RouterServer:
Agente Roteador Paralelo de Células VLSI

Evandro de Araújo Jardini

Orientador: Prof. Dr. Dilvan de Abreu Moreira

Dissertação apresentada ao Instituto de Ciências Matemáticas e de Computação – ICMC-USP, como parte dos requisitos para obtenção do título de Mestre em Ciências de Computação e Matemática Computacional.

USP – São Carlos

Dezembro de 2000

A minha mãe Yone a ao meu pai Dermival (in memory)

Agradecimentos

Agradeço primeiramente a Deus, pela a ajuda em tudo que consegui e pelo apoio espiritual, confortando a minha alma nos momentos difíceis.

A minha mãe Yona e minha irmã Vanessa por me apoiarem diretamente nesta conquista.

Meus tios Sebastião Villela e Maria Rosa e a meu primo Tiago que me incentivaram a chegar até aqui.

Agradeço especialmente meu orientador e companheiro, Prof. Dr. Dilvan de Abreu Moreira por ter-me ajudado muito e sem o qual este trabalho não teria terminado.

A Cristiane, pela paciência que teve comigo.

A meus amigos da republica Alexandre, André, Adriano, Jener, Marcelo e ao Boro.

Ao amigo e chefe Alberto "Beco" Basílio, por ter segurado minha barra nos momentos que eu precisava me ausentar do serviço.

Aos amigos Mauro, Edmilson, Tatiana, Tomaz, Stênio, Enzo, Walter, Cláudio, Chandler, Emerson.

Aos amigos do grupo de pesquisa Carlos, Werley, Orlando, Flavia, Vanderlei, Elaine, Paulo.

Aos amigos aqui de São Carlos Rodrigo, Bibbo, João, Bruno, Everton.

E a todos aqueles que direta ou indiretamente contribuíram para o termino do trabalho.

Sumário

Capítulo 1 .... **1**

Introdução **1**

1.1 – Considerações Iniciais 1

1.2 - Motivação.. 3

1.3 - Objetivo 3

1.4 - Organização da Monografia 3

Capítulo 2 .... **5**

Projeto e fabricação de Circuitos Integrados **5**

2.1 – Introdução 5

2.2 – Escalas de Integração.. 5

2.3 – O Projeto de Circuitos 6

2.4 – O sistema Agents 8

2.5 - Posicionamento de componentes e roteamento. 9

2.6 – Considerações Finais 10

Capítulo 3 .... **11**

Agentes de Software e a Linguagem Java **11**

3.1 – Considerações Iniciais 11

3.2 - Agentes de Software.. 11

3.2.1 - Definições e Características de Agentes de Software 12

3.3 - A Linguagem Java 13

3.3.1 - Características da linguagem Java.. 15

3.4 - O Sistema Java 19

3.5 - Considerações Finais 20

Capítulo 4 .... **21**

Programação Paralela em Java **21**

4.1 – Considerações Iniciais 21

4.2 - Conceitos Básicos 22
4.2.1 - Concorrência e Paralelismo 22
4.2.2 - *DeadLock* 22
4.3 - Threads 23
4.3.1 - Estados dos *threads* 24
4.4 - Threads em Java 24
4.4.1 - Sincronização de *Threads* 26
4.5 – Considerações Finais 26
Capítulo 5 28
***O Servidor Router Serial* 28**
5.1 – Considerações iniciais 28
5.2 – Funcionamento do Servidor Router 28
5.2.1 – O Papel do Projetista 29
5.3 – Os Agentes RouterExpert e Connect 30
5.4 – Paralelismo Simulado 32
5.5 – Considerações Finais 34
Capítulo 6 35
***O RouterServer* 35**
6.1 – Considerações Iniciais 35
6.2 – Execução do Programa 35
6.3 – Implementação 36
6.3.1 – Paralelização da Máquina Vitual dos *Connects* 37
6.4 – Problemas de Sincronismo 38
6.5 - Considerações Finais 39
Capítulo 7 41
Resultados 41
7.1 – Considerações Iniciais 41
7.2 - Conjunto de testes em máquinas individuais 42
7.2.1 – Ambiente de Realização dos Testes 42
7.2.2 – O Tipos de Testes 42

7.2.3 – Resultado dos Testes com Componentes Posicionados 43

7.2.4 – Testes com o *Placer* e *RouterServer* executando em conjunto 48

7.3 – Conjunto de testes Distribuídos 49

7.3.1 – Ambiente de Realização dos Testes 50

7.3.2 – Resultados dos Testes 50

7.4 – Considerações Finais 52

Capítulo 8 54

Conclusões 54

8.1 - Considerações Iniciais 54

8.2 – Principais Contribuições 55

8.3 – Possíveis Trabalhos Futuros 55

8.4 - Considerações Finais 56

Referências Bibliográficas...57

Lista de Figuras

2.1: Exemplo da utilização de máscaras no processo de fabricação de circuitos integrados... 7
3.1: Diagrama de compilação e execução de um programa em Java...................... 15
3.2: Arquitetura da Máquina Virtual Java... 18
4.1: Threads sendo executados em processos............... 23
5.1: Circuito mapeado em um *grid* para utilização do algoritmo *maze routing*...... 29
5.2: Pontos Interessantes. 30
5.3 - Agentes *Connects* procurando por pontos.. 31
5.4: Máquina paralela virtual.. 32
7.1: Resultado de Roteamento LATCH CMOS de 1 Bit... 43
7.2: Teste realizados no Computador 1- 4 PentiunsPro.. 44
7.3: Resultados dos teste realizados no Computador 2.. 46
7.4: Resultados dos testes realizados no Computador 3... 47
7.5: Testes realizados com 1 servidor *Placer* e 4 servidores *Router*Servers............ 49
7.6: Teste distribuído utilizando circuito de Porta nand BICMOS com resistor...... 51
7.7: Teste distribuído utilizando circuito latch CMOS de 1 bit............................... 51
7.8: Resultado de Roteamento de uma Porta nand BICMOS com célula analógica... 52

Lista de Tabelas

2.1: Fases de fabricação de um circuito integrado.. 6

Resumo

O modo tradicional de criar o *layout* de circuitos ASICs (Application Specific Integrated Circuits) requer um projetista humano para interagir com um programa. Ele utiliza uma metodologia de *layout* baseada em bibliotecas de células-padrão (standard cells). Essa metodologia é boa para os projetos ASIC porque os processos realizados para geração de *layouts* podem ser automatizados, diminuindo o tempo gasto no projeto e aumentando a sua confiabilidade. Porém, essa metodologia possui inconvenientes em relação à manutenção da biblioteca de células e em relação ao número total e variedade de células existentes, pois, algumas células requeridas num projeto, podem não existir na biblioteca forçando uma adaptação do projeto.

O sistema *Agents 2* foi desenvolvido para automatizar a geração de células para circuitos integrados, mais precisamente células-padrão. O sistema é composto por vários agentes servidores, o *Placer* (que posiciona os componentes do circuito) e vários *Routers* (que conectam os componentes do circuito). Os servidores trabalham de forma distribuída em uma rede de computadores e a escalabilidade do sistema aumenta à medida que novos computadores são introduzidos na rede. Entretanto, o sistema não é capaz de explorar os recursos computacionais de máquinas multiprocessadas.

Para resolver esse problema foi desenvolvido nesse trabalho um novo servidor de roteamento, chamado de *RouterServer*. Ele foi desenvolvido usando a linguagem Java e programação multithreaded, permitindo que o sistema *Agents 2* possa explorar o paralelismo existente em computadores multiprocessados com memória compartilhada, mas mantendo sua escalabilidade em sistemas distribuídos em rede.

Abstract

The traditional way for creating layout for Application Specific Integrated Circuits (ASICs) demands that a human designer interacts with a computer program. This program uses a layout generation methodology based in standard cells libraries. This methodology is effective for designing ASICs because the processes for layout generation can be automated, reducing the time spent in designing circuits and increasing its reliability. However, this methodology has problems related to the maintenance of the cell libraries and to the total number and variety of available cells, because some cells required in a project may not exists in the library forcing design adaptations.

The *Agents 2* system was developed to automate the cell generation for integrated circuits (standard cells). Many server agents compose the system: the *Placer* (that places the components of a circuit) and many *Routers* (that wires the circuits' components). The servers work distributed over a computer network and system scalability increases as new computers are added to the network. However, the system is not capable of using the resources of computers with more than one CPU.

To solve this problem a new router server was developed, called *RouterServer*. It was developed using the Java language and a multithreaded design. It allows the *Agents 2* system to use the parallelism that exists in multiprocessors machines with shared memory, at the same time that it retains its scalability in distributed networked systems.

Capítulo 1

Introdução

1.1 – Considerações Iniciais

O projeto de circuitos integrados é normalmente baseado numa estrutura hierárquica de especificações. Os módulos em níveis superiores são compostos por submódulos e estes são formados por outros submódulos e assim sucessivamente. No fim dessa estrutura estão módulos formados apenas por transistores. Estes módulos são denominados de células-padrão (standard cells) [01].

O modo tradicional de criar o *layout* de circuitos ASICs (Application Specific Integrated Circuits) requer um projetista humano para interagir com um programa do tipo CAD (Computer Aided Design). Ele utiliza uma metodologia de *layout* baseada em bibliotecas de células padrão. Essa metodologia é boa para os projetos ASIC porque os processos realizados para geração de *layouts* (principalmente posicionamento e roteamento dos componentes) podem ser automatizados, diminuindo o tempo gasto no projeto e aumentando a sua confiabilidade [01].

Porém, essa metodologia possui inconvenientes em relação à manutenção da biblioteca de células no que se refere à atualização do processo de manufatura e adicionalmente, o número total e a variedade de células existentes, pois, algumas células requeridas em um projeto podem não existir na biblioteca forçando uma adaptação do projeto. Em conseqüência disso, o desempenho do circuito pode ser sacrificado [01].

A solução para os problemas apresentados é a geração de células-padrão específicas à medida que surgirem necessidades em projetos. Utilizando-se ferramentas

para geração automática de *layout* de células-padrão obtém-se grandes vantagens. Elas são capazes de gerar *layouts* para uma grande quantidade de circuitos SSI (Small Scale Integration) para diferentes processos de manufatura. Como essas ferramentas produziriam células que se encaixariam exatamente nas necessidades do projeto para diferentes tecnologias utilizadas para fabricação, elas solucionariam as duas principais inconveniências da metodologia baseada em células.

O sistema *Agents 2* tem justamente o objetivo de gerar automaticamente *layouts* de células-padrão para circuitos integrados. O sistema foi desenvolvido baseado na tecnologia de agentes de software e era, originalmente, escrito em C++ e Lisp. O sistema original foi portado para Java e é agora formado por dois programas servidores. Os programas se comunicam por uma rede de computadores e usam a linguagem de especificação EDIF (Electronic Design Interchange Format) para representar os circuitos. Os programas e suas respectivas funções são:

- Servidor **Placer**: Este servidor recebe em EDIF a descrição dos circuitos a serem gerados. Posiciona os componentes pertencentes ao circuito gerando diversos layouts com posicionamentos diferentes. Cada layout gerado é enviado a um servidor *Router* para ser roteado.

- Servidor **Router**: Este servidor tem o objetivo de rotear as trilhas que interconectam os componentes posicionados em um circuito eletrônico.

As informações relativas às regras de projeto que o layout tem de obedecer (que variam de acordo com o processo de fabricação) são lidas de um arquivo pelo *Placer* e transmitido para todos os *Router*s. Ambos os servidores, *Placer* e *Router* foram desenvolvidos para trabalharem de forma distribuída em uma rede de computadores. Isso permite um ganho de escalabilidade à medida que o número de servidores *Router*s aumenta.

1.2 - Motivação

Apesar do sistema *Agents 2* explorar o paralelismo inerente aos sistemas distribuídos em rede, espalhando os servidores *Router* por vários computadores através da rede, ele não consegue aproveitar recursos de computadores que possuam mais de um processador.

Para obter este proveito, o servidor *Router* tem de ser reescrito utilizando recursos de programação paralela, pois esses servidores são responsáveis pela maior parte do processamento.

1.3 - Objetivo

O objetivo principal deste trabalho é reescrever o servidor *Router,* denominado agora de *RouterServer*, para que o sistema *Agents 2* possa explorar o paralelismo existente em computadores multiprocessados com memória compartilhada.

O principal resultado esperado é um aumento de escalabilidade dos servidores *RouterServer* para máquinas de múltiplos processadores. A medida que houver aumento no número de processadores em um computador a velocidade de processamento do servidor deve aumentar. Além disso, o sistema deve manter a sua escalabilidade em sistemas distribuídos.

Nos objetivos secundários, estão inclusos os estudos das tecnologias de agentes de software, processamento distribuído, programação concorrente e paralela.

1.4 - Organização da Monografia

A monografia esta organizada da seguinte forma:

- O capítulo 2 descreve a tecnologia de microeletrônica e sobre o sistema *Agents* 2.

- No capítulo 3 são apresentados os conceitos de agentes de software e da linguagem Java.

- No capítulo 4 são descritos os conceitos e características de programação paralela e concorrente.

- O capítulo 5 descreve o servidor *Router* serial usado no sistema Agents 2 e o algoritmo de roteamento utilizado.

- No capítulo 6 são descritos o funcionamento interno do novo servidor *RouterServer* e a maneira como foi implementada a sua paralelização.

- O capítulo 7 traz os resultados de vários testes executados no *RouterServer* e no sistema *Agents 2* com esse novo servidor.

- O capítulo 8 apresenta as conclusões deste trabalho, considerando as dificuldades e contribuições, bem como opções de continuação e extensão desta pesquisa.

Capítulo 2

Projeto e fabricação de Circuitos Integrados

2.1 – Introdução

Apesar do foco desse trabalho não ser a geração de layout de circuitos integrados, para melhor entendê-lo é necessário algum conhecimento sobre a tecnologia básica da fabricação de circuitos integrados. Um circuito integrado é formado por componentes eletrônicos (transistores, diodos, capacitores, resistores) colocados numa pastilha de silício, denominada *chip*, através de um processo de fabricação [02].

Diferente do processo de fabricação de automóveis, onde as peças são adicionadas separadamente, no processo de fabricação de um circuito integrado não há a adição de componentes de forma separada, o processo é feito como um todo. Todos os componentes são fabricados ao mesmo tempo em camadas. Essa tecnologia de fabricação é denominada tecnologia de produção integrada [02]. Todo o circuito é encapsulado em um único invólucro e compartilha os mesmos pinos de contatos. Essa técnica possibilita reduzir os custos de fabricação dos circuitos.

2.2 – Escalas de Integração

O número de dispositivos eletrônicos presentes em um circuito está associado à escala de integração do mesmo. Circuitos com um número de componentes inferior a cem apresentam um baixo nível de integração, sendo designados pela sigla SSI (*Small Scale Integration*). Se este número estiver na faixa de cem a mil, o circuito é classificado como MSI (*Médium Scale Integration*). Circuitos na faixa de mil a cem mil

elementos pertencem à classe LSI (*Large Scale Integration*) e acima de cem mil elementos à classe VLSI (*Very Large Scale Integration*). Atualmente já são fabricados circuitos VLSI com vários milhões de dispositivos [02].

