal no deslocamento, é feita uma chamada para a chave hexagonal correspondente. Se esta última não estiver alocada, é feita uma chamada para a chave adjacente na camada, para cima (*up*) ou para baixo (*down*) dependendo da camada em que se encontra o elemento destino. Caso a chave hexagonal na camada adjacente já esteja alocada, então retorna-se à chave hexagonal que originou a chamada e busca-se outra chave hexagonal que possibilite a mudança de nível.

unesp

É importante notar que as chaves hexagonais não são utilizadas somente quando os processadores de origem e destino não se encontram na mesma camada. Elas podem ser utilizadas também para fazer conexões dentro de uma mesma camada.

Os caminhos entre os processadores não podem se cruzar, uma vez que uma chave tenha sido utilizada para estabelecer uma rota entre dois processadores, ela não pode ser reutilizada.

Foi possível observar que pelo fato do algoritmo ser heurístico, algumas vezes ele acaba fazendo um caminho maior que o necessário, principalmente quando ele encontra chaves já utilizadas em outras conexões no seu caminho. Isso pode forçá-lo a fazer um desvio bem maior. Esse tipo de situação pode ser superada através da função de seleção que, neste caso específico, promoverá indivíduos que utilizem o menor número de chaves para as conexões.

A figura 4.2 mostra um caso em que os componentes (E1 e E2) não foram alocados de maneira alinhada (linha e coluna).

unesp

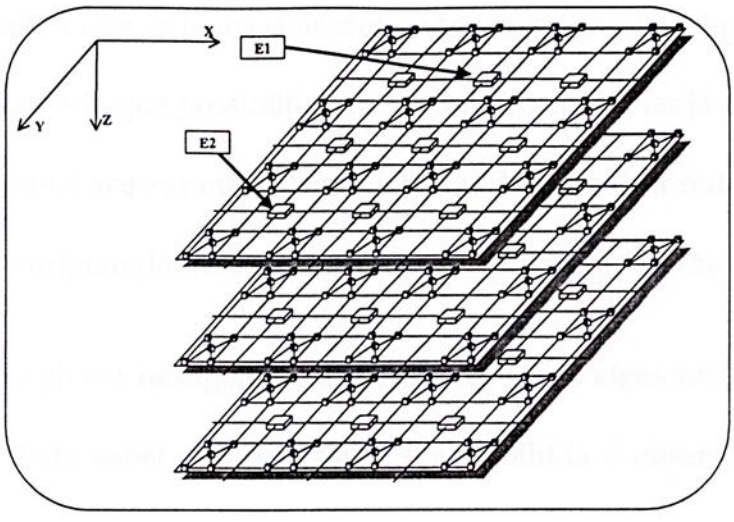


Figura 4.2: Distância no eixo Y maior que distância no eixo X, nesse caso prioriza-se o deslocamento no eixo Y.

E1 = (x=5, y=3, z=0)

E2 = (x=2, y=8, z=0)

Observa-se que não há alinhamento nem no eixo x, nem no eixo y.

Neste caso, a direção a ser priorizada é a que tiver maior diferença. No caso citado acima, ter-se-ia:

$$\Delta E_x = |E(x_d - x_i)| = 1$$

$$\Delta E_y = |E(y_d - y_i)| = 2$$

onde y_d e y_i são os índices dos elementos destino e inicial, respectivamente, no eixo y; x_d e x_i são os índices dos mesmos elementos, destino e inicial, no eixo x. ΔE representa a diferença entre a origem e o destino. Vê-se que a distância no eixo y é maior que no eixo x, portanto a primeira direção a ser procurada será no eixo y. Quando a distância for igual, então se priorizará o deslocamento no eixo x.

Imagine ainda que se tenha dois elementos numa mesma camada e que todas as chaves nesta camada que possibilitariam a conexão entre eles já estejam ocupadas. Nesse caso, é possível acessar outra camada tentando encontrar outra rota e voltando para a camada de origem de modo a se tentar estabelecer a conexão.

Estando na chave hexagonal e supondo-se que o elemento destino esteja na mesma camada. Para saber qual direção a ser escolhida é observada a posição do elemento destino.

Esses procedimentos são realizados a cada passo durante o deslocamento, verificando-se os caminhos que poderão ser percorridos.

O algoritmo também é recursivo, ou seja, durante o processo de roteamento, caso um caminho se "choque" com uma chave que já tenha sido utilizada por outra conexão, então ele retorna à chave precedente e tenta "caminhar" por outra rota, se não for possível ele ainda pode retornar para a precedente a esta última e assim por diante, podendo retornar até à sua origem e determina a rota infactível para a tentativa de conexão. O caminho somente será confirmado quando os dois elementos (origem e destino) estiverem conectados completamente através das chaves.

Além disso, quando o algoritmo de roteamento chega em uma chave onde não é mais possível continuar no caminho, ele a marca através de uma *flag* específica para que não retorne novamente por aquele caminho. Uma vez marcada então, ele retorna à chave precedente e procura outro caminho.

unesp

O roteamento define quantas chaves são necessárias para determinar as conexões através dos processadores. Através desse número é possível ter o valor de qualidade do indivíduo.

Antes de começar o roteamento entre dois componentes num dado mapeamento, verifica-se a disponibilidade das chaves que dão acesso ao processador destino. Se essas chaves estiverem ocupadas, não se realiza o algoritmo de roteamento e o indivíduo é considerado infactível, já que a conexão entre os dois processadores em questão não pode ser realizada.

4.5 Operadores de Cruzamento e Mutação

4.5.1 Operador de Cruzamento

Tendo todos os indivíduos sido criados aleatoriamente, pode-se dar início à utilização dos operadores de cruzamento e mutação.

No início do processo de cruzamento toda a população é combinada dois a dois, formando pares de genitores. Para isso, convém sempre formar uma população inicial com número par de indivíduos. Os pares são combinados aleatoriamente entre si. Uma vez formados, é verificada a probabilidade de ocorrer o cruzamento entre eles. Essa probabilidade - como já foi comentado - é variável, dependendo do problema. Caso o par não seja selecionado para o cruzamento, eles são novamente inseridos na população sem nenhuma alteração. Isso dá a possibilidade de propagar suas características.

unesp

Os operadores de cruzamento e mutação para a resolução desse tipo de problema são diferentes daqueles para o modo padrão de AGs. Isso porque o modo padrão poderia gerar indivíduos com genes (componentes) duplicados ou ausentes, o que não caracterizaria um indivíduo, pois isto corresponde a uma alteração do circuito a ser simulado.

Para resolver esse problema foi utilizado um método específico para que o cruzamento e a mutação se tornassem possíveis de serem realizados. O método para o cruzamento chama-se Cruzamento Baseado em Ordenamento Uniforme (*Uniform Order-Based Crossover*) semelhante ao PMX [14] descrito no capítulo 2. Ele funciona da seguinte forma:

Dois elementos são selecionados para ser aplicado o operador de cruzamento.

Genitor 1	1	2	3	4	5	6	7	8
Genitor 2	8	7	6	5	4	3	2	1

Um padrão binário é gerado aleatoriamente para cada par de indivíduos.

Padrão : 0 1 1 0 1 1 0 0

Esse padrão é gerado tendo por base o número de genes de um indivíduo, ou seja, no exemplo acima o padrão deveria gerar um vetor de 8 posições com valores aleatórios de 0 e 1.

São então definidos dois vetores (filhos) que receberão as combinações de seus genitores de acordo com o padrão binário citado acima.

Para a combinação dos genes, é criado um laço com o mesmo número de genes dos genitores para que se estabeleça a comparação do padrão binário criado. Por exemplo, quando o número do padrão binário for igual a "1" atribui-se os valores referentes ao genitor "1" no filho "1". Quando o número do padrão binário for igual a "0", atribui-se os valores referentes ao genitor "2" no filho "2". Em ambos os casos os valores são atribuídos nas mesmas posições referentes aos pais.

Para o primeiro descendente, os nós do genitor 1 correspondentes às posições do padrão com valor 1 são mantidos, isto é:

-	2	3	-	5	6	-	-

As falhas (-) são preenchidas com os nós restantes, seguindo a ordem em que se encontram no genitor 2, isto é (8, 7, 4, 1). Então, o primeiro descendente seria o seguinte:

Descendente

8	2	3	7	5	6	4	1

Analogamente, para o outro descendente, os nós do genitor 2 correspondentes às posições indicadas com 0 no padrão são copiados como estão, isto é:

8	-	-	5	-	-	2	1

Neste caso as falhas (-) são preenchidas pelos nós restantes, ordenadas de acordo com o genitor 1, ou seja (3, 4, 6, 7), que apenas por coincidência é a mesma ordem, resultando em:

Descendente	8	3	4	5	6	7	2	1

4.5.2 Operador de Mutação

Após o processo de cruzamento, cada um dos novos indivíduos criados é submetido ao operador de mutação. O processo de mutação também tem uma certa probabilidade de ocorrer ou não. A mutação no modo usual é feita *bit* a *bit*, ou gene a gene de um dado indivíduo, mudando-o ou não. No presente trabalho esse método usual também pode ocasionar problemas, sendo assim essa probabilidade não é aplicada gene a gene, mas sim ao indivíduo como um todo. Então foi utilizado um processo de mutação denominado "mutação embaralhada" (*Scrambled Mutation*).