2.3 – O Projeto de Circuitos

O processo de fabricação de um circuito integrado é divido em três etapas básicas [02] como mostra a tabela 2.1.

Tabela 2.1: Fases de fabricação de um circuito integrado.

Etapa	Descrição
Etapa 1	Projeto do circuito e fabricação das máscaras.
Etapa 2	Obtenção de camada de Silício, Fotolitografia, Corrosão, Epitaxia, Metalização e Teste dos chips.
Etapa 3	Corte das pastilhas, Soldagem, Encapsulamento e Teste.

Para esse trabalho a primeira etapa, o projeto do circuito, é a mais importante. Antes de iniciar o processo de fabricação, é elaborado um projeto para a confecção do circuito. O projeto é a especificação de todo circuito que será desenvolvido. Fazendo uma analogia, o projeto do circuito pode ser comparado a uma maquete de uma construção, na qual se tem um esboço de toda a obra. O projeto é composto de várias máscaras. Essas máscaras são *layouts* específicos para cada camada que será sobreposta para formar o chip. A figura 2.1 demonstra o processo de fabricação de um circuito integrado utilizando mascaras.

O *layout* de cada máscara está relacionado ao material presente na camada do circuito que ela representa. Resulta da função que o circuito deve desempenhar.

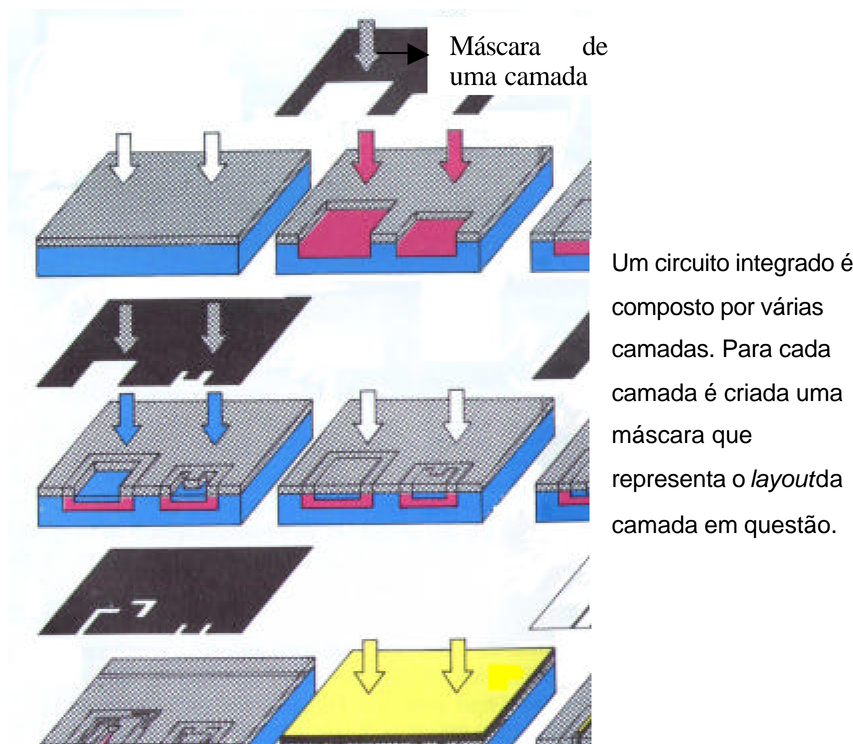


Figura 2.1: Exemplo da utilização de máscaras no processo de fabricação de circuitos integrados

Nos dias de hoje, o projeto de circuitos integrados digitais ASIC (Application Specific Integrated Circuit) é feito usando-se ferramentas de alto nível, como a linguagem VHDL (Very High Speed Integrated Circuit Hardware Description Language), que permitem o projeto e teste dos circuitos a nível funcional. O projetista não tem mais de especificar cada componente eletrônico individual do circuito. A partir desses projetos de alto nível, programas, chamados compiladores de silício, podem gerar o layout das várias máscaras do circuito para envio a uma fábrica.

Para gerar o layout final das máscaras de um circuito, os compiladores de silício usam bibliotecas que contém o layout de centenas de pequenas células de circuitos (chamadas *leaf cells*), tais como flip-flops, portas lógicas, registradores de deslocamento, etc. Essas bibliotecas de células-padrão de circuitos são feitas manualmente para cada processo de fabricação de chips de cada fabricante. Esse processo é demorado e custoso.

Após escolher as células-padrão mais apropriadas o compilador de silício posiciona cada componente na parte correta do circuito e os conecta por pistas condutoras (de alumínio ou cobre), completando assim o layout do circuito.

Porém o uso de bibliotecas de células padrões apresenta problemas. Um dos problemas é a possível falta de uma célula padrão que corresponda a um determinado componente, afinal existe um número finito de células na biblioteca, forçando adaptações no projeto que podem levar a resultados aquém do desejado. Outro problema é a geração da própria biblioteca, centenas de células de circuito têm de ser desenhadas e testadas individualmente, o que é um processo caro e demorado.

A utilização de ferramentas que automatizem a geração do layout dessas células, como o sistema *Agents*, pode solucionar problemas desse tipo. Uma vez que essas ferramentas geram células sob demanda, não há a necessidade da construção de bibliotecas e cada célula pode ser especificada para ter um tamanho específico e desempenhar uma função específica.

2.4 – O Sistema Agents

O sistema Agents é um conjunto de programas criados para gerar automaticamente *full custom* layout de *leaf cells* nas tecnologias CMOS, BICMOS e circuitos mistos digitais/analógicos. Ele foi desenvolvido originalmente [01] nas linguagens C++ e Lisp e foi posteriormente portado para a linguagem Java (*Agents 2*). O *Agents 2* é formado por dois módulos: os servidores *Placer* e *Router*. Esses módulos podem rodar distribuídos em rede usando o modelo cliente/servidor, com vários servidores *Router*s para cada servidor *Placer*.

O servidor *Placer* posiciona componentes em uma célula e usa os *Routers* para interconectá-los; o *Router* interconecta os circuitos que lhe são enviados. Um arquivo, que é lido pelo *Placer* e enviado a todos os *Router*s, guarda todas as informações que são dependentes do processo de fabricação (regras de projeto) específico de cada fabricante.

Esses servidores se comunicam através de uma rede de computadores usando o protocolo TCP/IP. O *Placer* recebe do seu cliente, conectado através da rede, a descrição e a *netlist* do circuito a ser gerado. O formato EDIF *(Electronic Design Interchange Format*) [03] é usado para a descrição do circuito. Depois de gerado o circuito o *Placer* retorna ao seu cliente, através da mesma rede, o layout do circuito codificado em EDIF.

O layout gerado pelo *Agents 2* não usa um *grid* virtual. Ele é um layout de máscara pronto para ser usado na confecção do circuito integrado. O programa não usa *grid* virtual nem durante o posicionamento dos componentes, nem durante o roteamento destes, todas as operações com layout são executadas ao nível de *layout* de máscara. Uma das vantagens do sistema *Agents 2* é sua flexibilidade em relação às tecnologias com que ele pode trabalhar, incluindo CMOS, BICMOS e bipolar. Além dessas tecnologias, o programa ainda pode trabalhar com pequenas células analógicas dentro de circuitos digitais.

2.5 - Posicionamento de componentes e roteamento.

O servidor *Placer* é quem recebe dos clientes a descrição dos circuitos a serem gerados [01]. Ele trabalha em três fases:

- Geração de colunas de transistores MOS que tem suas portas conectadas entre si ou de outros componentes que estejam interconectados (transistores bipolares, por exemplo).
- União das colunas de fets para formar grupos. Para se unir, colunas de fets tem de partilhar um certo numero de conexões via dreno ou fonte. A idéia é unir FETs que possam ser difundidos numa mesma linha de difusão.
- Posicionamento de grupos de componentes usando o algoritmo genético e envio do circuito resultante para um servidor *Router* para roteamento. Vários circuitos são gerados e enviados para roteamento, até que um deles seja roteado satisfatoriamente.

O servidor *Placer* usa diversos servidores *Router* ao mesmo tempo, desse modo ele pode gerar vários circuitos, solicitar aos roteadores que os conectem e coletar os

resultados à medida que os circuitos são roteados. Essa técnica explorara o processamento distribuído, repartindo a tarefa de rotear circuitos pelos vários computadores ligados a uma rede.

O servidor *Router* tenta rotear os circuitos enviados a ele pelos seus clientes. Esse servidor tenta imitar o modo que projetistas usam um programa CAD (Computer Added Design) para rotear circuitos. O CAD oferece aos projetistas as ferramentas básicas para manipular e armazenar o projeto e o projetista fica a cargo de todas as decisões importantes. As funções de CAD e projetistas ficam divididas entre grupos de objetos dentro do *Router*. O servidor *Router* será mais bem explicado no capítulo 5.

2.6 – Considerações Finais

O processo fabricação de circuitos integrados, mais necessariamente circuitos VLSI, não é uma tarefa trivial. Fatores como tipo de tecnologia de fabricação, finalidade do circuito, elaboração da mascara do *layout* do circuito, devem ser levados em consideração no momento da elaboração do projeto.

Iniciar hoje um projeto de circuito sem utilização bibliotecas de células-padrão e de ferramentas que auxiliam a geração destes circuitos é algo impraticável. Com a competitividade das empresas de componentes eletrônicos, o tempo e o custo são fatores essenciais para o sucesso de um projeto.

Porém, ao se projetar componentes eletrônicos baseados em bibliotecas, muitas vezes o projetista se depara com a falta de células-padrão especificas, sendo obrigado a sacrificar o projeto. Para solucionar este problema, ferramentas como o sistema *Agents 2*, podem ser utilizadas. Essa ferramenta é dividida entre a tarefa de posicionamento dos componentes do circuito, feita pelo *Placer*, e de interligação destes feitas pelo *Router*.

O objetivo desse trabalho é de produzir uma nova versão do servidor *Router* que use a tecnologia de *threads* e possa rodar de maneira paralela em máquinas com mais de um processador.

Capítulo 3

Agentes de Software e a Linguagem Java

3.1 – Considerações Iniciais

Este capítulo descreve duas das principais tecnologias utilizadas no trabalho, que são os agentes de software e a linguagem Java.

3.2 - Agentes de Software

Agentes são componentes de software que se comunicam com seus pares através da troca de mensagens usando uma linguagem de comunicação [04]. Essa característica permite aos agentes cooperarem entre si. Os agentes podem ser tão simples quanto uma sub-rotina ou ser entidades maiores como um programa com algum grau de autonomia [01]. Por meio do trabalho cooperativo, eles podem gerar *layouts* de circuitos VLSI e resolver problemas relacionados ao posicionamento e roteamento em circuitos [05].

O conjunto de tarefas ou aplicações em que um agente de software pode auxiliar é praticamente ilimitado: filtragem de informações; obtenção de informações em bases de dados; gerenciamento de email; escalonamento de reuniões; seleção de livros, filmes, música; etc. [06].

Devido à sua grande versatilidade, os agentes podem ser utilizados por diversas disciplinas da ciência da computação, como inteligência artificial, algoritmos genéticos,

processamento distribuído, segurança de dados, etc.

3.2.1 - Definições e Características de Agentes de Software

Definir agente de software é uma tarefa difícil em vista da grande quantidade de áreas que os utilizam no campo da ciência da computação. Algumas definições encontradas em [07] são:

"Agentes são programas de computadores semi-autônomos que auxiliam o usuário com tarefas no computador. Agentes empregam técnicas de inteligência artificial para auxiliar usuários com tarefas diárias no computador como leitura de email, agenciamento de tarefas, buscas de informações, etc. [44]".

"Agentes são sistemas computacionais que habitam ambientes complexos e dinâmicos. Eles compreendem e agem nesses ambientes. Por fazerem isso, eles realizam um conjunto de metas ou tarefas [27, 28, 29]".

"Agentes são robôs de software. Eles podem pensar e agir em nome do usuário para realizar tarefas. Agentes ajudarão a encontrar mais funções para necessidades crescentes, computação pessoal, sistemas de telecomunicações, etc. Uso de agentes inteligentes inclui tarefas independentes, operações semi-autonomas e comunicações entre o usuário e recursos de sistemas [3, 16]".

Ainda que não exista apenas uma única definição para agentes de software, existem características básicas que os agentes deveriam possuir. Segundo Flanklin [08], essas características são:

- **Autonomia**: Os agentes devem poder operar sem intervenção humana.

- **Habilidade social**: Agentes interagem com outros através de algum tipo de linguagem de comunicação própria.

- **Reatividade**: Agentes "percebem" seu ambiente (que pode ser um usuário via interface gráfica, uma coleção de outros agentes, a Internet ou talvez todos esses combinados) e respondem à medida que ocorrem mudanças.

- **Proatividade**: Agentes não apenas agem em resposta a seu ambiente, como são capazes de exibir um comportamento direcionado a metas tomando iniciativas.

- **Mobilidade**: A habilidade do agente de mover-se de uma máquina para outra utilizando algum meio físico de comunicação.

- **Aprendizado**: A habilidade do agente em mudar seu comportamento baseado em experiências anteriores.

Nem todos os agentes apresentam todas essas características.

3.3 - A Linguagem Java

Quando alguém se refere à linguagem Java geralmente não esta falando apenas da linguagem de programação, mas sim de todo um conjunto de recursos cujas características serão descritas mais adiante. Inicialmente, serão definidos alguns termos:

- **Java**: Java não é somente uma linguagem de programação [09]: Java é também um conjunto de especificações de APIs (Application Programming Interface) e máquinas virtuais.
- **Maquina Virtual e interpretadores**: A máquina virtual Java é responsável por executar os programas em Java. Ela é composta por um interpretador Java e as bibliotecas básicas da linguagem.
- **Aplicações e *applets***: Existem dois tipos de programas que podem ser escritos em Java: programas que podem ser executados diretamente com o interpretador Java e os desenvolvidos para serem executados através de um

browser de Internet. Os programas do primeiro tipo são denominados de aplicações e os programas do segundo tipo são as *applets*.

A linguagem Java foi desenvolvida para resolver problemas de forma prática e portável. Inicialmente, era parte de um projeto maior, o desenvolvimento de *software* para pequenos dispositivos eletrônicos [09].