Se o indivíduo for selecionado para que ocorra a mutação então um trecho de seus genes é escolhido aleatoriamente para que ocorra um rearranjo também aleatório dos seus genes. O processo de rearranjo baseia-se no mesmo processo de mapeamento dos elementos sobre a arquitetura, só que neste caso o mapeamento da seqüência que foi escolhida é feita somente sobre os processadores referentes aos genes escolhidos.

4.5.3 O Processo de Mutação Embaralhada

Este tipo de mutação consiste na escolha aleatória de um subconjunto de nós de uma cadeia e a sua reordenação, ou seja, considere a cadeia seguinte:

Descendente	1	2	3	4	5	6	7	8

Suponha que seja escolhido o subconjunto {4 5 6 7} como alvo da mutação. Faz-se um embaralhamento resultando, por exemplo, na seqüência {5 7 6 4}. Esta seqüência é então recolocada no lugar da seqüência de origem, resultando na seqüência "pós-mutação":

Descendente após mutação	1	2	3	5	7	6	4	8

Terminadas as duas operações, mutação e cruzamento, é feito então o roteamento, já explicado acima. O roteamento define quantas chaves serão utilizadas para determinar as conexões através dos processadores. Através desse número é possível ter o valor de qualidade do indivíduo. Nesse caso, os indivíduos de maior qualidade ou mais aptos, serão os que tiverem menor custo, o que representará que foram utilizadas poucas chaves para as conexões, significando que seus componentes foram alocados próximos uns aos outros.

Terminado o roteamento e tendo os valores de cada indivíduo, referente ao número de chaves utilizadas para o processamento, é possível determinar os indivíduos que serão excluídos da população. A cada geração é mantido o mesmo número de indivíduos da população inicial, eliminando-se o excesso de indivíduos criados. Para isso se priorizará os indivíduos mais qualificados.

cm 1 2 3 4 5 6 7 unesp 1 12 13 14 15 16 17 18

5 DETALHES DA IMPLEMENTAÇÃO DO PROGRAMA

Neste capítulo apresenta-se a estrutura de dados de algumas funções utilizadas na implementação do programa de alocação e roteamento de elementos de circuitos. É mostrado o algoritmo para a criação da estrutura que possibilitou o mapeamento.

Para se guardar as informações de cada indivíduo, foi definido um vetor de tamanho igual ao número de processadores do circuito. Cada posição representa um dado processador do circuito. Sendo que cada uma dessas posições do vetor aponta para um dado endereço na estrutura da matriz de elementos e processadores.

5.1 A matriz de processadores e elementos:

A figura 1.3 mostra, de maneira resumida, a estrutura da matriz de processadores, os quais são interligados por barramentos. Entre esses barramentos existem chaves, que possibilitam a configuração das ligações entre os processadores. Cada uma dessas chaves, porém, só pode fazer um tipo de conexão por vez. Para cada mapeamento, cada uma dessas chaves, se forem utilizadas, só poderá assumir um único tipo de conexão.

Para a simulação de tal estrutura foi definida uma matriz para representar as chaves quadradas, pentagonais e processadores. Também foi definido um vetor para representar as chaves hexagonais.

A matriz tem forma quadrada, ou seja, mesmo número de linhas e colunas. O formato da matriz não interfere no resultado final da simulação. O fato de ter-se escolhido a forma quadrada, foi para facilitar o desenvolvimento do algoritmo de sua estrutura.

5.2 O algoritmo para a construção da estrutura da matriz

Na figura 5.1 pode-se observar que o número de chaves hexagonais por camada é igual a: $(x + 1)^2$. Com esse valor define-se o tamanho máximo do vetor de chaves hexagonais. O número máximo de chaves hexagonais no vetor é:

$$H = (x+1)^2 \times C$$

Onde, H é o número total de chaves hexagonais; x é o número de processadores e, C é o número de camadas da estrutura.

Com o valor H é possível definir qual será o tamanho do vetor de hexagonais. Cada uma das posições do vetor de chaves hexagonais possui ponteiros para as chaves com as quais ele pode estabelecer conexão.

A quantidade total de processadores por camada sempre deverá ser um número "quadrado". Para se achar o número total de processadores na estrutura deve-se multiplicar a quantidade de processadores por camada pelo número total de camadas.

A ordem da matriz que representa as chaves quadradas, pentagonais e os processadores é obtida pela equação:

$$q = \left(\left(4 \times \sqrt{p}\right) - \left(\sqrt{p} - 2\right)\right)$$

Onde q é a ordem da matriz, p é o número total de processadores por camada.

Lembrando que a ordem desta matriz representa apenas as linhas e colunas onde são representadas as chaves pentagonais, quadradas e processadores. As chaves hexagonais, por não estarem alinhadas aos demais elementos da estrutura, são armazenadas em um vetor de hexagonais para onde "apontam" as chaves pentagonais.

Na figura 5.1, pode-se observar que as chaves hexagonais não estão sendo mais representadas na estrutura. Na realidade elas são representadas em outro vetor. Pode-se observar que cada uma das chaves pentagonais está apontando para uma chave hexagonal. Cada uma dessas chaves pentagonais aponta para uma posição no vetor de hexagonais.

Para a montagem da estrutura da figura 5.1 observou-se a existência de um certo padrão quando se utiliza um número quadrado de processadores (4, 9, 16, 25, etc.).

Ao considerar os índices da matriz da figura 5.3 observa-se que, toda vez que o "resto" da divisão do "índice da linha" pelo número 3 tem como resultado o número 2 (*índice_linha* "mod" *3* = *2*) e também para a coluna (*índice_coluna* "mod" *3* = *2*), a posição deve representar um "processador". Na linguagem C os índices da

BCpIS - FEIS - UNESP

matriz começam com valor igual a "0" (zero). Por exemplo, a posição (5, 5) representa um processador (5 *mod* 3 = 2).

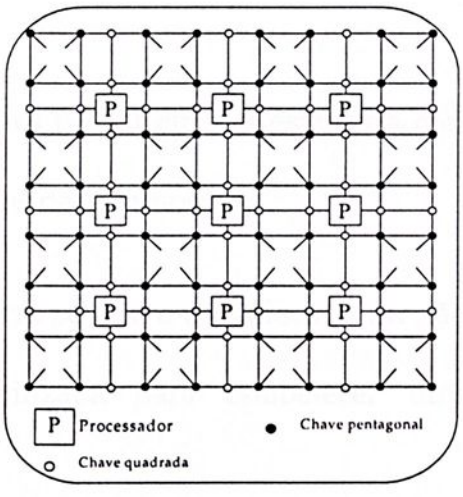


Figura 5.1: Esquema da estrutura da matriz que representa chaves quadradas, pentagonais e processadores. As chaves hexagonais encontram-se representadas num outro vetor

Caso a divisão do "índice da linha" por 3 tenha resultado igual a 2 (*linha* mod *3 = 2)* mas, o resto da divisão do "'índice da coluna" por 3 não resulte num valor igual a 2, (*coluna* mod *3 ≠ 2)* então a posição deve representar uma chave quadrada. Esse procedimento trata as linhas onde se encontram os processadores e as chaves quadradas somente.

Nas linhas onde o resto da divisão do "índice da linha" por 3 for diferente de 2 (*linha* mod *3 ≠ 2*) então significa que a linha só terá chaves pentagonais e quadradas (não terá processadores). Para se fazer tal verificação foi observado que quando o resultado da divisão do "índice da coluna" por 3 for igual a 2 (*coluna* mod *3 = 2*), a chave deve representar uma chave quadrada, caso contrário representa uma chave pentagonal.

No final deste procedimento a estrutura está montada, representando na posição correspondente da matriz, os processadores, chaves quadradas e pentagonais.

Cada uma das chaves possui em sua estrutura componentes para que possa ser representada, dentre eles destacamos:

- Uma *flag* que mostra se chave está alocada ou não, (serve para mostrar se a chave já está sendo utilizada para estabelecer uma conexão entre outros processadores);

- O custo que ela possui, (o custo é utilizado para totalizar a quantidade de chaves que foram utilizadas para se estabelecer todas as conexões entre os elementos mapeados na estrutura. Esse valor mostra a qualidade de cada indivíduo);

- Um campo alfanumérico indicando se a posição se trata de chaves ou processadores. Por exemplo a chave quadrada tem representação "Q", o processador tem representação "*", a chave pentagonal "P".

- Uma *flag* que indica se, durante um dado roteamento entre dois processadores, já foi feita a tentativa de se passar por aquela chave. Isso evita que o algoritmo tente percorrer um caminho que já foi testado sem obter sucesso.

- As chaves pentagonais possuem um componente extra utilizado para informar para qual chave hexagonal ela deverá apontar.

Depois da montagem da estrutura contendo os processadores, chaves quadradas e hexagonais, são acertados os "ponteiros" entre as chaves pentagonais e

unesp

hexagonais, de maneira que cada chave pentagonal possa apontar para a sua chave hexagonal correspondente e vice-versa.

Na figura 5.2 é possível observar a representação gráfica do canto superior (esquerdo) da matriz onde se encontram as posições (0,0), (0,1), (1,0) e (1,1). Estas posições correspondem a chaves pentagonais, portanto elas podem estabelecer conexão com a "primeira" posição do vetor de chaves hexagonais. A posição 0 (zero) no vetor de hexagonais poderá apontar para todas as chaves pentagonais que se encontram à sua volta. A chave pentagonal (0,0) aponta para a posição 0 do vetor de hexagonais, este por sua vez, estabelece a conexão com a chave pentagonal através de seu componente "NO", o qual guarda em sua memória um "apontador" para a posição da chave pentagonal (0,0) (figura 5.2).

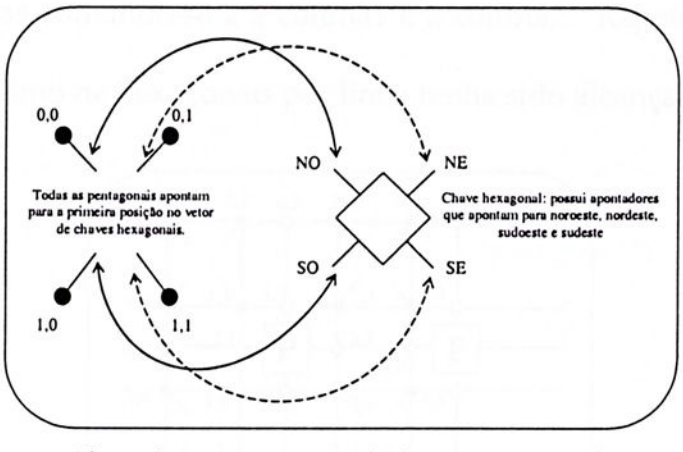


Figura 5.2: Esquema gráfico de como se estabelece o ajuste de ponteiros entre chaves hexagonais e pentagonais.

Verifica-se que todos os processadores pentagonais em torno de uma chave hexagonal apontam sempre para a mesma chave hexagonal, independente de ser NO, NE, SE ou SO. Mas o contrário já não é verdadeiro. A posição NO da primeira

posição no vetor de hexagonais deve, obrigatoriamente, apontar para a posição (0,0) da matriz.

Para o acerto dos ponteiros entre chaves pentagonais e hexagonais foi observado outro padrão; a figura 5.2 mostra alguns índices na matriz. A posição (0,0) e a posição (1,0) fazem conexão com a posição (0) do vetor de hexagonais. No código do programa foram definidas linha1=0 e linha2=1 e, coluna1=0 e coluna2=0. Dessa maneira, pode-se atribuir à primeira posição na matriz (linha1, coluna1); a segunda posição seria (linha2, coluna1); com esses índices é possível acertar os ponteiros para as duas primeiras chaves pentagonais citadas no início deste parágrafo. A terceira posição seria (linha1, coluna2 + 1) e a quarta (linha2, coluna2 + 1). Depois de acertadas estas chaves, continua-se a caminhar no primeiro par de linhas, bastando atualizar as colunas somando-se 2 à coluna1 e à coluna2. Repete esse processo até que o número máximo de hexagonais por linha tenha sido alcançado.

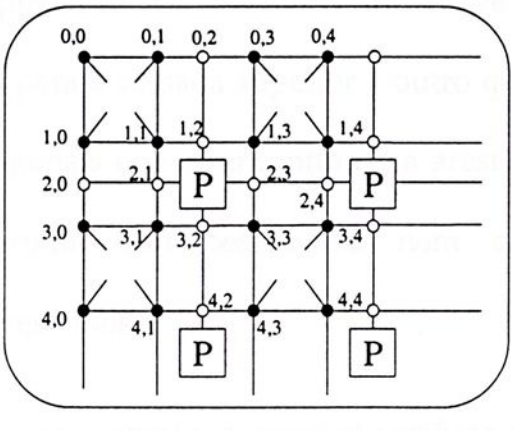


Figura 5.3: Representação de parte dos índices da matriz (canto superior esquerdo).

Quando esse número é alcançado atualiza-se o par de linhas, somando-se 3 à linha1 e à linha2; o valor das coluna1 e coluna2 volta ao valor inicial. Dessa maneira,

o procedimento anterior é repetido para o conjunto de posições (3,0), (4,0), (3,1) e (4,1) e assim sucessivamente.

As chaves hexagonais, como já foi dito, são armazenadas em um vetor. Essas chaves, além de acertar seus "ponteiros" para as chaves pentagonais, devem ainda acertar os ponteiros entre si, já que são elas que se comunicam com chaves hexagonais que se encontram em camadas adjacentes. São as chaves hexagonais que possibilitam a passagem de uma camada à outra.

O número de chaves hexagonais é sempre proporcional ao número de processadores, por exemplo, se uma estrutura possui 2 camadas e 9 processadores por camada, o número de chaves hexagonais será 32 (representado em um único vetor de igual tamanho), sendo 16 em cada uma das camadas.

Para que se possa estabelecer conexão entre as chaves hexagonais foram adicionados dois componentes na estrutura da chave hexagonal. Existe um componente que aponta para a camada superior e outro que aponta para a camada inferior. As chaves hexagonais que se encontram na aresta superior ou inferior do cubo não podem permitir conexões acima nem abaixo, respectivamente, permanecendo com valores nulos.

Na figura 5.4 (a), por exemplo, é possível verificar que a arquitetura possui duas camadas. As setas indicam quais chaves hexagonais deverão fazer ligações entre si. Na figura 5.4 (b) é possível visualizar uma representação do que seria o vetor de processadores. Esse vetor possui todas as posições referentes às chaves hexagonais. Tomando como exemplo a figura 5.4(b), vê-se que o vetor possui 32

cm 1 2 3 4 5 6 7 unesp 1 12 13 14 15 16 17 18

posições. Cada conjunto de 16 posições representaria uma camada. A chave hexagonal "1" aponta para a "17" (que representa a primeira chave da segunda camada)

Tendo as estruturas sido montadas é possível fazer o roteamento entre os componentes.

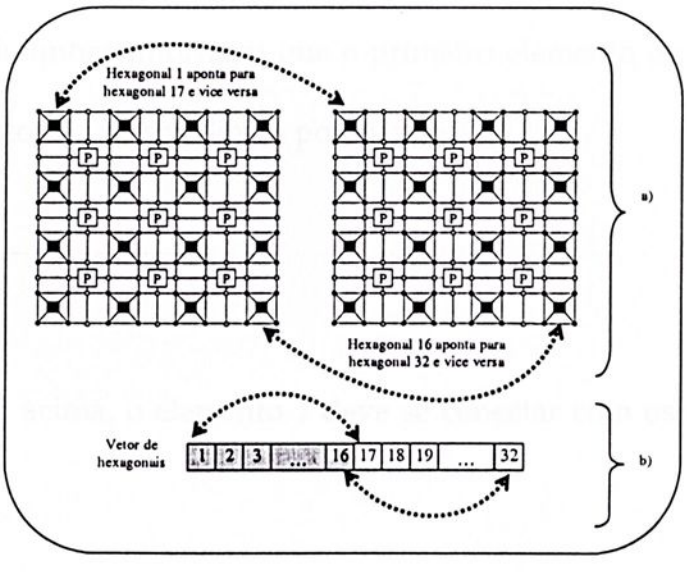


Figura 5.4: Representação da criação do vetor que possibilita o acerto de apontadores entre chaves hexagonais.

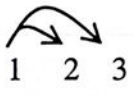
5.3 O mapeamento

Para que aconteça o mapeamento sobre a arquitetura especificada é necessário saber quantos elementos deverão ser mapeados e as conexões entre os componentes. Estes dados são informados através de um arquivo de entrada, onde é especificada, na primeira linha, a quantidade de elementos. Esse valor é necessário para o mapeamento. As demais linhas informam como devem ser feitas as conexões.

Por exemplo, como já visto no capítulo 4, esse arquivo de entrada poderia ter seguinte formato:

```
7
1 2 3
2 4 6
3 5 7
4 1 2
5 6 7
7 3 1
```

A primeira linha corresponde ao total de elementos do circuito que se deseja simular. As demais linhas informam que o primeiro elemento da linha é conectado aos demais elementos da mesma linha, por exemplo:



No exemplo acima, o elemento 1 deve se conectar com os elementos 2 e 3; e assim por diante.

O mapeamento foi elaborado para que ocorra de maneira aleatória e cada mapeamento realizado é considerado um indivíduo.

Quando o programa lê o arquivo de entrada de dados, ele verifica a quantidade de indivíduos que deve possuir o circuito. Sabendo-se o número de elementos, ele começa a sorteá-los sobre os processadores, mapeando-os. Foi criada uma função que sorteia um valor de 0 até o número máximo de processadores da estrutura. Quando o número é sorteado, verifica-se se ele já foi alocado com outro elemento, caso já tenha sido alocado, sorteia-se novamente outro processador até que se encontre um processador livre, alocando o elemento desejado. A alocação é feita

marcando-se uma *flag* com valor "1". Processadores não-alocados possuem os valores de suas *flag's* iguais a "0" (zero).

Quando um processador é sorteado para representar um elemento, além de se marcar sua *flag* de alocação com valor igual a "1", atribui-se-lhe o nome do elemento que está sendo mapeado. O nome do processador serve para identificar com quais outros processadores ele deve ser conectado, possibilitando o roteamento após o mapeamento completo de todos os elementos. Esse nome é "1", "2", "3", ... , "7"; que são os números (nome) que representam os elementos.

5.4 O roteamento

Para o roteamento, foi criada uma função recursiva que escolhe a cada ponto do percurso qual o caminho a seguir, tendo em vista as chaves que se encontram livres. Antes de se iniciar o processo de roteamento entre dois elementos de um indivíduo, verifica-se se as chaves quadradas que circundam o processador destino já foram utilizadas. Caso já tenham sido utilizadas, define-se o indivíduo pois isso caracterizaria uma impossibilidade de conexão entre dois elementos.