Java é uma linguagem de programação orientada a objeto com sintaxe similar ao C++, porém mais simplificada [10]. As aplicações em Java são, de certa forma, mais robustas do que aplicações desenvolvidas nas linguagens C ou C++, isso porque todo o gerenciamento de memória é realizado pela máquina virtual Java, o que provê mais robustez e segurança às aplicações [10]. As applets, programas desenvolvidos em Java e copiados da Internet pelo usuário, podem ser executadas sem representar perigo devido aos mecanismos de segurança da máquina virtual Java. Esses mecanismos protegem contra o comportamento mal intencionado de aplicações restringindo, por exemplo, o acesso ao sistema de arquivos. Outro ponto de interesse na linguagem Java é o seu suporte a *threads*, que permite que aplicações sejam executadas de forma concorrente ou paralela (em máquinas com mais de um processador) aumentando assim seu desempenho em tempo de execução.

Além das características citadas acima, a linguagem Java possui uma outra grande atração, o fato de ser completamente portátil. Um código escrito em Java não precisa ser recompilado para ser executado em plataformas diferentes. Em lugar de produzir códigos específicos para uma determinada máquina, é produzido um código neutro intermediário denominado *bytecode* (figura 3.1). A máquina virtual Java traduz estes *bytecode* para as instruções específicas do processador e sistema operacional que estão executando a aplicação. Ela pode fazer isso usando um interpretador ou um JIT (Just in Time Compiler). Para executar uma aplicação num determinado ambiente, tudo que é necessário é ter uma máquina virtual Java específica instalada nesse ambiente.

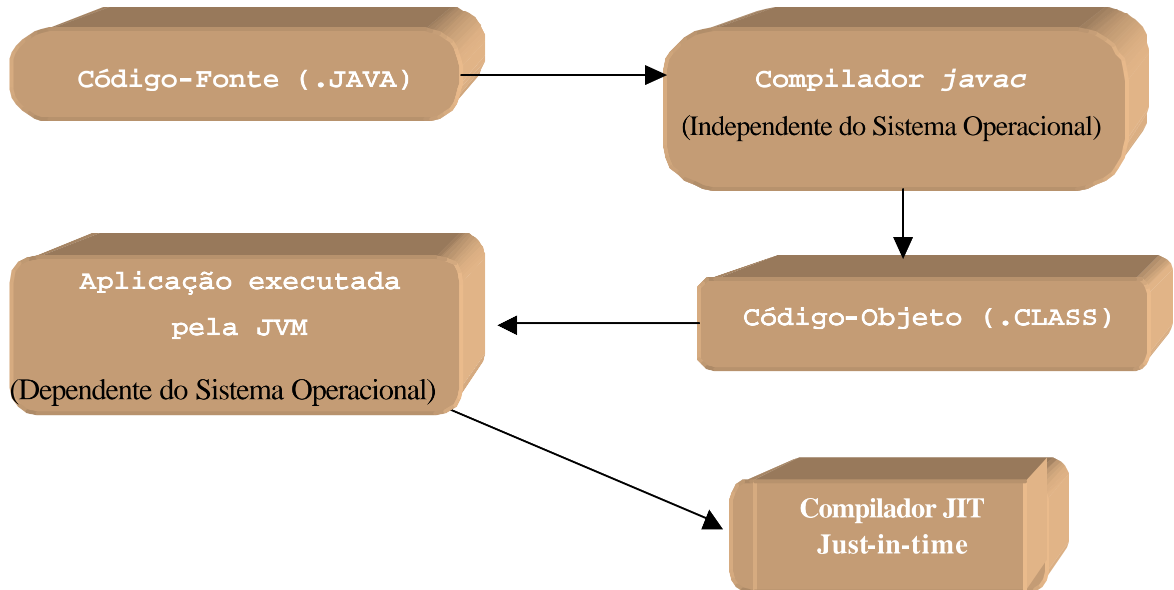


Figura 3.1: Diagrama de compilação e execução de um programa em Java.

3.3.1 - Características da Linguagem Java

Segundo a Sun Microsystems [09], as principais características da linguagem Java são:

3.3.1.1 - Simplicidade

A Java oferece toda a funcionalidade de uma linguagem potente como o C ou o C++, porém sem muitas das dificuldades apresentadas por estas. O C e C++ são linguagens com muitos recursos, mas também trazem dificuldades para o programador. Nestas linguagens todo o gerenciamento de memória fica a cargo do programador, o que é muito custoso. O C ou C++ não possuem um mecanismo gerenciador de memória. Em Java, o *garbage collector* foi projetado para cuidar do problema de liberação de memória. O mecanismo de *garbage collecion* consegue perceber quando um objeto não está mais em uso e elimina-o da memória [09].

3.3.1.2 - Orientação a Objetos

A programação orientada a objetos agrada as pessoas responsáveis pelo desenvolvimento de *software*. Para gerentes, ela promete desenvolvimento e manutenção rápidos e baratos. Para analistas e projetistas, a modelagem de processos torna-se simples e produz projetos limpos e gerenciáveis. Para programadores, a elegância e claridade do modelo de objetos e o poder das ferramentas e bibliotecas orientadas a objeto, torna a tarefa de programação mais agradável [11].

Java implementa a tecnologia básica de orientação a objetos do C++ com algumas melhorias. Ela trabalha com dados na forma de objeto e suporta três características próprias do paradigma da orientação a objetos:

- **Encapsulamento**: Consiste em definir os métodos para acesso e manipulação de dados dentro do objeto.
- **Herança**: Mecanismo pelo qual uma classe (local onde são definidos características e comportamentos de um objeto) herda atributos de outra classe.
- **Polimorfismo**: é a característica que permite um método ter várias implementações, que são selecionadas com base nos parâmetros deste método.

3.3.1.3 – Capacidade de Executar Aplicações Distribuídas

Em Java, existem bibliotecas que permitem a manipulação do protocolo TCP/IP. Estão incluídas as implementações extensíveis de ftp, http, smtp, bem como *sockets* de rede de nível inferior e interfaces de nomes. Isto permite interagir com poderosos serviços de rede sem ter que realmente entender os detalhes de baixo nível de muitos desses protocolos [11].

3.3.1.4 - Robustez

A linguagem Java realiza verificações em busca de problemas tanto em tempo de compilação como em tempo de execução. Java obriga a declaração explícita de tipos nos métodos, reduzindo assim as possibilidades de ocorrência de erros. O gerenciamento de memória é uma tarefa pesada nos ambientes tradicionais, o que força o programador a controlar toda a memória alocada e ter certeza de liberá-la de volta para o sistema quando ela não for mais necessária. Quando isto não ocorre, a memória fica sendo ocupada por códigos inúteis e o pior ocorre quando o programador libera um código que outra parte do programa espera continuar utilizando.

A linguagem Java remove a tarefa do gerenciamento de memória das mãos do programador. As funções *malloc* e *free* da linguagem C, não existem em Java. Em vez disso, o gerenciamento de memória é baseado em objetos e referência a objetos. Quando um objeto não é mais referenciado, ele é liberado da memória de forma automática pelo mecanismo de *garbage collection* da linguagem. Com isso, os erros relativos a gerenciamento de memória são eliminados.

3.3.1.5 - Independência de Plataforma

A linguagem Java foi projetada para suportar aplicações que sejam desenvolvidas para ambientes de rede heterogêneos. E, nestes ambientes, as aplicações devem ser capazes de serem executadas em uma variedade de arquitetura de hardwares e sistemas operacionais diferentes e interoperar com múltiplas interfaces de linguagem de programação. Para acomodar as diversidades de ambientes operacionais, o compilador Java gera os *bytecode*, um código neutro independente de arquitetura desenvolvido para transportar código eficientemente entre múltiplas plataformas de hardware e *software*.

A independência de plataforma se dá pelo uso da máquina virtual Java específica para cada plataforma operacional. Ela é uma máquina virtual para qual os compiladores da linguagem geraram códigos executáveis. A máquina virtual Java é baseada primeiramente sobre as especificações de interface POSIX, um padrão industrial de definições para especificações de sistemas portáteis. Atualmente existem *runtime* Java

para os sistemas Solaris 2.x, SunOS 4.1,x. *Windows* 95, *Windows* NT, Linux, Irix, AIX e *MacOS*.

A figura 3.2 representa a arquitetura da Máquina Virtual Java sobre uma plataforma operacional qualquer.

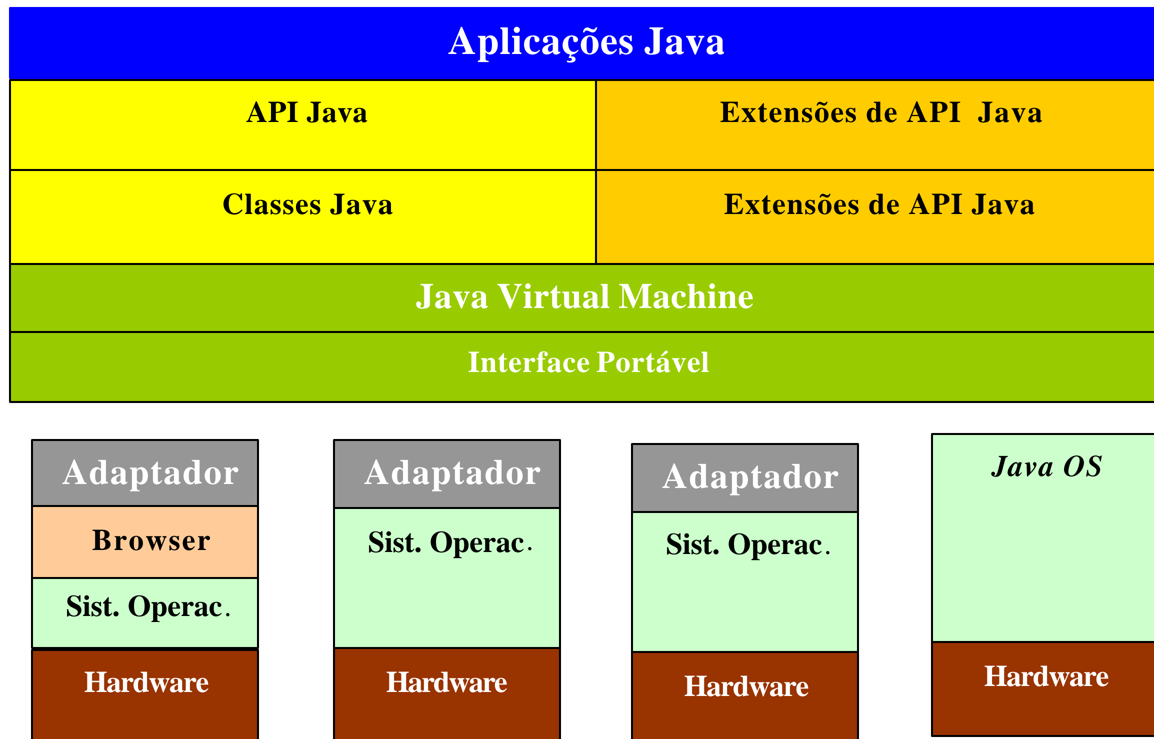


Figura 3.2: Arquitetura da Máquina Virtual Java

3.3.1.6 - Segurança

Java foi projetada para operar em ambientes distribuídos. Nestes ambientes, a segurança é um fator de grande importância. A maioria das maneiras consideradas óbvias, para invadir um sistema, simplesmente não podem ser feitas com um programa Java [11]. Visto que os programas Java não podem chamar funções globais e ter acesso a recursos arbitrários de sistema sem permissão do módulo *SecurityMannager*, há um certo nível de controle que pode ser exercido pelo *runtime* Java e que não pode ser conseguido em outros sistemas.

Antes de ser executado na máquina, o código Java passa por muitos testes através de um verificador de *bytecodes,* que compreende o formato dos fragmentos do código e analisa-os para detectar fragmentos de código ilegal, como violação de direito de acesso a um determinado arquivo; chamada de alguma função que possa apresentar comportamento indevido, como a finalização de um processo.

3.3.1.7 - Interpretada

O interpretador Java pode executar *bytecodes* Java diretamente sobre qualquer máquina no qual o sistema *runtime* foi portado. Em um ambiente interpretado como o Java, a fase de ligação de um programa é simples e leve, pois consome menos recursos do que as fases de compilação e ligação em uma linguagem geradora código de máquina para uma determinada arquitetura. O benefício disto é obtenção de melhoras no ciclo de desenvolvimento do *software.* Hoje em dia muitos interpretadores Java oferecem o recurso de um compilador JIT (Just In Time) para melhorar o tempo de execução dos programas.

3.3.1.8 - Dinâmico

Enquanto o compilador é rígido no momento de geração dos *bytecodes*, o sistema de *runtime* é dinâmico no estágio de ligação. *Class*es são ligadas somente quando necessárias. Novos módulos podem ser ligados sobre demanda. Nas aplicações do tipo *browser* de navegação na Internet, semelhantes ao *Hotjava*, códigos executáveis podem ser carregados de qualquer lugar permitindo *upgrades* automáticos das aplicações.

3.4 - O Sistema Java

O sistema Java completo inclui bibliotecas de classes e métodos úteis para o desenvolvimento de aplicações multiplataforma. Abaixo segue uma descrição resumida dessas bibliotecas:

- **java.lang**: A coleção de tipos bases que sempre são importados dentro de um programa compilado. Aqui são encontrados as declarações *Object* e *Class*, *Threads*, *Exceptions*, tipos de dados primitivos e uma variedade de outras *class*es fundamentais;
- **java.util**: Nesta biblioteca são encontradas as *class*es *Dictionary*, *HashTable* e *Stack*, entre outras, mais técnicas de codificação e decodificação e as classes *time* e *date*.
- **java.awt**: Nesta biblioteca encontra-se um *Abstract Windowing Toolkt* que provê uma camada de abstração que permite portar aplicações Java facilmente de um sistema *windows* para outro. Esta biblioteca contém classes para interface básica de componentes como eventos, cores e controles do tipo botões e barras de rolagem.

3.5 - Considerações Finais

Este capítulo apresentou uma visão geral do sistema Java e de suas características como arquitetura, bibliotecas, etc. Hoje, a maioria das aplicações desenvolvidas nesta linguagem são voltadas para ambientes de pesquisas ou ambientes distribuídos, tanto em rede local quanto na Internet.

O *RouterServer* é um programa de características científicas. Características como simplicidade na sintaxe da linguagem, orientação a objetos, distribuição, robustez, a independência de plataforma operacional e principalmente suporte a programação *multithreaded* foram fatores importantes na escolha da linguagem para o desenvolvimento desse projeto.

Capítulo 4

Programação Paralela em Java

4.1 – Considerações Iniciais

O termo programação paralela está ligado ao aumento de desempenho e escalabilidade. Neste capítulo do trabalho, serão discutidos características e conceitos envolvidos em paralelismo e concorrência de aplicações.

Segundo Almasi [13], as principais necessidades do processamento paralelo são a busca por maior desempenho, melhor custo/desempenho e produtividade. O processamento paralelo é uma coleção de elementos de processamento que podem comunicar e cooperar entre si para resolverem rapidamente problemas complexos. Navison [14], em seu artigo afirma que a maioria dos problemas computacionais podem ser resolvidos com programas paralelos. Programas paralelos podem ser implementados em vários ambientes.