Durante o percurso é feito uma chamada à chave que se encontra mais próxima ao destino, à direita, esquerda, etc. Isso é feito através da verificação dos índices da matriz, linha, coluna e camada. Através da comparação dos índices da posição atual com a posição onde se deseja chegar, é possível verificar onde se encontra o processador onde o percurso deve chegar.

Se o processador destino não está na mesma camada, a primeira coisa que ele verifica é se a chave onde ele está é capaz de fazer essa chamada de função para que se possa passar de uma camada para a subseqüente. Dessa maneira, ele faz uma chamada para ir à chave hexagonal mais próxima; caso esteja sobre uma chave pentagonal, ele faz a chamada para a chave hexagonal ligada a esta, se ela não estiver sendo utilizada em outra conexão. Caso a chave hexagonal já esteja sendo utilizada então ele continua na mesma camada, agora verificando a diferença entre linha e coluna para se aproximar da posição em que se encontra o processador na outra camada. Mas a cada mudança de chave o algoritmo verifica se é possível fazer a chave a uma chave hexagonal, para assim poder mudar de camada.

Ao chegar no destino, o algoritmo retorna como resposta que foi possível chegar ao destino e faz os cálculos para calcular a qualidade do indivíduo. Esse cálculo é feito através do somatório das chaves utilizadas para possibilitar a conexão entre os processadores.

Se durante o percurso o algoritmo verifica que todas as chaves que circundam a chave onde ele se encontra já não estão mais disponíveis, ou seja, já foram utilizadas para estabelecer outras conexões. O algoritmo, por ser recursivo, retorna à posição anterior à que se encontra, mas antes marca com uma *flag* que informa que aquele percurso já foi tentado, porém, sem sucesso. Em todas as chaves existe uma *flag* que guarda o valor "1" (representa que já se tentou passar por aquele caminho) ou "0" (zero – representa que aquele caminho ainda não foi utilizado).

unesp

Caso seja possível avançar para a próxima chave à qual o algoritmo fez uma chamada, então marca-se uma outra *flag* responsável para informar que a chave já foi utilizada e não pode ser utilizada novamente, a não ser que o algoritmo retorne sobre o percurso que vinha fazendo para tentar um outro caminho.

Enquanto o algoritmo percorre o caminho entre as posições da matriz ele sempre verifica se essas posições se tratam de chaves ou processadores. Os processadores não podem fazer o papel de chaves, portanto eles não podem ser usados para estabelecer conexões entre outros dois processadores. Dessa maneira, durante o percurso, antes de se mudar de posição, é preciso também verificar se a próxima posição que foi chamada não se trata se um processador. Essa verificação é feita através do componente que demonstra se a posição se trata de uma chave ou um processador.

5.5 Armazenamento dos indivíduos

Para que se guardem os elementos criados, foi definido um vetor de processadores. Esse vetor guarda a posição de todos os processadores da matriz, cada item do vetor "aponta" para a sua posição na matriz.

Para a criação desse vetor de processadores, que representa um indivíduo, verifica-se no arquivo de entrada quantos elementos possui o circuito a ser mapeado. Após essa verificação sorteia um dos processadores da estrutura para representar o primeiro elemento; em seguida, sorteia-se outro processador para representar o segundo elemento, e assim por diante. Até que todos estejam mapeados. Os demais

unesp

processadores permaneceram sem alocação e sem representação para tipo de elemento (fig. 5.5).

Foi definida uma estrutura capaz de armazenar esses indivíduos criados. Assim, cada um desses mapeamentos são guardados em um vetor de indivíduos, após a verificação de factibilidade, formando-se a população inicial.

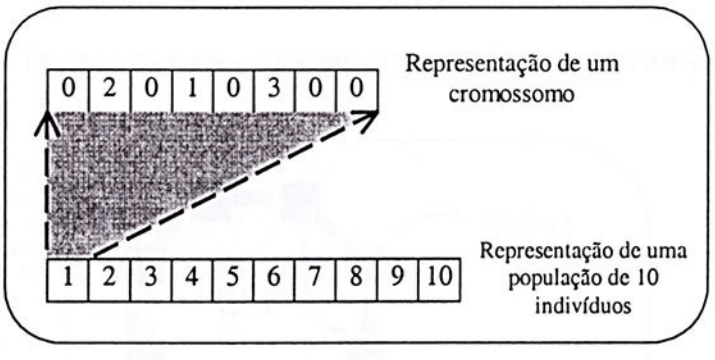


Figura 5.5: Representação de um cromossomo.

5.6 A implementação da função de cruzamento

Foi definida uma estrutura para representar todos os indivíduos criados, para que assim possam ser manipulados posteriormente (l[MAX]).

Nesta estrutura estão representados todos os indivíduos. É então executado um procedimento para que os indivíduos possam ser distribuídos novamente e aleatoriamente dentro de um novo vetor. Esta nova distribuição serve para formar pares de indivíduos. São esses pares que formam os casais para serem submetidos ao cruzamento. Isso garante que, a cada geração, a escolha dos indivíduos para o cruzamento ocorra aleatoriamente.

A estrutura de seleção para esta escolha funciona da seguinte maneira. Imagine que se tenha um círculo representando todos os indivíduos da população, como mostra a figura 5.6. É sorteado um valor que varia de 1 até o valor máximo de indivíduos representados no círculo, o que seria, a princípio, o total de indivíduos no vetor de indivíduos. Esse número sorteado é o total de passos que se deve dar para que um dos indivíduos seja retirado do vetor e colocado em um outro vetor que serve de base para formar os pares a serem submetidos ao cruzamento.

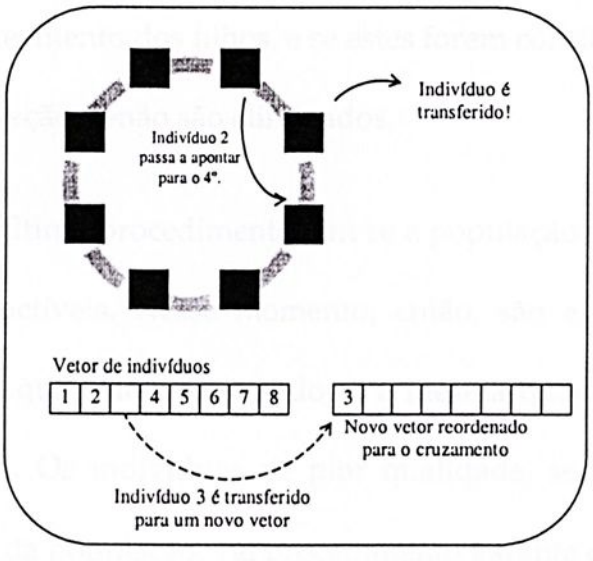


Figura. 5.6: Exemplificação do ordenamento dos indivíduos para serem submetidos ao cruzamento.

O número de passos deverá sempre ter o valor total de indivíduos, para que todos os indivíduos tenham a mesma chance de serem sorteados ou não. Portanto, conforme um indivíduo é eliminado do vetor de indivíduos da população, o número de passos também diminui, permanecendo igual ao total atual de indivíduos ainda a serem distribuídos no novo vetor.

Após este procedimento, tem-se um novo vetor com números inteiros que representam os indivíduos. Com esse vetor é possível então gerar os pares para que sejam submetidos ao cruzamento.

A função de cruzamento gera os filhos e antes de inseri-los na população, cada um deles pode ser submetido ao operador de mutação. Convém lembrar que nem todos são submetidos à mutação, já que existe um taxa percentual para que tal procedimento ocorra ou não. Uma vez todos inseridos junto à população inicial, pode-se começar o roteamento dos filhos, e se estes forem considerados factíveis, eles permanecem na população, senão são eliminados.

No final deste último procedimento tem-se a população inicial adicionada dos filhos considerados factíveis. Nesse momento, então, são escolhidos somente os indivíduos de melhor qualidade. Mantendo-se a mesma quantidade de indivíduos da população original. Os indivíduos de pior qualidade, sejam eles genitores ou filhos, são eliminados da população. Tal procedimento garante que a cada geração os melhores indivíduos sejam mantidos.

5.7 O processo de mutação

Os indivíduos submetidos ao processo de mutação têm uma parte da seqüência de seus genes mudada. A mutação no escopo deste trabalho, como já visto no capítulo precedente, não sofre mutação sobre um único gene, mas sobre um conjunto de genes.

unesp

Para saber qual seqüência de genes de um indivíduo deve ser submetida a mutação, foi criada uma função que sorteia o gene onde ocorrerá o corte. Uma vez escolhido o corte no vetor, tem-se duas partes de um mesmo cromossomo; uma parte antes do corte e outra após, como ilustrado na figura 5.7.

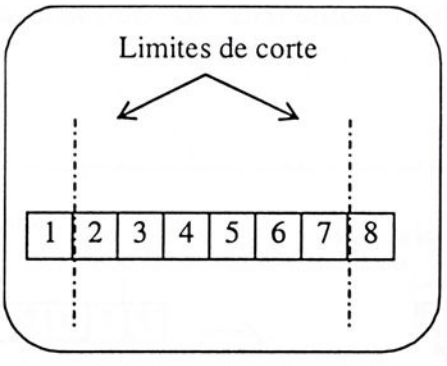


Figura 5.7: Ilustração dos limites onde podem ocorrer o corte no cromossomo para que se possa efetuar a mutação.