O paralelismo de aplicações pode ser utilizado em diversas áreas [13], sendo algumas:

- **Projetos de engenharia e automação:** Análise de elementos finitos – cálculos de barragens, pontes, navios, aviões, grandes edifícios, etc; Aerodinâmica – estudo das turbulências; aplicações em sensoriamento remoto – Análise de imagens de satélite para obtenção de informações sobre agricultura, florestas, geologia, etc.

- **Operações em bancos de dados:** Operações em banco de dados também oferecem oportunidade de utilização de processamento paralelo, incluindo paralelismo dentro de transações.
- **Automação no projeto de circuitos VLSI:** O paralelismo pode ser utilizado na automação do desenvolvimento de circuitos VLSI incluindo simulação lógica, simulação de circuito, posicionamento e roteamento, etc.

4.2 - Conceitos Básicos

Esta seção apresenta alguns conceitos básicos sobre programação paralela.

4.2.1 - Concorrência e Paralelismo

Concorrência de processos ocorre quando um processo é iniciado sem o término de outros antes iniciados, passando a impressão de que os processos estão sendo executados de forma simultânea. O paralelismo, entretanto, é a execução simultânea de processos em unidades de processamento distintos.

Concorrência e paralelismo de processos podem acontecer simultaneamente em computadores multiprocessados, pois, mesmo havendo execução simultânea de processos, os mesmos competem entre si em tempo de execução nos processadores.

4.2.2 - *DeadLock*

O *deadlock* é um problema em potencial para qualquer sistema operacional [15]. Ele ocorre quando um grupo de processos tem garantido o uso exclusivo de um conjunto de recursos (um recurso pode ser uma unidade de fita, um registro em uma base de dados, uma impressora, etc), e cada um ainda precisa de mais recursos que pertencem aos outros processos do grupo. Neste caso todos estarão bloqueados e nenhum vai poder continuar sua execução.

4.3 - Threads

Até esse momento o paralelismo e a concorrência estavam sendo tratados ao nível de processos. Nesta seção, no entanto, será tratado um recurso muito utilizado atualmente no projeto de software que são os *threads*.

Um *thread* (figura 4.1) é uma seção de código executada independentemente de outras dentro de um processo [16]. Por serem executadas dentro dos processos, os *threads* compartilham a mesma área de memória, variáveis globais, e têm um custo menor, em comparação a um processo, para serem criados ou destruídos pelo sistema operacional. Sistemas que utilizam *threads* são denominados *multithreaded*.

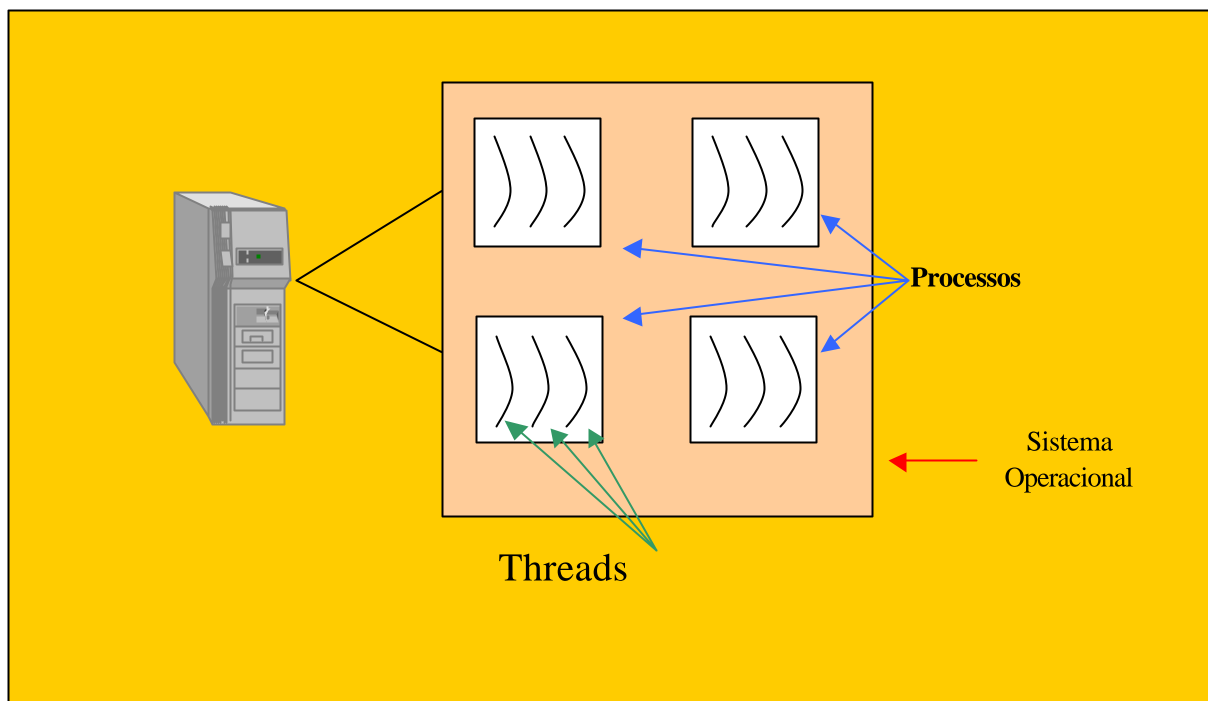


Figura 4.1: Threads sendo executados em processos

Existem várias vantagens em se utilizar programação *multithreaded*, algumas citadas por Moreira [17] são:

- Melhora do tempo de resposta das aplicações;
- Uso eficiente dos processadores em computadores multiprocessados;
- Menor utilização dos recursos do sistema;
- Melhor desempenho das aplicações.

4.3.1 - Estados dos *Threads*

Os *threads* podem assumir um dos seguintes estados [18]:

- **Novo**: Quando se cria um *thread* com o operador *new*, ele ainda não está executando. Isto significa que ele está no estado novo, ponto pra ser executado.
- **Executável**: Quando o *thread* é ativado pelo método *start*, ele torna-se executável. Um *thread* executável pode ainda não estar rodando. Fica a cargo do sistema operacional dar-lhe tempo de CPU para fazer isso.
- **Bloqueado**: Neste estado, o *thread* está interrompido e não pode ser executado. Um *thread* entra em estado bloqueado quando esta realizando uma operação de entrada/saída, quando os métodos *wait* e *sleep ()* são invocados ou quando acessa um monitor (esses termos serão tratados posteriormente).
- **Encerrado**: Um *thread* é encerrado quando o código que ele está executando terminar. Outra maneira de encerrar um *thread* é a utilização do método *stop*. Porém essa forma não é aconselhável, pois faz com que um *thread* termine de forma anormal, isto é, não encerrando o contexto (variáveis, contadores, etc) do mesmo, podendo causar inconsistência de informação.

4.4 - Threads em Java

A linguagem Java provê suporte a programação *multithreaded,* como parte fundamental da linguagem, permitindo facilmente a programas executarem de forma paralela.

É interessante fazer uma distinção entre os conceitos de *multithreads* e multitarefa. O primeiro refere-se a múltiplas seções de código sendo executadas independentemente dentro de um processo. O segundo refere-se à capacidade do sistema operacional em executar mais de um processo simultaneamente. Os *threads*, assim como os processos, podem ser executados de forma concorrente, paralela ou ambos.

O controle da execução dos *threads* em Java pode ser feito de duas maneiras, dependendo do tipo da implementação utilizada:

- **Utilizando *Green Threads***: O controle de execução dos *threads* em aplicativos utilizando *green threads* é feito pela própria máquina virtual Java. Neste tipo de implementação, o escalonamento dos *threads* é feito dentro da máquina virtual e nunca utiliza mais de um processador [19]. Sistemas Unix como Solaris, Linux, etc, podem utilizar esta implementação.

- **Utilizando *Native Threads*:** Neste tipo de implementação, o escalonamento dos *threads* é feito pelo próprio sistema operacional, assim, se uma aplicação desejar utilizar os processadores disponíveis em um computador, é este tipo de implementação que deverá utilizar [19]. Os sistemas Windows 95, Windows 98 e Windows NT 4.0 utilizam *native threads* como implementação e sistemas Unix também podem utilizá-lo.

Se existem duas maneiras de executar *threads,* como escolher entre uma ou outra? Em plataforma Windows não há a necessidade de escolha, toda programação *multithreaded* implementa *native threads*. Em sistemas Unix, existe uma *flag* no arquivo `/javahome/bin/.java_wrapper` que determina qual o tipo de implementação que será utilizado. Vale ressaltar que o código da aplicação não sofre qualquer modificação.

4.4.1 - Sincronização de *Threads*

Uma vez que os *threads* utilizam os mesmos espaços de memória, tem acesso às mesmas variáveis globais, pilhas, contadores de programa, etc, podem ocorrer problemas de sincronização de informações. Para resolver esses problemas a linguagem Java dispõe de mecanismos de sincronização. Alguns são descritos abaixo:

- ***Locked***: Esse termo se refere ao acesso obtido por um *thread* a um determinado método sincronizado.

- ***Synchronized*****:** Cláusula que indica que um método ou um pedaço de código de um método é sincronizado.

- ***Monitor***: É o principal conceito de sincronização utilizada em Java [20]. Cada objeto Java que tenha um método ou um trecho de um método sincronizado tem um monitor. Se um *thread* quiser executar um método sincronizado (ou trecho sincronizado) desse objeto, ele primeiro tem que conseguir o monitor do objeto. Apenas um *thread* por vez pode ter o monitor de um objeto.

- **`Wait()`, `Notify()` e `sleep()`:** São métodos Java que permitem fazer sincronização entre *threads*. Os métodos `sleep(time)` e `wait()` fazem com que um *thread* entre em estado bloqueado. O `notify()` serve para trocar o estado bloqueado para estado de execução de um *thread* bloqueado pelo `wait()`.

4.5 – Considerações Finais

Este capítulo tratou de um assunto chave no trabalho desenvolvido, que é a programação paralela de aplicações. O objetivo do uso deste tipo de programação é obter melhor performance do sistema à medida que mais agentes de software, na forma de *threads,* forem executados em uma máquina com mais de um processador.

Dos conceitos apresentados, o de maior importância é o *thread.* Em Java, as ferramentas para programação *mulithreaded* foram projetadas como parte fundamental

da linguagem e não incluídas depois da linguagem pronta (como é o caso da linguagem C). O resultado é uma facilidade de uso bem maior que é explorada na implementação do *RouterServer*.

Capítulo 5

O Servidor Router Serial

5.1 – Considerações Iniciais

O servidor *Router* original é um servidor de roteamento de circuitos integrados, mais precisamente células-padrão. Ele fazia parte de um sistema maior denominado *Agents 2*. Foi desenvolvido originalmente em linguagem C++ e depois portado para Java, sua execução é de forma seqüencial. Esse capítulo enfocará o funcionamento desse servidor.

5.2 – Funcionamento do Servidor Router

O *Router* foi desenvolvido tentando imitar o modo pelo qual os projetistas humanos utilizam um sistema de CAD (Computer Aided Design) para rotear circuitos. O projetista toma todas decisões importantes sobre o projeto, como por exemplo, onde os fios serão traçados, a qualidade do roteamento, a necessidade de reroteamento dos fios, etc. O sistema CAD fornece ao projetista ferramentas para representar e manipular o projeto.

O *Router* possui os componentes responsáveis por imitar o papel do projetista e o papel do CAD. O papel do projetista é feito pelos agentes de software *RouterExpert* e *Connect* e o papel do CAD é feito pelo objeto *Design*.

O objeto Design guarda os dados do projeto assim como os métodos para analisá-lo ou mudá-lo. Como ele faz o papel do CAD, ele prove os outros dois agentes

com os meios para coletar informações sobre o projeto (se dois fios se chocam, a área ocupada por uma subnet, etc.) e os métodos para implementar mudanças no projeto (inserir fios, destruir fios, etc.).

5.2.1 – O Papel do Projetista

Os agentes *RouterExpert* e *Connect*, realizam as tarefas executadas por um projetista, no que diz respeito ao roteamento dos circuitos.

Juntos, os agentes *RouterExpert* e *Connect* implementam o roteamento usando uma variação do algoritmo *maze routing* [21]. Esse algoritmo baseasse num *grid* retangular que representa o circuito. Ele mapeia este *grid* determinando quais células estão livres ou bloqueadas e para cada célula é atribuído um custo (figura 5.1). O algoritmo analisa e expande os vértices da árvore de busca gerada até encontrar o objetivo.

bloqueado	5	6	7	8	9
3	4	5	6	7	8
2®	3®	4®	5®	6®	7 (Objetivo)
1-	2	3	bloqueado	7	8
0- (Origem)	bloqueado	bloqueado	bloqueado	bloqueado	7
1	2	3	4	5	6

Figura 5.1: Circuito mapeado em um *grid* para utilização do algoritmo *maze routing*.

Uma variação desse algoritmo introduz o conceito de pontos interessantes [22]. Ao invés de expandir os vértices em todas as direções, os agentes *Connect* expandem diretamente para um ponto interessante. Um ponto é considerado interessante (figura 5.2) quando está alinhado com o ponto objetivo (figura 5.2A) ou com as extremidades de um obstáculo (figuras 5.2B e 5.2C). Na verdade à extremidade dum obstáculo é

somada uma margem de segurança proporcional à distância mínima de separação do obstáculo e do fio em roteamento mais a metade da largura desse fio.

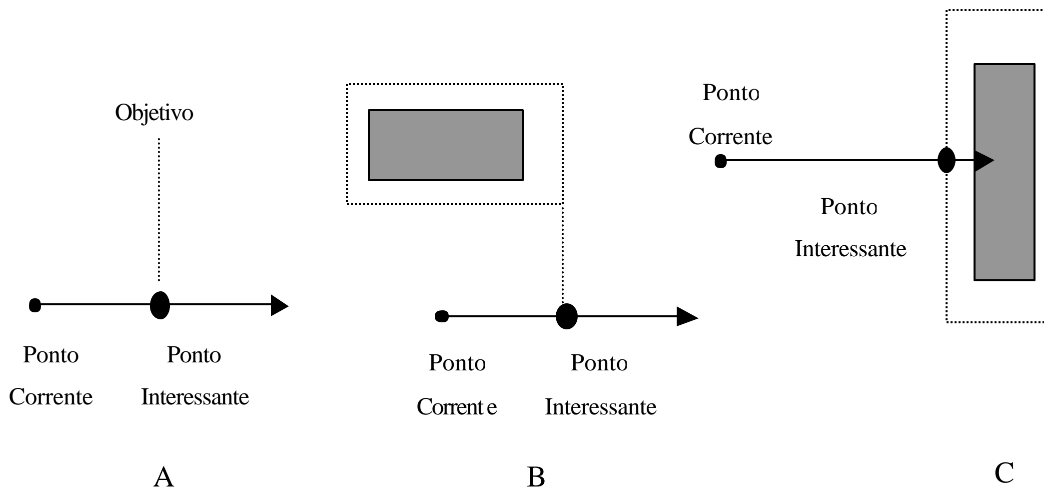


Figura 5.2: Pontos Interessantes.