Uma vez escolhida a posição no vetor, escolhe-se uma das duas seqüências para sofrer a mutação. Por exemplo, supondo-se que a posição escolhida para o corte tenha sido a posição 4, escolhe-se em seguida, aleatoriamente, a seqüência de 1 até 3 ou a seqüência de 4 até 8.

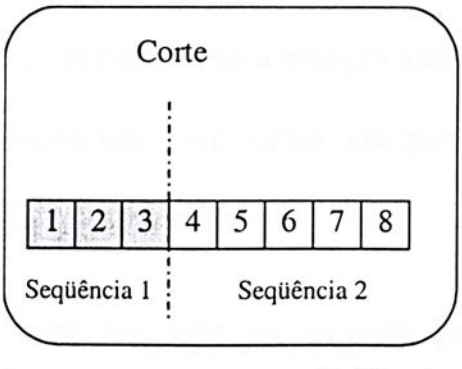


Figura 5.8: Exemplo de corte num cromossomo, dividindo-o em duas seqüências.

Os genes da seqüência sorteada são eliminados do vetor que representa o indivíduo sendo copiados para um vetor auxiliar para que possam ser alocados novamente (fig. 5.9). Esse vetor auxiliar tem a função temporária de guardar os elementos que pertenciam à seqüência que deveria sofrer mutação; ele guarda os valores até que se tenha inserido os elementos novamente na seqüência do cromossomo.

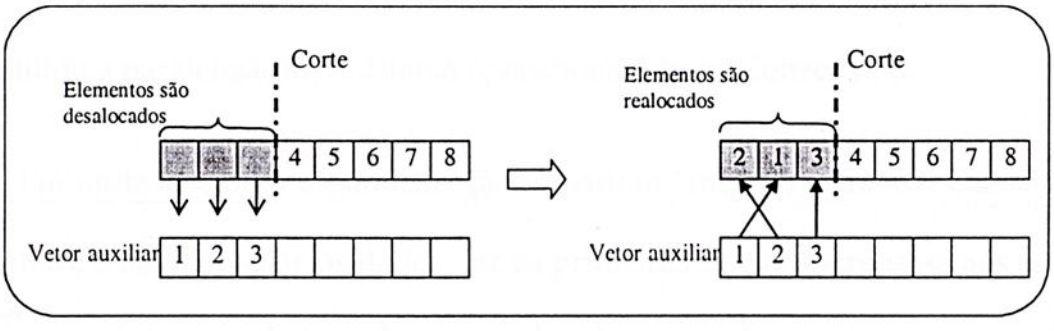


Figura 5.9: Exemplo da estrutura criada para se efetuar a mutação na seqüência selecionada.

Cada elemento que se encontra no vetor auxiliar é realocado na seqüência em que se encontrava, mas desta vez as posições em que eles serão inseridos são sorteadas, permitindo que sejam inseridos em posições diferentes daquela na qual se encontravam. Após o sorteio, verifica-se se a posição ainda se encontra desocupada, se estiver, aloca-se, senão faz-se um novo sorteio até que todos os elementos que se encontravam no vetor auxiliar sejam realocados.

Nota-se que, se o corte for feito na segunda posição e que a seqüência escolhida for a primeira (antes da segunda posição), o processo de mutação, mesmo que ocorra, não ocasionará mudanças, já que só existe uma única posição. Porém, mesmo quando existe mais de uma posição, ainda assim os elementos podem ser

sorteados de tal sorte que voltam ao cromossomo na mesma posição em que se encontravam.

5.8 A implementação paralela

Para a paralelização do algoritmo genético foram utilizadas três máquinas Pentium 750MHz com 128Mb de memória RAM e versão 3.4 do PVM (*software* que possibilitou a paralelização) e sistema operacional Linux, Conectiva 8.

Foi utilizado, para a paralelização, o padrão "mestre/escravo". Ou seja, dois programas, o *mestre* recebe os dados, faz os primeiros cálculos e repassa aos *escravos*. Os *escravos* recebem os dados, efetuam as devidas operações e retornam seu valor ao programa *mestre*.

Algumas mudanças tiveram que ser realizadas para a implementação do AG paralelizado. O principal problema encontrado foi a abertura do arquivo texto onde se encontravam as informações referentes ao circuito (número de elementos e a interconexão de tais elementos).

A princípio, o programa *mestre* mandava as informações aos programas escravos, mas estes últimos não retornavam os valores referentes aos melhores indivíduos encontrados. Após alguns testes foi possível verificar que os *escravos* não conseguiam acessar as informações de leitura no arquivo texto onde se encontravam as informações referentes às interconexões.

unesp

Afim de superar este problema foi preciso modificar a função de roteamento (com relação à leitura dos dados) e criar um vetor para guardar os dados referentes às interconexões entre os elementos.

Esse vetor é passado aos escravos que por sua vez efetuam a leitura para saber como se darão as interconexões. Os dados são lidos no vetor de duas em duas posições. Cada par de posições representa uma interconexão.

Imagine um circuito de 4 elementos e o vetor demonstrativo na figura 5.10, onde estão representadas todas as conexões entre os processadores. Nesse caso pode-se observar que o elemento 1 estará conectado ao elemento 2, depois (no próximo par) o elemento 1 estará conectado ao elemento 4, e assim por diante. No final da leitura do vetor, todos os elementos já estarão roteados.

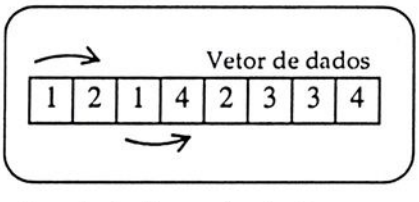


Figura 5.10: Representação do vetor de leitura de dados.

Após receber os dados o programa escravo analisa-os e inicia o processo de cruzamento e mutação como no programa paralelo. Neste caso os escravos possuem apenas uma parte da população para simularem.

Após término da análise de todas as gerações, as informações referentes ao melhor indivíduo são repassadas ao "mestre". Além das informações do melhor indivíduo é passada a qualidade que possuía o melhor indivíduo antes do início das iterações (gerações). Isso permite mostrar a porcentagem da melhora de qualidade

obtida. Também é passada a geração na qual o melhor indivíduo foi criado; o número de gerações total e a porcentagem de melhora da qualidade do indivíduo.

O tempo total de processamento é fornecido quando o programa *mestre* recebe os resultados de todos os *escravos*.

Para comparar o desempenho do AG paralelo em relação ao AG seqüencial foram testadas quantidades iguais de elementos de circuito nos dois programas. Tanto o programa paralelo, como o seqüencial foram submetidos a testes com parâmetros iguais – mesmas condições. No próximo capítulo são apresentados os testes realizados e são comentados os resultados.

6 DESCRIÇÃO DOS TESTES E DISCUSSÃO DOS RESULTADOS

Neste capítulo é comentado como os testes foram realizados e mostram-se os resultados alcançados comparando-se os dois programas, seqüencial e paralelo.

Os testes foram realizados em computadores Pentium III de 750MHz, 128Mb de memória RAM e com sistema operacional Linux (Conectiva 8). Para a simulação paralela foi montada uma máquina virtual - utilizando-se o PVM (versão 3.4) - com 3 computadores (todos com a mesma configuração citada acima). Para a execução do programa seqüencial utilizou-se apenas um dos três computadores.

Foram comparados os resultados de alguns testes realizados nos programas seqüencial e paralelo para observar o desempenho dos dois diante de um mesmo mapeamento. Ou seja, ambos simulavam um mesmo circuito (com mesmo número de componentes e mesma interconexão), sobre um mesmo arranjo (mesmo número de processadores, chaves e camadas). Dessa maneira foi possível comparar os resultados sob as mesmas condições.

Para os testes realizados, a arquitetura da matriz não pôde assumir um valor maior que 200 processadores. A escolha de muitos processadores ou camadas gerava um erro de "estouro de memória" no momento da simulação, devido à limitação deste recurso nos computadores utilizados. Outro fator que gerava "estouro de memória" era uma quantidade muito grande de indivíduos na população, na ordem de 200. Circuitos com mais de 30 componentes geravam estouro de memória,

unesp

aparentemente devido à recursividade existente no módulo de roteamento do programa. Mesmo com essa limitação foi possível verificar resultados que mostraram a eficiência do método.

Para os testes realizados foram utilizados circuitos com diferentes números de componentes, como se pode observar logo abaixo:

CK 5: circuito com 5 componentes;
CK 6: circuito com 6 componentes;
CK 10: circuito com 10 componentes;
CK 13: circuito com 13 componentes;
CK 15: circuito com 15 componentes;
CK 20: circuito com 20 componentes;
CK 30: circuito com 30 componentes.

A tabela 6.1 mostra o tipo de arranjo onde cada um dos circuitos mencionados acima, foi mapeado. Cada um dos circuitos foi testado em mais de uma configuração de arranjo, para prover uma base de comparação.

Arranjos / Circuitos	- 16 proc. - 5 cam.	- 25 proc. - 3 cam.	- 25 proc. - 4 cam.	- 49 proc. - 2 cam.	- 64 proc. - 2 cam.	- 81 proc. - 2 cam.	- 100 proc. - 2 cam.
CK 5							
CK 6							
CK 10							
CK 13							
CK 15							
CK 20							
CK 30							

Tabela 6.1: Distribuição de circuitos testados por arranjo.

As porcentagens de cruzamento e mutação usados foram, respectivamente, 0,7 e 0,1. A população inicial foi gerada com 80 indivíduos para todos os casos. O

número máximo de iterações (gerações) foi mantido em 100. Isso porque foi possível observar que, para os testes realizados, os melhores resultados foram encontrados antes da centésima iteração. Para cada uma das situações, o mapeamento ocorreu no modo seqüencial e paralelo, podendo-se comparar a diferença no tempo de processamento em ambos os modos.

Nos testes realizados verificou-se que a economia de tempo no modo paralelo variou entre 30% e 70%, dependendo da quantidade de componentes que o circuito possuía. Esses resultados foram obtidos através da comparação do tempo gasto no processamento pelo programa paralelo e seqüencial.

A Tab. 6.2 mostra resultados comparativos da execução das versões paralela e seqüenical do AG com taxas de cruzamento de 70% e de mutação de 10%. Cerca de 60% dos melhores resultados, foram encontrados entre as primeiras 70 gerações. Observa-se que para os circuitos CK5, CK6, CK10 e CK13 a versão seqüencial gerou resultados com qualidade final cerca de 5% melhor que a versão paralela. A partir de CK15, a versão paralela passa a gerar resultados com qualidade melhor do que a versão seqüencial. Este ganho na qualidade final mostra uma tendência de ampliação, uma vez que o ganho para CK15 foi de 5% em favor da versão paralela e para CK30 este ganho já chega a 12%.

Circuitos	Tempo seqüencial (s)	Custo final seqüencial	Tempo paralelo (s)	Custo final paralelo
CK5	8	51	3	56
CK6	30	43	19	52
CK10	68	49	40	77
CK13	363	109	188	114
CK15	1449	68	665	64
CK20	633	372	342	353
CK30	1429	419	636	359

Tabela 6.2: Comparação da media dos resultados obtidos.

unesp

A taxa de mutação interfere pouco sobre o resultado final. Uma das razões para isto é porque nem sempre a mutação gera indivíduos de melhor qualidade. Ouro fator é a possibilidade de sua ocorrência ser menor que a do cruzamento. Desta forma, a taxa de mutação estabelecida neste trabalho foi de 10%.

O operador de cruzamento foi de grande importância para a obtenção de melhores indivíduos. Ele permitiu chegar no final das iterações da população com resultados melhores que os iniciais. O cruzamento interfere mais no resultado final devido ao fato de seu número de ocorrência ser superior ao de mutação. Ele também possibilita a combinação de diversas soluções.

Pôde-se observar que, em média, o processo de cruzamento após 80 iterações raramente causava mudanças adicionais, trazendo resultados melhores aos indivíduos. Entretanto, os resultados alcançados e apresentados mostraram possuir boa qualidade, resultando numa redução dos custos. O que realmente confirmou a otimização do resultado final.

Num dos testes, modificou-se a taxa de cruzamento, de 70% para 20%. Foi possível observar que, para a obtenção de resultados melhores, foi necessário um número maior de gerações. Normalmente o resultado após o término do processamento não teve uma melhora significativa, ficando dentro de uma faixa de 20% a 30%.

unesp

Outra verificação dos resultados obtidos com taxas pequenas de cruzamento foi que algumas vezes as mudanças que trouxeram melhora à população, ocorreram logo nas primeiras iterações (ou gerações), e as demais trouxeram pouca melhora.

Esses resultados mostram a importância do processo de cruzamento para o resultado final das soluções, contribuindo não apenas para resultados melhores, como também na obtenção mais rápida de tais resultados.

Na figura 6.1 observa-se a comparação do melhor resultado alcançado quando a taxa de cruzamento foi estabelecida por volta de 20%. As taxas de comparação foram primeiramente 20% e depois 70%. A arranjo da matriz de processadores e chaves foi de 25 processadores e 3 camadas simulando um circuito de 5 componentes.

Pode-se observar que houve uma considerável diferença. Quando a taxa de cruzamento foi de apenas 20%, o melhor resultado foi encontrado num número consideravelmente maior de gerações (figura 6.1).

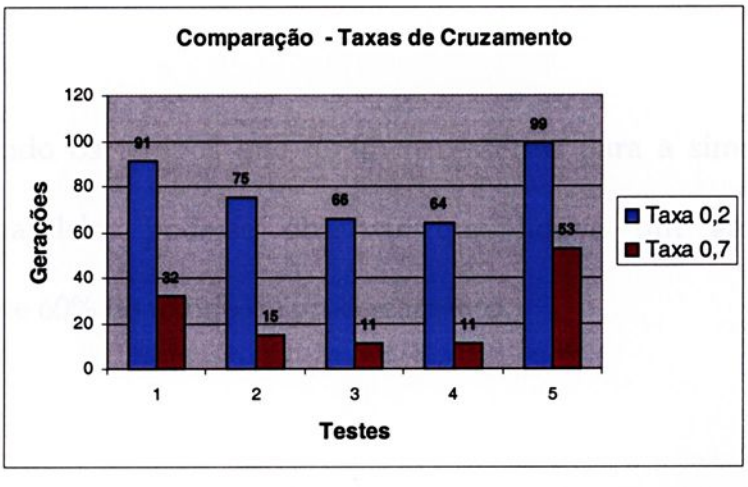


Figura 6.1: Demonstrativo das taxas de cruzamento para um mesmo arranjo e mesmo circuito.

Na figura 6.2 pode-se observar a média do custo dos melhores indivíduos encontrados com taxas diferentes de cruzamento. Verifica-se que tal custo foi menor quando a taxa de cruzamento foi maior. Ou seja, quanto maior a ocorrência do operador de cruzamento, melhor o resultado final apresentado (menor custo).

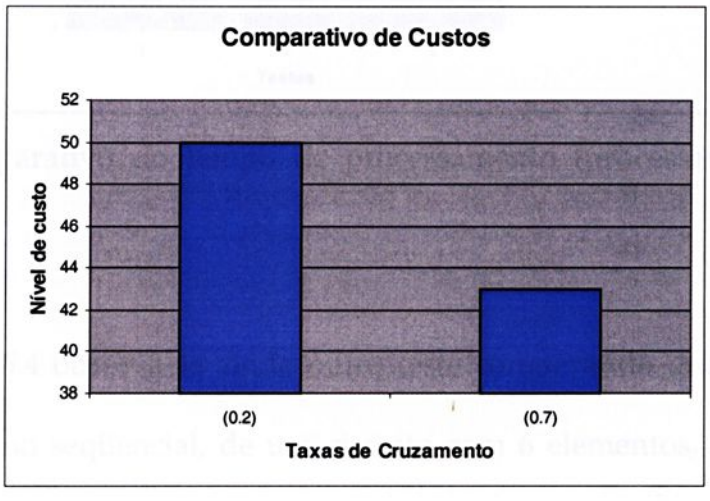


Figura 6.2: Demonstrativo da qualidade alcançada com taxas diferentes de cruzamento.

Na figura comparativa 6.3 é possível observar o ganho que se obteve numa simulação com 5 elementos numa arquitetura com 4 camadas e 25 processadores por camada.

Comparando os tempos que foram necessários para a simulação no modo seqüencial e paralelo, pode-se observar que houve um ganho médio de aproximadamente 60% no tempo de processamento.

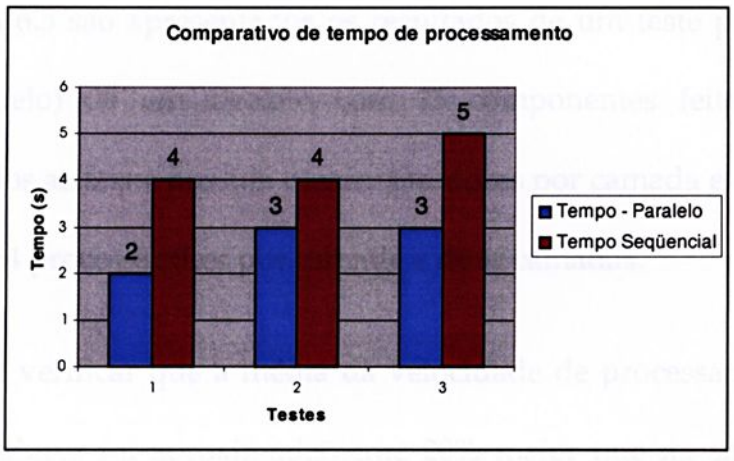


Figura 6.3: Comparativo do tempo de processamento (processamento paralelo e seqüencial).

Na figura 6.4 observa-se ainda outro teste comparando duas simulações, no modo paralelo e no seqüencial, de um circuito com 6 elementos, simulado em um arranjo de 25 processadores por camada, com 4 camadas.

Neste caso vê-se que a média de tempo do modo paralelo teve um ganho de aproximadamente 35% em relação ao modo seqüencial.