5.3 – Os Agentes RouterExpert e Connect

Quando o servidor *Router* recebe um circuito do servidor *Placer*, ele o recebe como um objeto *Design* serializado. Ele recompõe o objeto e o passa para o agente *RouterExpert* dar início ao processo de roteamento.

De posse do objeto *Design*, o *RouterExpert* realizará, inicialmente, roteamentos simples. Ele conecta pontos que já estejam alinhados em polissilício ou difusão.

Para conectar os demais componentes restantes, os nós desconectados são colocados em uma lista, ordenados por importância e tamanho. Os menores serão os primeiros da lista. Deste estágio em diante, serão utilizadas apenas as camadas de *polissilício, metal1* e *metal2* para interconexão.

No objeto *Design,* cada nó contém uma lista de *subnets* que conectam parcialmente os componentes do nó (lista *routingNets*). Uma *subnet* é o conjunto de fios que interligam parte dos componentes que pertencem ao mesmo nó do circuito. Já uma *net* conecta todos os componentes do mesmo nó. O método *connectNode* do agente *RouterExpert* utiliza-se desta lista para conectar as *subnets* pertencentes a um nó. Ele faz isto utilizando o método *connectSubnet*. Se o método não conectar uma *subnet*, ela será colocada no final da lista e a seguinte será conectada. O método retorna uma condição de erro, se alguma *subnet* não for totalmente conectada.

A conexão das *subnets*, por fios¸ está baseada nos pontos interessantes e é realizada pelos agentes *Connect*. Eles analisam os pontos interessantes que estão próximos ao seu ponto de origem. Um agente *connect* é criado para cada ponto interessante. A partir desta análise, várias operações podem ser realizadas pelos *connects*: eles podem mudar de camada, conectar o fio com o destino, criar um novo pedaço de fio, se reproduzirem ao encontrar um obstáculo no caminho (figura 5.3), etc.

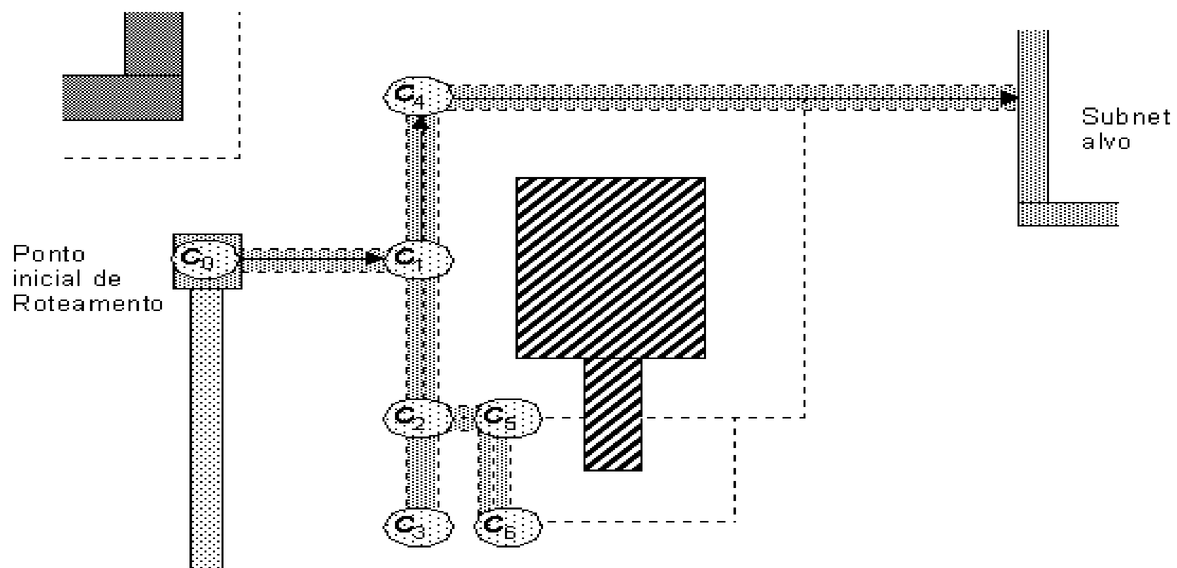


Figura 5.3 - Agentes *Connects* procurando por pontos

O objetivo dos agentes *Connect* é alcançar um ponto na *subnet* destino. O funcionamento básico dos agentes *Connects* inicia-se com a criação um agente *Connect*

em um ponto de origem dentro do *layout* (ponto C_0 na figura 5.3). Este agente analisará os pontos interessantes a sua volta. Para cada ponto interessante a ser explorado é criado um novo agente (pontos C_1, C_2, .., C_n) . Os agentes irão se reproduzir até encontrarem seu objetivo: a *subnet* destino.

5.4 – Paralelismo Simulado

O agente *RouterExpert* controla uma população de microagentes *Connect* e a maneira como eles executam o roteamento. A idéia é ter uma população de agentes simples (daí o nome de microagente para o *Connnect)* cooperando entre si de forma a encontrar uma solução. Se um agente encontra um novo ponto interessante, ele se reproduz. Se ele gastou todas as suas opções de procura, ele morre. Se completar um fio, ele envia este fio para o agente *RouterExpert*. O *RouterExpert* então, se encarrega de "matar" todos os agentes *Connects* cujo custo do fio seja maior ou igual ao encontrado.

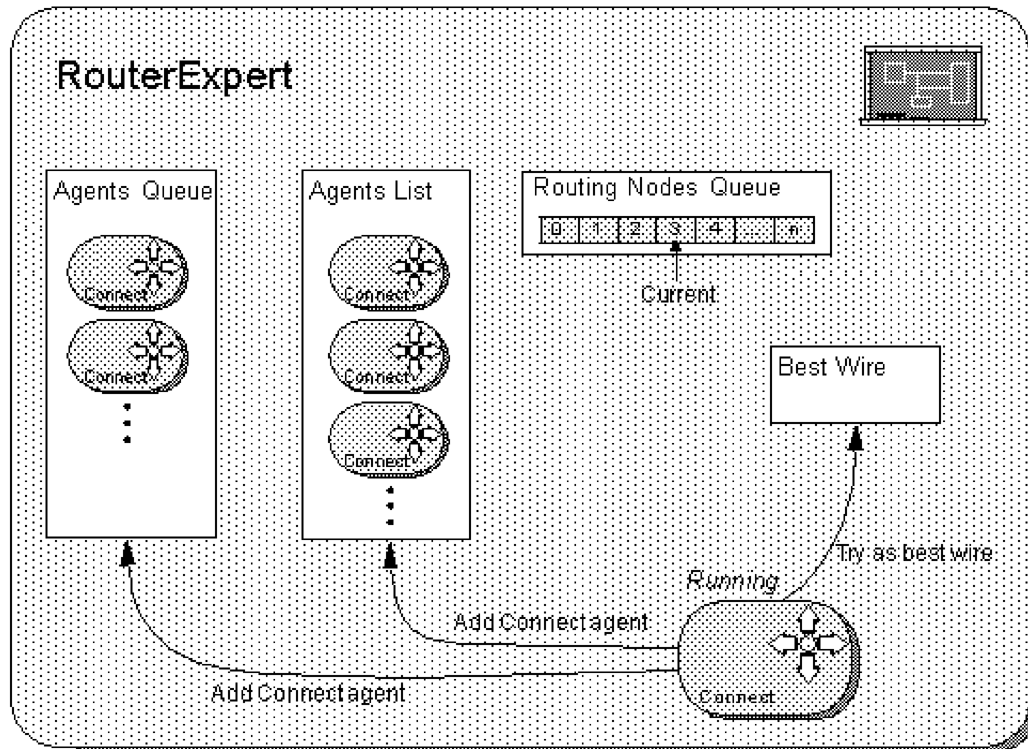


Figura 5.4: Máquina paralela virtual.

Para implementar essa "fazenda" de agentes *Connect*, controlada pelo agente *RouterExpert*, foi criada uma espécie de máquina virtual capaz de simular um sistema paralelo usando filas (figura 5.4). Um agente *Connect* por vez é rodado, à medida que ele roda ele pode sugerir *Best Wires* ao *RouterExpert* ou "procriar" e gerar novos agentes *Connect* para pesquisar outros pontos interessantes do projeto. Os novos agentes *Connect* criados são postos em duas filas:

- *Agents List* - Uma fila com agentes *Connect* comuns.
- *Agents Queue* - Uma fila para agentes *Connect* que iriam implementar a ação de desfazer conexões de um nó que bloqueava o seu caminho.

Essas duas listas são criadas em separado porque a operação de desfazer um nó é muito custosa computacionalmente (já que o nó desfeito terá de ser posteriormente refeito). Assim primeiramente são executados os agentes na *Agents List* e quando essa estiver vazia, é executado pelo menos um agente da *Agent Queue*.

Essa máquina virtual foi criada em C++, ao invés de se usar *thread*s e rodar os agentes *Connect* realmente em paralelo, porque na época não se dispunha de um computador com sistema operacional que permitisse o uso de múltiplos *thread*s. Ela foi mantida intacta no porte do servidor *Router* para a linguagem Java, gerando uma versão serial incapaz de usar os recurso de múltiplos *thread*s de execução em máquinas com muitos processadores.

5.5 – Considerações Finais

O servidor *RourterServer* é o software responsável por interligar os componentes dos circuitos do sistema *Agents 2*. Para realizar sua tarefa, ele utiliza basicamente dois tipos de agentes de software, o agente *RouterExpert* e o agente *ConnectExpert.*

Esses dois agentes são respons áveis pelo papel cabível a um projetista humano, ou seja, eles são diretamente responsáveis pelo roteamento do circuito. Enquanto um objeto Design guarda o projeto propriamente dito e tem os métodos para alterá-lo (papel do CAD).

O agente *RouterExpert*, controla uma população de agentes *Connects*. Esses agentes são os responsáveis pela procura das rotas de conexão dos componentes. Apesar das tarefas serem divididas entre um *RouterExpert* e vários *Connects*, elas são realizadas de forma seqüencial, gerando um impacto no tempo final de processamento.

Para que as tarefas realizadas pelos agentes *Connect*s fossem feitas de forma paralela, o servidor teve de ser reescrito. O capítulo seguinte descreve a maneira como isso foi feito.

Capítulo 6

O RouterServer

6.1 – Considerações Iniciais

O projeto do novo servidor de roteamento paralelo, *RouterServer*, partiu da versão seqüencial escrita em Java e foi adaptado para explorar o paralelismo existente em computadores com mais de um processador. Esse capítulo enfocará os detalhes de funcionamento e a implementação do *RouterServer*, descrevendo os principais métodos utilizados.

6.2 – Execução do Programa

Como o *RouterServer* foi desenvolvido utilizando a linguagem Java, existe a necessidade de uma máquina virtual Java instalada no computador. Dentre as várias máquinas virtuais existentes, a utilizada é do JDK (Java Developer Kit) versão 1.1.8 da IBM. A escolha desta máquina virtual se deu pelo seu melhor desempenho entre as que foram testadas. Tentou-se a versão mais nova da Sun, o JDK 1.2x, mas ele foi descartado pela constatação de um custo muito grande de chaveamento de contexto de *threads,* o que acarretaria em problemas para o projeto. A nova versão da Sun JDK 1.3x não estava disponível no início do projeto e não foi testada.

A execução do *RouterServer* é através de linha de comando. Pode-se passar parâmetros, para uma execução customizada, ou executá-lo sem parâmetro algum. Neste último caso o software executará com parâmetros *defaults*. Os parâmetros possíveis são:

- **-nthreads**: Indica a quantidade de *threads* que serão executadas simultaneamente.

- **-port**: O parâmetro indica qual a porta do protocolo TCP/IP que será utilizada. Utilizado na versão servidor.

- **-noserver**: Com esse parâmetro, o software irá trabalhar de forma *stand alone*, não aceitará requisições para roteamento de circuito através da rede. Após o roteamento do circuito, a execução do software termina.

- **-rules**: Esse parâmetro indica o nome do arquivo de regras de projeto a serem utilizadas pelo software. Nas regras estão incluídas informações como a distância mínima entre os fios de camadas diferentes, a largura máxima de um fio, qual camada pode sobrepor outra, etc. Utilizado apenas na execução *stand alone*.

- **-circuit**: Indica o nome do arquivo EDIF a ser roteado. Utilizado na execução *stand alone*. Na execução *stand alone*, o arquivo que será rotado já deve ter os seus componentes já posicionados.

Quando não é informado o parâmetro *–noserver*, significa que o *RouterServer* atuará como um servidor de roteamento trabalhando em conjunto com o *Placer*. Nesta configuração, o *RouterServer* atua como parte do sistema *Agents 2*. No momento da execução do software, é informada uma porta de escuta (recurso de comunicação do protocolo TCP/IP), para realização da comunicação entre o *Placer* e o *RouterServer*. A partir deste momento o software ficará em execução aguardando circuitos para serem roteados e devolvidos ao *Placer*. Sua execução só termina pela intervenção do usuário.

6.3 – Implementação

Os detalhes da paralelização do software serão descritos aqui. Apesar de trabalhar de forma distribuída, o servidor de roteamento original do sistema *Agents* não

era capaz de tirar proveito do paralelismo existente em computadores com mais de um processador. Contudo, seguindo-se a concepção de microagentes *Connect* com que o servidor original foi desenvolvido, o paralelismo em nível de máquina era o próximo passo na evolução do software.

O processo de paralelização iniciou-se com o estudo do código fonte original. Esse estudo serviu, antes de tudo, para detecção e correção de problemas existentes na versão original. Foram analisadas as partes do programa que poderiam ser executadas paralelamente e quais os impactos que essas modificações causariam.

Basicamente o modo dos agentes *Connect* procurarem por caminhos no circuito (figura 5.3) não foi alterado. O que foi mudada foi a máquina virtual que os roda. Sua nova implementação permite que os agentes *Connect*, que agora implementam a interface Java Runnable, rodem em *threads* de execução separados, permitindo que eles rodem em paralelo em máquinas com mais de um processador.

6.3.1 – Paralelização da Máquina Vitual dos *Connects*

Após a análise do código ficou definido que a paralelização do algoritmo seria feito na máquina virtual (figura 5.4) no momento em que houvesse a necessidade de executar agentes *Connect*s para procurar pontos interessantes, permitindo a execução simultânea desses agentes.

Quando um agente *Connect* é executado ele começa a analisar os pontos interessantes a sua volta. Para cada ponto interessante a sua volta, ele cria outros *Connect* agentes. Ao invés desses agentes serem colocados na lista *AgentsList*, que não existe mais, o agente *Connect* pai chama um método no agente *RouterExpert* que os põe numa lista, o *thread* de execução do *RouterExpert* tem um *loop* que cria novos *threads* para os agentes *Connect* nessa lista e os executa imediatamente (é nesse *loop* onde se pode controlar o número de agentes *Connect* que poderão rodar em paralelo, de um único agente até sem limites).

No mais o modo de funcionamento da máquina virtual permanece inalterado. Contudo o resto de seu código teve de ser alterado para resolver problemas de sincronismo entre os vários agentes *Connect* rodando e o *RouterExpert*. A lista *AgentsQueue* não foi removida, pois ela guarda os agentes *Connect* que irão implementar a ação de desfazer conexões de um nó (que bloqueia o seu caminho) e desfazer um nó é uma operação muito custosa. A lista *AgentsQueue* permite guardar esses agentes retardando a sua execução dando chance que um caminho menos custoso seja encontrado antes.

O *RouterServer* baseia-se no parâmetro *–nthread* para permitir a quantidade de agentes *Connect* que serão executados simultaneamente.

6.4 – Problemas de Sincronismo

Na versão seqüencial do roteador não ocorriam problemas de sincronismo, uma vez que só havia um agente *Connect* sendo executado por vez. Entretanto na nova versão, o problema de sincronismo foi introduzido, sendo um dos principais obstáculos a conclusão do projeto.

A solução foi analisar o código do agente *RouterExpert* e verificar quais métodos eram acessados pelos agentes *Connects*. Essa análise identificou que os seguintes métodos deveriam ser sincronizados:

- **Método *AddAgent***: Este método é chamado pelos agentes *Connects* toda vez em que há a necessidade de criar novos agentes *Connect* para pesquisa de novos pontos interessantes.
- **Método *AddAgentInQueue***: Este método é chamado pelos agentes *Connects* toda vez em que há a necessidade de criar novos agentes *Connect* que irão desfazer conexões em algum nó que os bloqueia.
- **Método *TryAsBestWire***: Quando um agente *Connect* alcança uma *subnet* com seu fio, ele chama esse método para propor esse fio ao *RouterExpert* como o

melhor fio. Esse método determina se o fio encontrado é melhor que o último proposto (cada fio tem um custo proporcional a seu material e comprimento).

Outro problema de sincronismo acontece quando um fio considerado aceitável pelo *RouterExpert* é mandado por um agente *Connect*. Esse fio vai ser então usado para conectar as duas *subnet*s. Mas vários agents *Connect* podem ainda estar rodando, o problema é como pará-los sem criar nenhum problema (inconsistência de variáveis, interlock entre *thread*s, etc.). Os próprios agentes *Connect* resolvem esse problema:

- Eles constantemente monitoram se seus fios são piores que o melhor fio que o *RouterExpert* tem, caso isso aconteça eles param.
- Eles monitoram se o *thread* deles foi interrompido pelo *RouterExpert* (chamando o método `interrupted()` no *thread* do agente *Connect*), caso isso aconteça eles param.

Finalmente algumas adaptações tiveram de ser feitas em algumas estruturas de dados internas que não possuíam suporte a *multithreads*.

6.5 - Considerações Finais

Esse capítulo procurou demonstrar as diferenças de funcionamento entre o servidor *Router* serial e o novo servidor *RouterServer* paralelo, tanto na sua execução quanto no seu funcionamento interno. A maneira como foi desenvolvido torna o *RouterServer* um software flexível e escalável:

- Flexível porque permite ser executado tanto como um programa *stand alone* ou como um servidor de roteamento, trabalhando de forma paralela e distribuída.
- Escalável porque sua implementação utiliza recursos de programação paralela, permitindo a exploração dos recursos computacionais de computadores com mais de um processador.

Essa nova versão do servidor de roteamento foi incorporada ao sistema *Agents 2* em substituição a versão do servidor antiga não paralela. Vale destacar que as maiores dificuldades práticas em paralelizar o servidor de roteamento se deram em dois pontos principais: entender o funcionamento do software, pois o sistema *Agents 2* possui cerca de 10000 linhas de código, e resolver os problemas de sincronização entre *threads*.

No capítulo seguinte serão analisados os resultados obtidos com o novo servidor de roteamento *RouterServer*, tanto com o programa rodando sozinho quanto com ele rodando como parte do sistema *Agents 2*.

Capítulo 7

Resultados

7.1 – Considerações Iniciais

Neste capítulo serão discutidos os resultados obtidos dos testes realizados com o novo servidor *RouterServer*. Para os testes, foram utilizados dois circuitos de processos de fabricação diferentes: O circuito da porta nand BICMOS, que utiliza o processo de fabricação Orbit [23] e o circuito latch de 1 bit CMOS, que utiliza o processo de fabricação ES2 (*Europen Silicon Structures*) [23], os mesmos utilizados em [1].

Uma das motivações para realização deste trabalho foi o interesse em pesquisas envolvendo a exploração dos recursos fornecidos por computadores multiprocessados. A idéia é que a medida em que há um aumento da utilização desses recursos o tempo final gasto no processo de roteamento dos circuitos deve diminuir. O objetivo dos testes aqui reportados foi testar se as modificações realizadas no servidor *RouterServer* permitiram maior escalabilidade à medida que o número de processadores em um computador aumentar.

Foram feitos dois conjuntos de testes: O primeiro, testou o desempenho do servidor *RouterServer* em relação à sua utilização com diferentes números de *threads* e processadores. O servidor *RouterServer* foi executado em apenas uma máquina de cada vez. Esses testes analisaram sua escalabilidade e também tentaram determinar o número ótimo de *threads* simultâneos para um melhor desempenho em cada tipo de configuração. O segundo conjunto de testes analisou o desempenho do programa como parte integrante do sistema *Agents 2* sendo executado de forma distribuída numa rede de computadores heterogêneos.

Nas seções seguintes serão mostrados e discutidos os resultados obtidos nos testes. Primeiramente será mostrado o conjunto de testes realizados em uma máquina de cada vez e em seguida o conjunto realizado como parte do sistema *Agents 2*.

7.2 - Conjunto de Testes em Máquinas Individuais

Os teste em máquinas individuais visam explorar os recursos de processamento de computadores com memória compartilhada. O principal resultado esperado nesses testes são a diminuição do tempo gasto para rotear os circuitos conforme haja um aumento da capacidade de processamento do computador.

7.2.1 – Ambiente de Realização dos Testes

Para a realização dos testes foram utilizadas as seguintes máquinas:

- Computador 1 – Computador Intergraph com 4 processadores Pentium Pro de 200 Mhz utilizando Windows NT 4.0.
- Computador 2 – Computador com 2 processadores Pentium Pro de 200 Mhz utilizando Windows NT 4.0.
- Computador 3 – Notebook com 1 processador K6 II 400mhz utilizando Windows 98

7.2.2 – O Tipos de Testes

O conjunto de testes em máquinas individuais foi realizado de duas formas. A primeira testou o servidor *RouterServer* realizando o roteamento de circuitos com componentes já posicionados, a vantagem desta forma é a de se testar o servidor *RouterServer* independentemente do *Placer*. A segunda forma testou os servidores *Placer* e *RouterServer* sendo executados simultaneamente. Deve-se ressaltar que nesse caso a qualidade dos resultados e o tempo de roteamento dependem da maneira com que o servidor *Placer* irá posicionar os componentes.

7.2.3 – Resultado dos Testes com Componentes Posicionados

Os *layouts* mostrados nas figuras 7.1, é o resultados do roteamento obtido do circuito latch *CMOS de 1 bit*.

Os gráficos gerados na figura 7.2 demonstram os resultados obtidos com a execução do servidor *RouterServer* com os circuitos citados anteriormente.

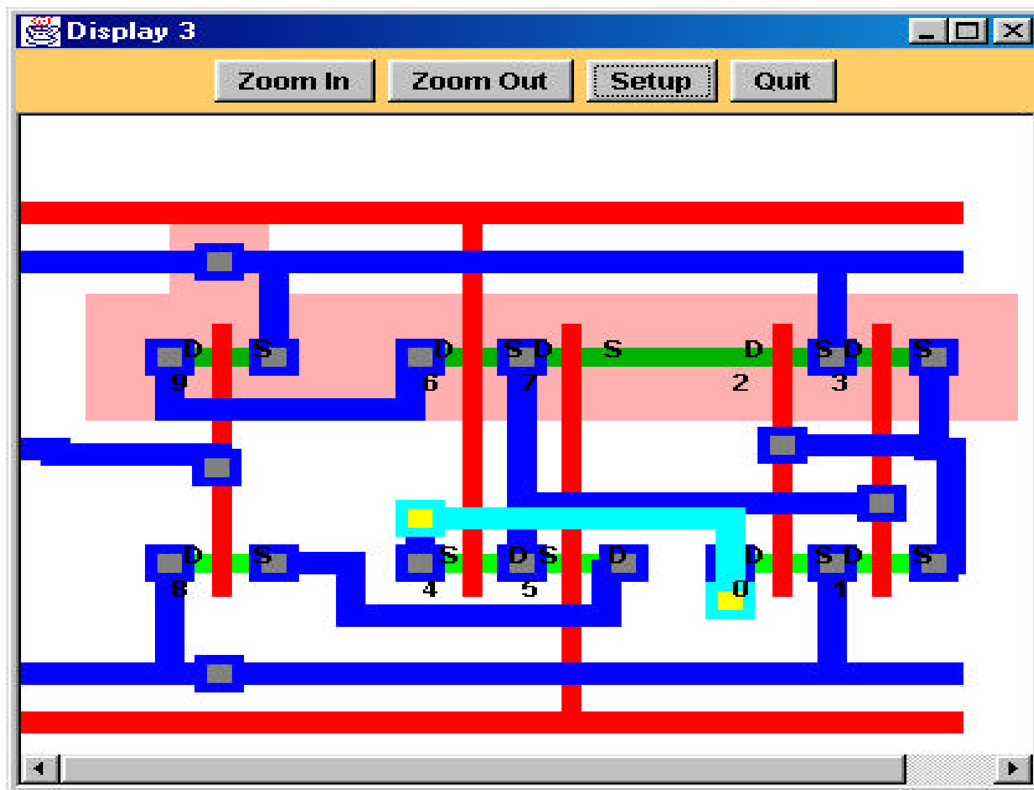
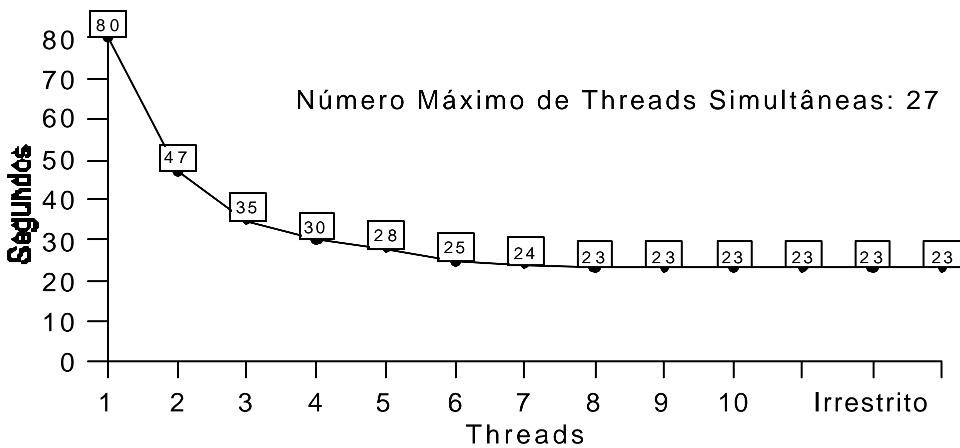
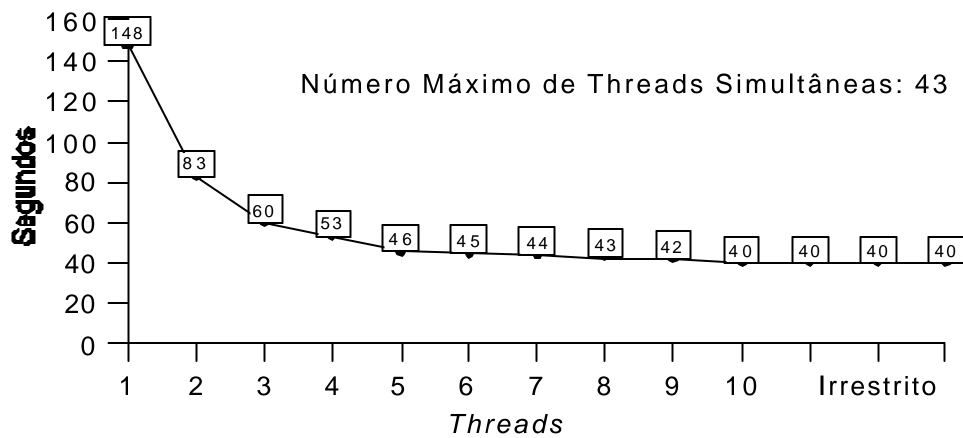


Figura 7.1: Resultado de Roteamento LATCH CMOS de 1 Bit

Computador 1





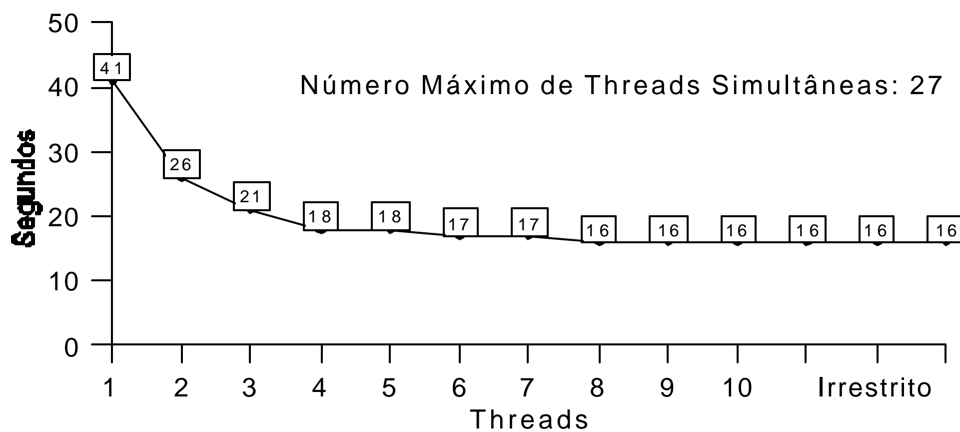


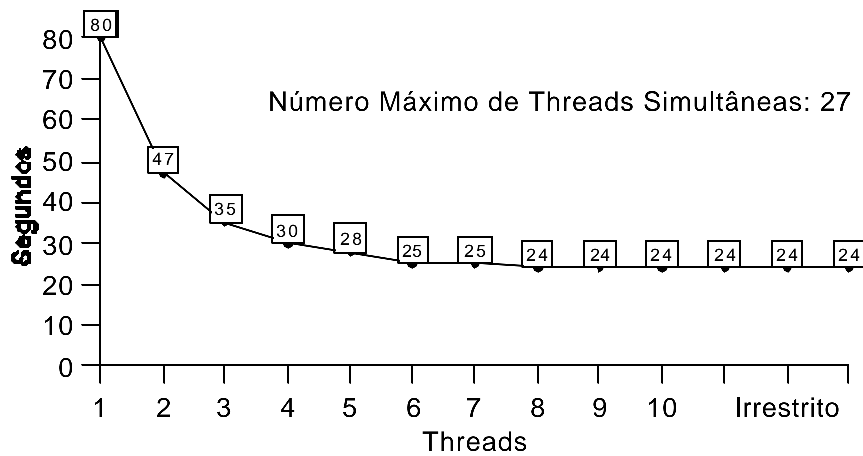
Figura 7.2: Teste realizados no Computador 1- 4 PentiunsPro.