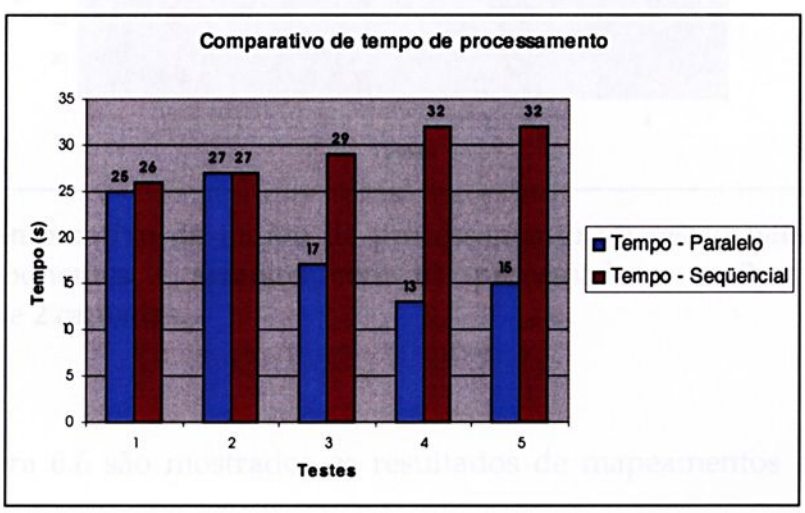


Figura 6.4: Comparativo do tempo necessário de processamento – entre implementação seqüencial e paralela

Na figura 6.5 são apresentados os resultados de um teste para mapeamento (no modo paralelo) de um circuito com 15 componentes feito sobre arranjos diferentes. Um dos arranjos possuía 81 processadores por camada e duas camadas. O outro, continha 64 processadores por camada e duas camadas.

É possível verificar que a média da velocidade de processamento no arranjo com 81 processadores foi aproximadamente 20% maior que na simulação sobre o arranjo com 64 processadores. Como o arranjo com 81 processadores possui maior número de chaves e processadores, há um aumento no percurso de roteamento que deve ser realizado entre os elementos, aumentando consecutivamente o tempo de processamento.

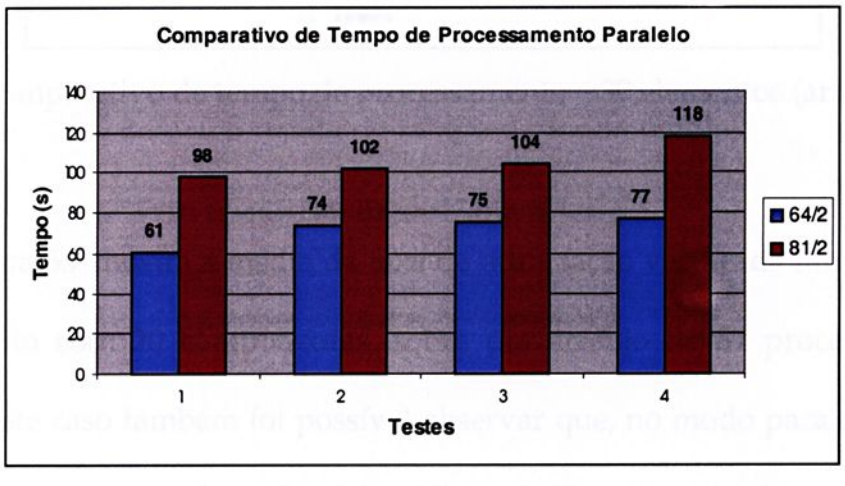


Figura 6.5: Comparativo de tempo de processamento no modo paralelo – circuito com 15 componentes - arranjos com 81 processadores e 2 camadas e; 64 processadores e 2 camadas.

Na figura 6.6 são mostrados os resultados de mapeamentos de um circuito com 30 elementos sobre um arranjo de 100 processadores por camada, com 2 camadas. Observa-se que nesse caso também houve ganho de velocidade no processamento, de cerca de 55% no modo paralelo.

A taxa média de otimização neste último caso foi de 40% no modo seqüencial e, 49% no modo paralelo. Dessa maneira, foi possível observar que, além de ganho de velocidade, a otimização no modo paralelo também foi maior. A taxa de otimização é medida em função do custo inicial da melhor solução comparada com o custo final da melhor solução após as iterações do algoritmo genético.

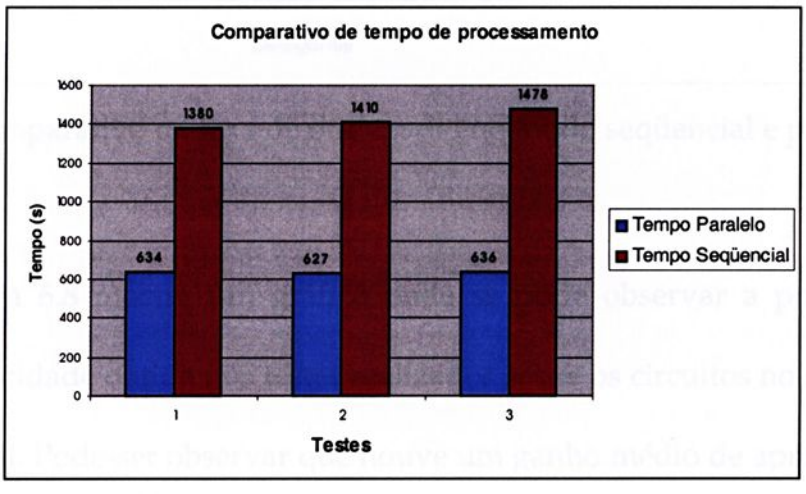


Figura 6.6: Comparativo de tempo de processamento – 30 elementos (arranjo 100/2).

A figura 6.7 mostra a média da taxa de otimização verificada numa simulação de um circuito com 20 componentes, sobre um arranjo de 81 processadores e 2 camadas. Neste caso também foi possível observar que, no modo paralelo, a taxa de otimização foi maior em relação à seqüencial.

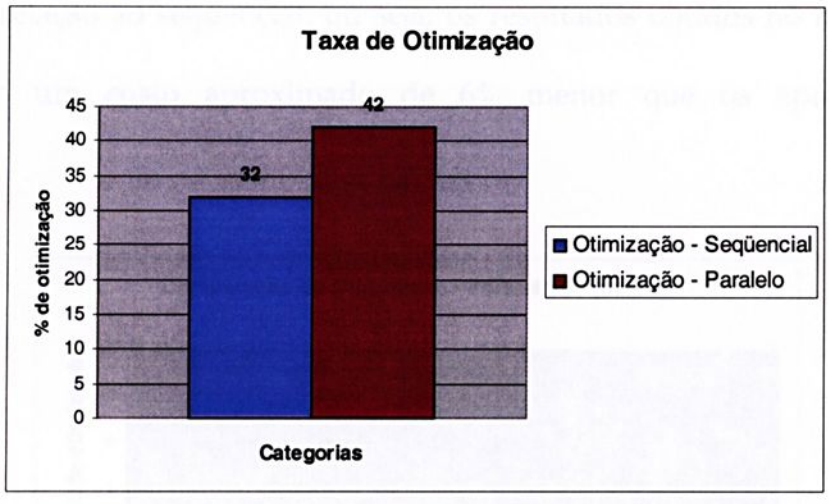


Figura 6.7: Comparativo da taxa de otimização no modo seqüencial e paralelo.

A figura 6.8 mostra um gráfico onde se pode observar a porcentagem do ganho de velocidade obtida nos testes realizados sobre os circuitos no modo paralelo e no seqüencial. Pode-ser observar que houve um ganho médio de aproximadamente 50% nos casos testados.

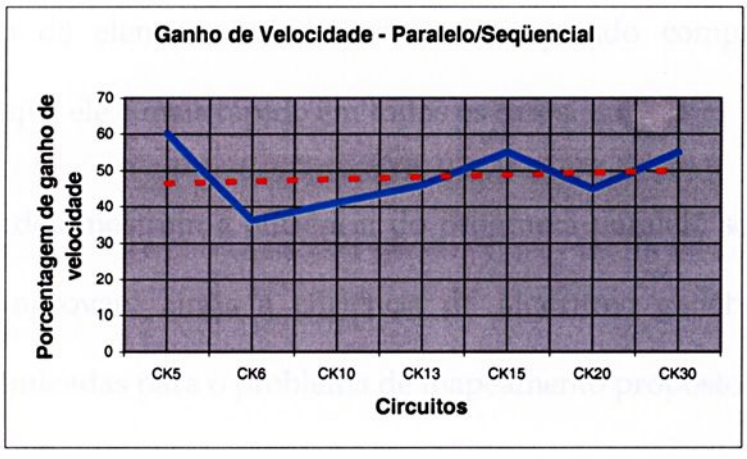


Figura 6.8: Porcentagem do ganho de velocidade na simulação do programa paralelo sobre o seqüencial.

A figura 6.9 mostra a porcentagem de otimização da implementação paralela sobre a seqüencial. Nota-se que houve um ganho aproximado de 6% no modo

paralelo em relação ao seqüencial, ou seja, os resultados obtidos no modo paralelo apresentaram um custo aproximado de 6% menor que os apresentados no seqüencial.

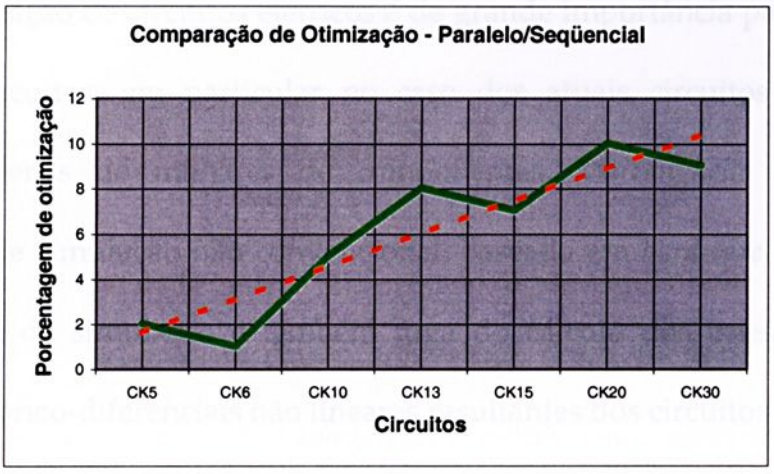


Figura 6.9: Comparação da otimização alcançada na implementação paralela sobre a seqüencial.