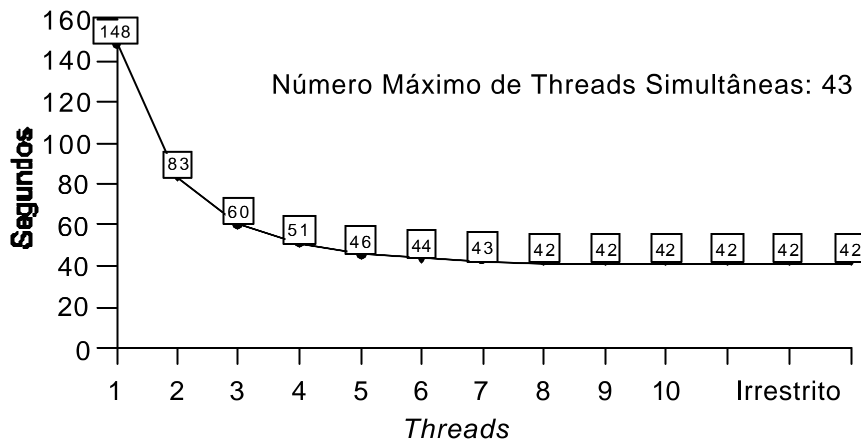
Os gráficos da figura 7.2 demonstram a escalabilidade obtida na execução do roteamento dos circuitos apresentados. O eixo X do gráfico representa a quantidade de *threads* em execução. Note que à medida que o número de *threads* aumenta, o tempo, representado no eixo Y, diminui. Porém esta queda ocorre até um determinado momento, a partir deste ponto ocorre uma estabilidade e independente da quantidade de *threads* em execução ou dos processadores disponíveis, o tempo será o mesmo.

O motivo da estabilidade no tempo ocorre porque a partir de um certo ponto não há mais aumento da execução simultânea de *threads*. Como já citado no capítulo 6, os agentes *Connects* são implementados na forma de *threads* e são criados à medida que surgem pontos interessantes para serem pesquisados. A partir do momento que não há mais pontos interessantes, não haverão novos *threads*, e mesmo que haja ociosidade nos processadores, esses não serão utilizados. Isso quer dizer que todo o paralelismo que o programa podia explorar foi explorado.

Os gráficos das figuras 7.3 e 7.4 demonstram os resultados obtidos com os computadores 2 e 3.

Computador 2





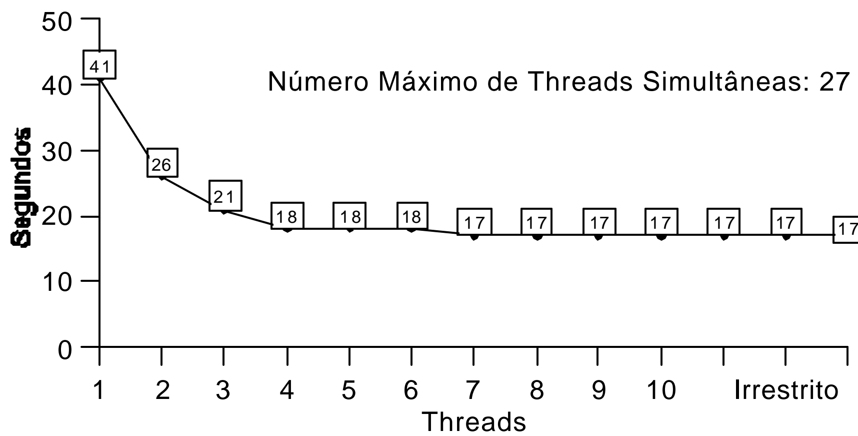


Figura 7.3: Resultados dos teste realizados no Computador 2

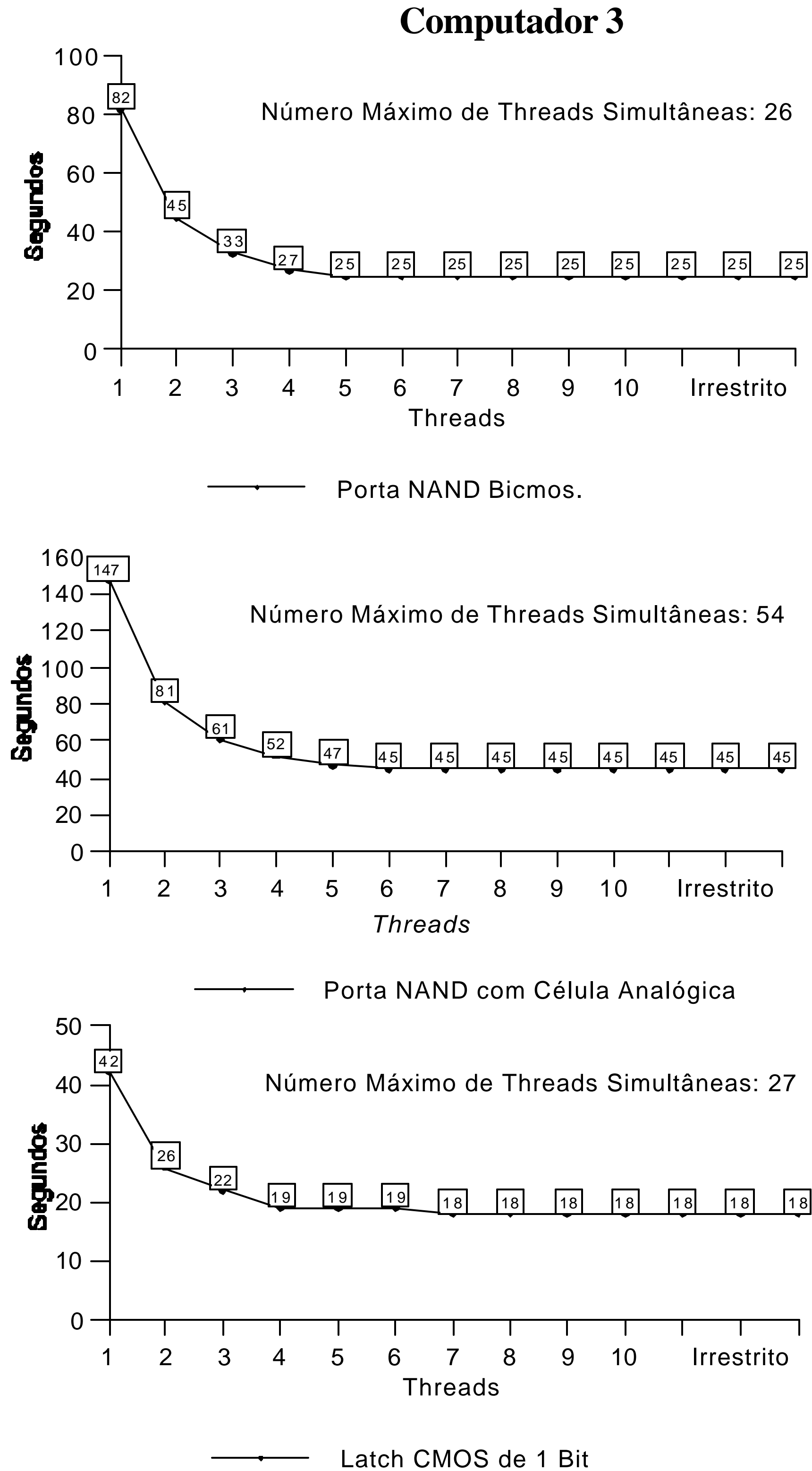


Figura 7.4 : Resultados dos testes realizados no Computador 3

Os gráficos apresentados na figura 7.4 comprovam novamente a escalabilidade do *RouterServer* em relação ao número de *threads* disponíveis. Primeiramente, como já foi discutido, se for feita uma análise dos gráficos separadamente, fica comprovado que à medida que o número de *threads* disponíveis aumenta, o tempo de roteamento dos circuitos diminui até um ponto limite. Agora, se os gráficos dos computadores 1, 2 e 3 para cada circuito forem comparados uns com os outros, comprova-se a escalabilidade do *RouterServer* em relação ao número de processadores disponíveis. À medida que o número de processadores disponíveis vai aumentando, os tempos gastos nos roteamentos dos circuitos tendem a diminuir.

Vale ressaltar que a diferença entre os tempos de execução entre as máquinas não foi maior pelo motivo de que apenas um só *Router* gerando um só circuito não necessita de muito poder de computação. Mais adiante quando vários *RouterServers* estiverem trabalhando no mesmo computador essas diferenças se tornarão mais visíveis.

Outro problema que deve ser relembrado é que as máquinas envolvidas em todos os testes não diferem apenas em números de processadores, mas também em velocidade, memória, *motherboards*, etc. Assim os valores obtidos não podem ser considerados valores precisos e sim aproximações razoáveis dos verdadeiros resultados.

7.2.4 – Testes com o *Placer* e *RouterServer* Executando em Conjunto

Os testes agora utilizam o servidor *Placer* para posicionar os componentes e enviá-los para servidores *RouterServer*. Para esses testes foram utilizados um servidor *Placer* e vários servidores *RouterServer* variando em número. A idéia é saturar o poder de processamento dos computadores executando mais *RouterServers*. Desta forma computadores com mais processadores terão vantagens sobre os que contêm menos processadores.

Para a realização desses testes foram gerados 3 layouts por cada computador. Foram tomados os menores tempos e a média dos tempos de geração. A figura 7.5 mostra o *Placer* e o *RouterServer* gerando layouts para o circuito da porta nand

BICMOS com uma célula analógica (resistor). Não há restrições ao número de *threads* nos *Routers* (essa foi considerada a melhor opção de número de *threads*).

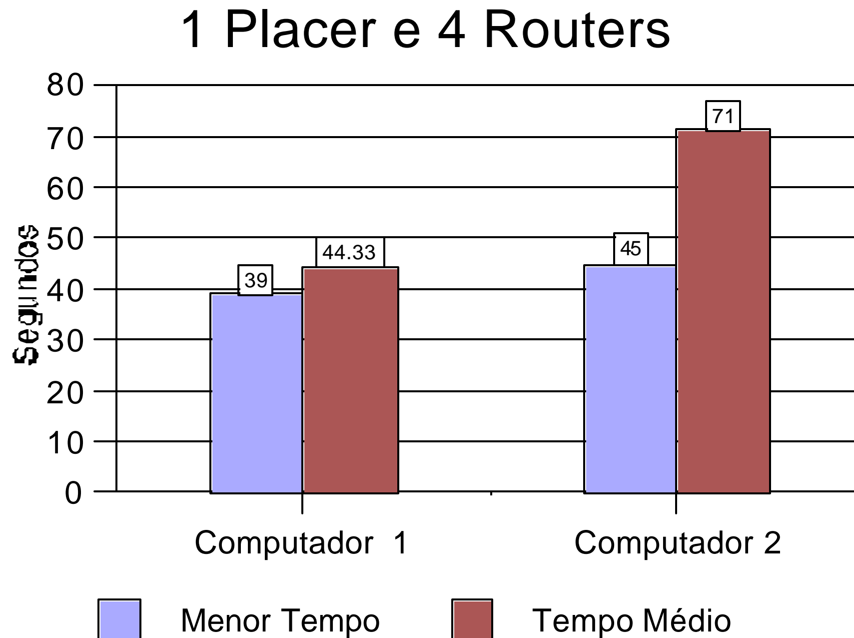


Figura 7.5: Testes realizados com 1 servidor *Placer* e 4 servidores *Router*Servers

Embora o gráfico não represente medidas precisas, já que as duas máquinas têm outras diferenças além do número de processadores (velocidade, memória, *motherboard*, etc.). Ele claramente mostra o desempenho superior da máquina de 4 processadores. Isso demonstra a escalabilidade do programa com o número de CPUs disponíveis numa máquina.

7.3 – Conjunto de Testes Distribuídos

Este trabalho também teve o comprometimento de manter a escalabilidade do sistema em ambientes distribuídos obtida na versão original do sistema *Agents*. O objetivo dos testes é de mostrar a diminuição dos tempos de roteamentos à medida que novas estações são acrescentadas.

7.3.1 – Ambiente de Realização dos Testes

Para a realização dos testes foram utilizados os seguintes computadores:

- Computador 1 – Microcomputador Intergraph com 4 processadores Pentium Pro de 200 Mhz utilizando Windows NT 4.0.
- Computador 2 – Microcomputador com 1 processador Pentium II de 266 Mhs utilizando Windows NT 4.0.
- Computador 3 – Microcomputador com 2 processadorres Pentium II de 350 Mhz utilizando Windows NT 4.0.

Estando eles conectados por uma rede Ethernet de 100 Megabits/s.

7.3.2 – Resultados dos Testes

Os testes foram realizados utilizando-se um computador, que não faz parte do ambiente de teste, para executar somente o servidor *Placer*. Este computador era o responsável por enviar circuitos posicionados aos demais computadores participantes do ambiente de teste. Em cada computador participante, estavam sendo executados 4 servidores de roteamento, sendo que não há limite de criação de *threads* nestes servidores.