O tempo de necessário para o processamento no modo paralelo pode crescer com o aumento de elementos de circuito, mas quando comparado ao modo seqüencial vê-se que ele é mais rápido em todos os casos.

Os resultados mostram a eficiência do programa paralelo sobre o programa seqüencial e comprovam ainda a eficiência do algoritmo genético na busca de soluções mais otimizadas para o problema de mapeamento proposto.

7. CONSIDERAÇÕES FINAIS

A simulação de circuitos elétricos é de grande importância para o processo de projeto de circuitos, em particular no caso dos atuais circuitos VLSI, os quais possuem centenas de milhares de componentes. *Marranghello* [20] criou uma metodologia de simulação não convencional, baseada em *hardware* buscando ganho de velocidade de simulação e também fuga do cálculo dos imensos sistemas de equações algébrico-diferenciais não lineares resultantes dos circuitos VLSI.

A objetivo deste trabalho era propor um modelo de mapeamento de elementos de circuito na arquitetura proposta por *Marranghello*. Uma implementação seqüencial já havia sido desenvolvida demonstrando bons resultados no mapeamento.

Neste trabalho, além da paralelização do programa seqüencial, o módulo de roteamento foi implementado de modo diferente com relação ao primeiro programa seqüencial. No primeiro programa, quando o algoritmo de roteamento deparava-se com uma situação onde era impossível seguir por qualquer uma das direções, ele declarava o mapeamento em questão infactível. Nesta nova abordagem o algoritmo pode voltar em seu percurso e procurar outras trajetórias.

Os testes comparativos demonstraram que a implementação paralela teve vantagem sobre a seqüencial. Na maioria dos casos o programa paralelo teve ganho na velocidade de processamento quando comparado com o programa seqüencial.

Além do ganho na velocidade, foi possível observar que a qualidade dos indivíduos gerados no modo paralelo era, ainda que pequena, melhor que a qualidade dos indivíduos gerados a partir do modo seqüencial. Esse fato provavelmente se deva ao fato do programa paralelo explorar um espaço de busca mais diversificado, evitando-se mínimos locais. Na implementação paralela a população é dividida em subpopulações, enquanto que no modo seqüencial todos indivíduos permanecem numa mesma população.

Finalmente, pode-se concluir que os resultados obtidos comprovaram as expectativas de ganho de velocidade no processamento, além de demonstrarem melhoras também na qualidade dos indivíduos encontrados.

cm 1 2 3 4 5 6 7 unesp 1 12 13 14 15 16 17 18

8 REFERÊNCIAS BIBLIOGRÁFICAS

[1] R. HAUSSER, R. MÄUNNER, M. MAKHANIOK; **NERV: A Parallel Processor for Standard Genetic Algorithms.** Proceedings of 9th International Parallel Processing Symposium; 1995. Pg.: 782–789.

[2] ANTONETTE M. LOGAR, EDWARD M. CORWIN, THOMAS M. ENGLISH, **Implementation of Massively Parallel Genetic Algorithms On the MasPar MP-1**, Proceedings of the 1992 ACM/SIGAPP symposium on Applied computing: technological challenges of the 1990's March 1992. Pg.: 1015–1020.

[3] LUIZ S. OCHI, LÚCIA M.A. DRUMMOND, ROSA M.V. FIGUEIREDO, **Design and Implementation of a Parallel Genetic Algorithm for the Travelling Purchaser Problem**, Proceedings of the 1997 ACM symposium on Applied computing April 1997. Pg.: 257–262.

[4] HAN YANG FOO, JIANJIAN SONG, WENJUN ZHUANG, HENRIK ESBENSEN, ERNEST S. KUH. **Implementation of a Parallel Genetic Algorithm for Floorplan Optimization on IBM SP2.** High Performance Computing on the Information Superhighway; 1997. Pg.: 456–459.

[5] NORIAN MARRANGHELLO, KAMAL KANT BHARADWAJ. **On Mapping a Circuit Onto a Computer Architecture Using Genetic Algorithms.** Proceedings of EUFIT'97; 1997. Pg.: 794-798.

cm 1 2 3 4 5 6 7 unesp 1 12 13 14 15 16 17 18

[6] DAVID BEASLEY, DAVID R. BULL, RALPH R. MARTIN. **An Overview of Genetic Algorithms: Part 1, Fundamentals**. University Computing; 1993. Pg.: 58-69.

[7] DAVID BEASLEY, DAVID R. BULL, RALPH R. MARTIN. **An Overview of Genetic Algorithms, Part 2, Research Topics**. University Computing; 1993. Pg.: 170-181.

[8] K.S. TANG, K.F. MAN, Y.C. HO, S. KWONG. **A Realizable Architecture For Genetic Algorithm Parallelism**. Control Engineering Practice 6; 1998. Pg.: 897-903.

[9] ERICK CANTÚ-PAZ, DAVID E. GOLDBERG. **Efficient Parallel Genetic Algorithms: Theory and Practice.** Comput. Methods Appl. Engrg; 2000. Pg.: 221-238.

[10] HENRIK ESBENSEN, ERNEST S. KUH. **EXPLORER: An Interactive Floorplaner for Design Space Exploration**. Design Automation Conference, 1996, with EURO-VHDL '96 and Exhibition, Proceedings EURO-DAC '96, European; 1996. Pg.: 356-361.

[11] KEVIN DOWD, CHARLES R. SEVERANCE. **High Performance Computing**; O'Reilly & Associates Inc., Sebastopol. 2nd ed.; 1998.

[12] PAULA L. VAUGHANT, ANTHONY SKJELLUM, DONNA S. REESE, FEI-CHEN CHENG; **Migrating From PVM to MPI, Part I: The *Unify* System.** Frontiers of Massively Parallel Computation; 1995. Proceedings of Fifth Symposium on the; 1994. Pg.: 488–495.

cm 1 2 3 4 5 6 7 unesp 1 12 13 14 15 16 17 18

[13] DARRELL WHITLEY, **A Genetic Algorithm Tutorial**; Computer Science Department; Colorado State University; 1993. (http://samizdat.mines.edu/ga_tutorial).

[14] RICARDO SALEM ZEBULUM, **Síntese de Circuitos Eletrônicos por Computação Evolutiva**. Rio de Janeiro, 1999. Tese (doutorado). Departamento de Engenharia Elétrica - PUC-Rio.

[14] ADENSON DÍAZ, FRED GLOVER, HASSAN M. GHAZIRI, J.L. GONZÁLEZ, MANUEL LAGUNA, PABLO MOSCATO, FAN T. TSENG. **Optimización, Heurística y Redes Neuronales**. Madrid, Ed. Paraninfo SA; 1966.

[15] FÁBIO VINICIUS PINTO E SILVA. **Mapeamento de Circuitos em uma Arquitetura Específica Utilizando-se Algoritmos genéticos.** São José do Rio Preto, 1999. Monografia de Projeto Final. Departamento de Ciência da Computação e Estatística – UNESP.

[16] MURILLO CATARUCI MAROUELLI. **Influência da População inicial e dos Operadores Genéticos de Mutação e Cruzamento Sobre o Resultado de um Algoritmo Genético.** São José do Rio Preto, 2000. Monografia de Projeto Final. Departamento de Ciência da Computação e Estatística – UNESP.

[17] FADLALLAH, G.; LAVOIE, M.; DESSAINT, L.-A. **Parallel Computing Environments and Methods. Parallel Computing in Electrical Engineering.** PARELEC 2000. Proceedings. International Conference on; 2000. Pg.: 2–7.

[18] CLEMATIS, A.; TAVANI, O. **An Analysis of Message Passing Systems for Distributed Memory Computers Parallel and Distributed Processing**, 1993. Proceedings. Euromicro Workshop on , 1993 Page(s): 299 –306.

[19] http://www.netlib.org/pvm3

[20] NORIAN MARRANGHELLO. **Uma Metodologia para a Simulação de Circuitos ULSI**. Campinas, 1992. Tese (doutorado). Departamento de Semicondutores, Instrumentos e Fotônica – Universidade Estadual de Campinas.

[21] GHASSAN FADLALLAH, MICHEL LAVOIE, LOUIS-A. Dessaint. **Parallel Computing Environments and Methods**. Parallel Computing in Electrical Engineering, 2000. PARELEC 2000. Proceedings of International Conference on; 2000. Pg.: 2–7.

[22] MAURIZIO REBAUDENGO, MATTEO SONZA REORDA. **GALLO: A Genetic Algorithm for Floorplan Area Optimization.** IEEE Trans. on Computer-Aided Design of Integrated Circuits and System, Vol. 15, n° 8; 1996. Pg.: 943–951.

cm 1 2 3 4 5 6 7 unesp 1 12 13 14 15 16 17 18

UNIVERSIDADE ESTADUAL PAULISTA
"JÚLIO DE MESQUITA FILHO"
Câmpus de Ilha Solteira
Programa de Pós-Graduação em Engenharia Elétrica
Av. Brasil Centro, 56
15385-000 Ilha Solteira - SP
www.dee.feis.unesp.br