Os gráficos das figuras 7.6 e 7.7 seguintes demonstram os resultados obtidos. A figura 7.8 é um exemplo do *layout* de uma Porta nand BICMOS com célula analógica

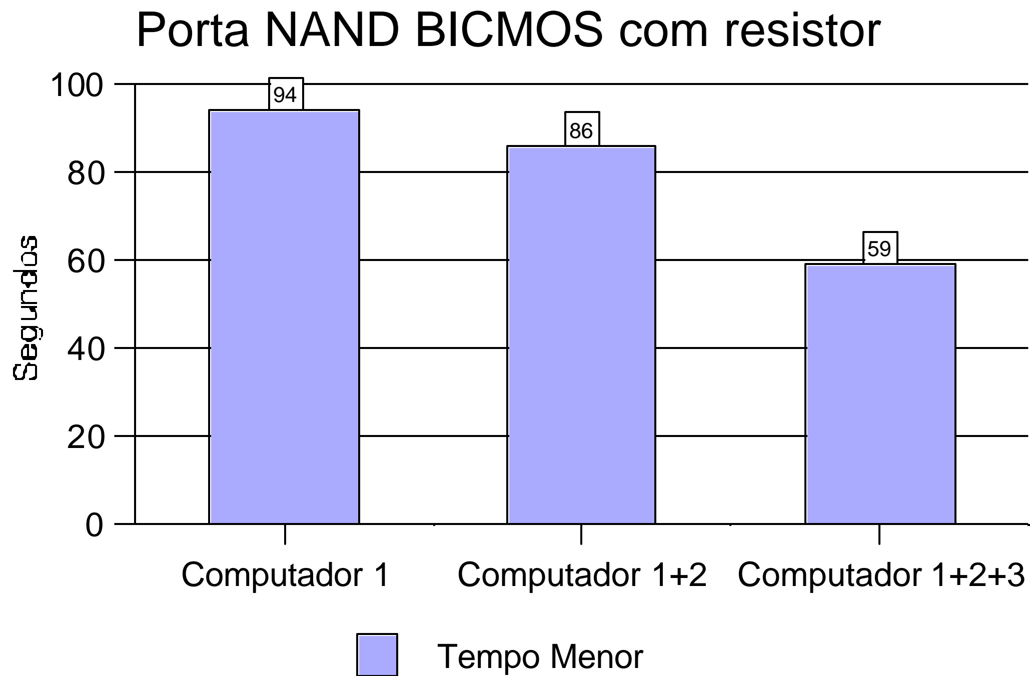


Figura 7.6: Teste distribuído utilizando circuito de Porta nand BICMOS com resistor.

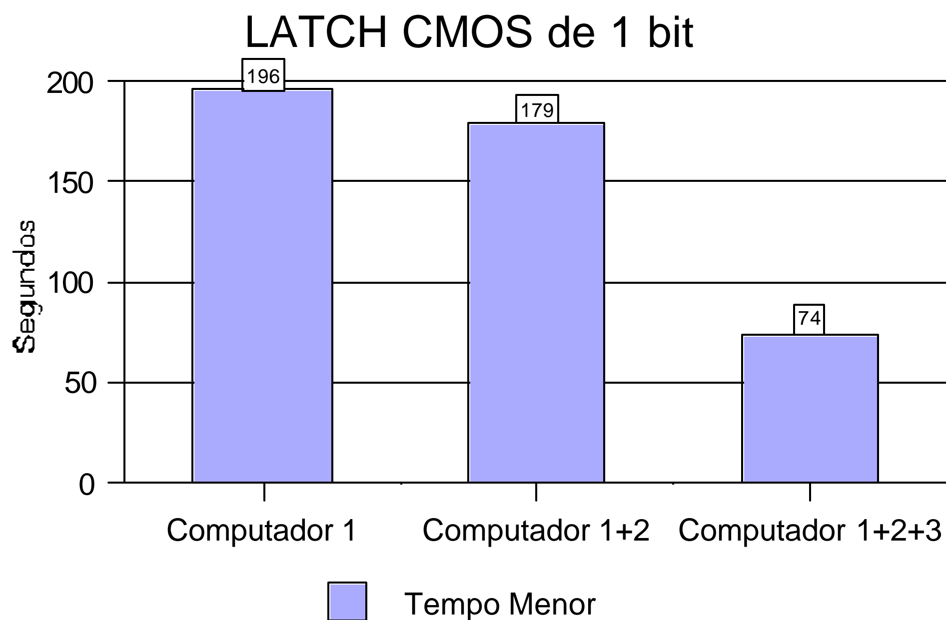


Figura 7.7 : Teste distribuído utilizando circuito latch CMOS de 1 bit.

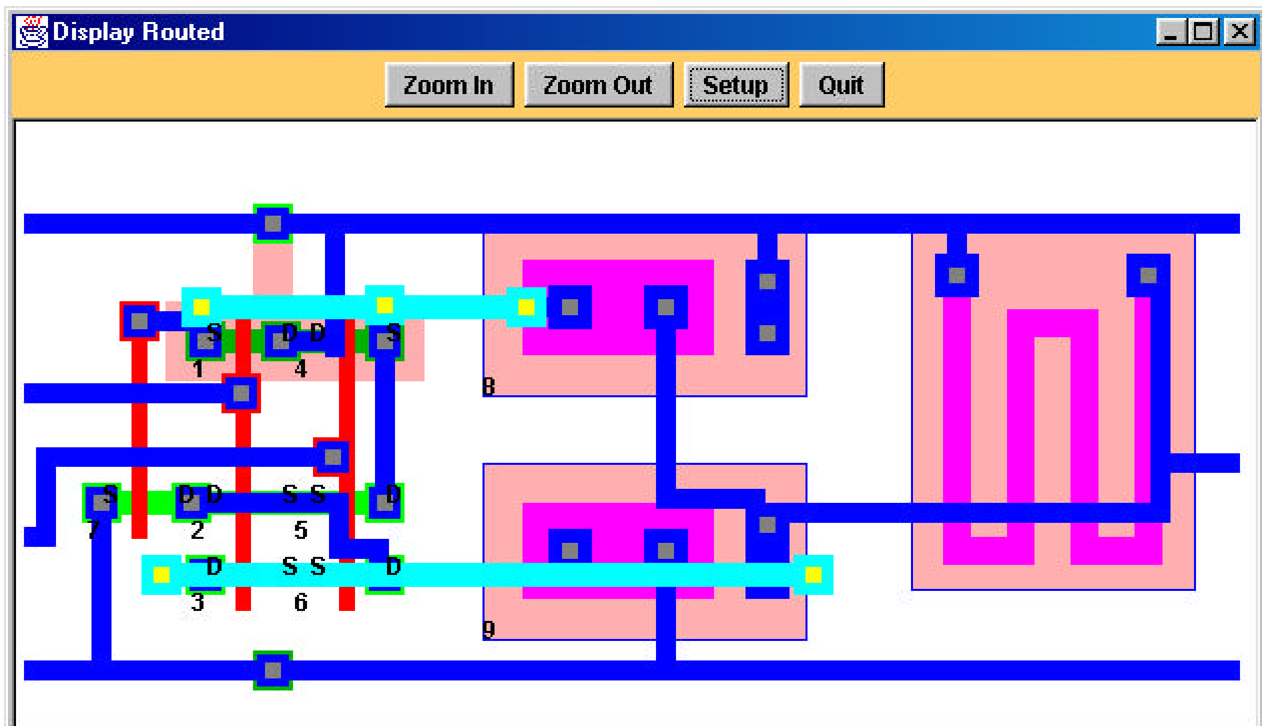


Figura 7.8: Resultado de Roteamento de uma Porta nand BICMOS com célula analógica.

Os gráficos apresentados demonstram que, além de obter escalabilidade em computadores com memória compartilhada, o sistema *Agents* 2 (com o novo roteador) ainda mantém a escalabilidade obtida no sistema original rodando distrbuído. Os resultados dos testes realizados podem variar um pouco a cada execução dos programas devido a fatores ligados as diferenças de configuração dos computadores e uso da rede.

7.4 – Considerações Finais

Este capítulo mostrou e analisou os resultados obtidos nos testes com o servidor *RouterServer*, executado em máquinas isoladas ou distribuído numa rede de computadores. Os testes procuraram cobrir o maior número de possibilidades de execução possíveis. O novo servidor *RouterServer* mostrou-se escalável para computadores com memória compartilhada e fazendo parte do sistema *Agents 2* em ambientes distribuídos.

Apesar do sistema *Agents 2* ainda não poder ser considerado um sistema acabado (ainda existem bugs), ele é um protótipo (ou versão beta) que demonstra que as características esperadas dessa nova versão em Java foram conseguidas. Ou seja, o principal objetivo de desenvolver um software gerador de layouts, similar ao sistema *Agents* original, mas que pudesse ser executado de forma paralela em máquinas com múltiplos processadores foi alcançado.

Capítulo 8

Conclusões

8.1 - Considerações Iniciais

O projeto de circuitos integrados não é uma tarefa trivial. Existem vários detalhes que devem ser levados em considerações. Para evitar problemas referentes ao *designer* a utilização de ferramentas que automatizem a geração de circuitos possui vantagens obvias.

Pensando nessas dificuldades, o sistema *Agents* foi desenvolvido para gerar automaticamente l*ayouts* de células-padrão e facilitar o projeto de circuitos integrados. Baseado no conceito de agentes de softwares, ele era composto por quatro agentes servidores, o *Placer,* o *Router, o Database e o Broker.* Os servidores trabalhavam de forma distribuída em uma rede de computadores e sua performance aumentava a medida que novos computadores eram introduzidos na rede.

O sistema *Agents2* é uma evolução do sistema *Agents.* O sistema inteiro foi portado para a linguagem Java e o servidor de roteamento foi reescrito e renomeado *RouterServer* para explorar recursos de multiprocessamento existentes em computadores com múltiplos processadores e memória compartilhada. Os resultados esperados de escalabilidade em computadores com múltiplos processadores foram alcançados e a escalabilidade em sistemas distribuídos foi mantida.

8.2 – Principais Contribuições

As principais contribuições desse trabalho são:

- Demonstração da utilização de agentes de software para realização de tarefas de forma paralela.
- Paralelização do servidor de roteamento do sistema *Agents 2*, criando o *RouterServer*.
- Obtenção de escalabilidade de processamento em computadores com memória compartilhada para o sistema *Agents 2*.
- Teste geral do sistema *Agents 2* para demonstrar sua escalabilidade em ambiente paralelos e distribuídos.
- Detecção e correção de diversos *bugs* no sistema.

8.3 – Possíveis Trabalhos Futuros

Como sugestões para trabalhos futuros, podemos citar:

- Teste de desempenho do servidor de roteamento *RouterServer* para circuitos integrados analógicos:
  O *RouterServer* é projetado para o roteamento de circuitos digitais sendo, talvez, inadequado para o desenvolvimento de circuitos analógicos. Pode-se realizar um trabalho que estudasse algoritmos de roteamentos que pudessem ser utilizados em circuitos analógicos.

- Atualização da versão do EDIF suportado:
  O EDIF é uma linguagem de especificação utilizada por várias ferramentas de projeto de circuitos eletrônicos. Atualizar o EDIF para a sua versão mais recente facilitaria que outras ferramentas se comuniquem com o sistema *Agents 2*.

- Desenvolvimento de uma versão do sistema *Agents* dedicada totalmente a circuitos analógicos.

  A tecnologia de agentes, com que foi desenvolvido o sistema *Agents2*, pode ser usada para o desenvolvimento de circuitos analógicos, para isso seria necessária a adaptação dos algoritmos do servidor de posicionamento e, provavelmente, do *RouterServer*.

8.4 - Considerações Finais

Este capítulo apresentou as conclusões deste trabalho, bem como suas contribuições para a comunidade e mostrou possíveis trabalhos futuros.

A principal contribuição do *RouterServer* foi permitir a evolução do sistema *Agents* para sua segunda versão: o *Agents 2*. Essa nova versão pode tirar proveito não só do paralelismo existente em computadores distribuídos numa rede, mas também do paralelismo de máquinas com múltiplos processadores e memória compartilhada (que estão se tornando cada vez mais comuns e baratas) sendo também executável em diversas plataformas graças ao uso da linguagem Java. Espera-se que este trabalho abra caminho para outras melhorias do sistema *Agents*.

Referências Bibliográficas

[01] – Moreira, D. A. A Distribuited Client/Server System for leaf Cell Generation, Phd thesis, University of Kent at Canterbury, 1995.

[02] – Enderlein, R. Microeletrônica. Edusp – Editora da Universidade de São Paulo, São Paulo, 1994.

[03] – Hu, T.C.; Kuh, E. S. VLSI Circuits Layout: Theory and Design. IEEE Press, 1985, pp 139-143.

[04] Genesereth, M.R.; Ketchpel, S.P., Software Agents, Communications of the ACM, Vol. 37, n.7, July 1994, pp. 48-53, 147.

[05] - Moreira, D. A. Using Softwares Agents to Generate VLSI Layouts, IEEE Expert, Vol 12, n.6, November/December 1997, pp. 26-32.

[06] – Maes, P. Agents that Reduce Work and Information Overload, Comunication of ACM, Vol 37, n.7, Jul. 1994", pp. 31-40.

[07] – Jansen, J. Using na Inteligent Agent to Enhance Search Engine. disponível *on-line* em http://www.firstmonday.dk/issues/issue2_3/jansen/, acessado em 01/12/200.

[08] – Franklin, S.; Graesser, A. Is it na Agent or Just a Program?: A Taxonomy for Autonomous Agents. disponível on-line em http://www.msci.memphis.edu/~franklin/AgentProg.html, acessado em 01/12/2000.

[09] – Sun Microsystems. The Java Language: An Overview. Disponível *on-line* em URL http://java.sun.com/docs/overviews/java/java-overview-1.html, acessado em 01/12/2000.

[10] – Kramer, D. The Java Plataform. Disponível *on-line* em http://java.sun.com/docs/white/platform/javaplatform.doc2.html, acessado em 01/12/2000.

[11] – Eckel, B. Thinking in Java" , Prentice Hall PTR. New Jerssey, 1998.

[12] – Naighton, P. Dominando o Java, Makron Books do Brasil Editora LTDA, São Paulo 1997.

[13] – Almasi, G.; Gottlieb, A. Highly Parallel Computing, The Benjamin Cummings Publishing Company, 2ª ed., 1994

[14] - Nevison, C. H. Parallel Computing in the Undergraduate Curriculum, IEEE Computer, 1995, December, pp. 51-56.

[15] – Tanembaum, A. S. Sistemas Operacionais Modernos, Editora Prentice-Hall do Brasil LTDA. 1995

[16] – Scott, O.; Wong H. Java Threads. O'Reilly & Associates, Inc., 1997.

[17] – Moreira, D. A. Operation System 2 Course, disponível *on-line* em http://java.icmc.sc.usp.br/os2_course, acessado em 01/12/2000.

[18] – Cornell, G; Horstmann, C.S. Core Java, Makron Books do Brasil Editora LTDA. São Paulo, 1995.

[19] – Lillelea, P. Java threads may not use all your CPUs, disponível *on-line* em URL: http://www.javaworld.com/jw-08-2000/jw-0811-threadscale.html, acessado em 11/10/2000

[20] – Holub, A. Programming Java threads in the real world, Part 2. disponível *on-line* em URL: http://www.javaworld.com/javaworld/jw-10-1998/jw-10-toolbox.htm, acessado em 11/10/2000

[21] Moore, E.F. Shortest Path Through a Maze. Annals of the Computation Laboratory of Havard University, Havard Univ. Press. Cambridge Mass., vol.30, 1959, pp. 285-292.

[22] Arnold, M.H., Scott, W.S. An Interactive Maze *Router* with Hints. Proc. Of the 25th ACM/IEEE Design Automation Conference, 1988 IEEE, pp. 672-676.

[23] Pucknell, D.A; Eshraghian, K. Basic VLSI Design. Third Edition. Prentice Hall, London, 1994.