TRABALHO DE IMPLEMENTAÇÃO DO PROBLEMA *FLOORPLAN DESIGN*

(por Luiz Marcio F A Viana)

1. INTRODUÇÃO

Este trabalho de implementação tem como objetivo complementar o curso de programação concorrente, introduzindo conhecimentos práticos de programação paralela através do desenvolvimento de cinco versões paralelas do problema do *Floorplan Design*. Cada implementação utiliza recursos de *software* e *hardwares* diferentes, sendo três delas desenvolvidas para máquinas de memória compartilhada centralizada utilizando **POSIX *Threads***, **IPC *Share Memory*** e a linguagem **SR**, e duas para sistemas de memória compartilhada distribuída utilizando **PVM** e **ThreadMarks**.

Na próxima seção descreveremos o problema do *Floorplan Design*. A terceira seção apresenta uma solução para este problema e a quarta seção descreve a implementação das cinco versões construídas. A última seção apresenta as conclusões obtidas com o trabalho.

2. *FLOORPLAN*: DESCRIÇÃO DO PROBLEMA

O processo de confecção de circuito integrado atravessa várias etapas. Entre elas está a fase de *lay out* onde os componentes são posicionados dentro da pastilha.

A fase de *lay out* é dividida em três partes. Na primeira parte são construídos conjuntos de blocos retangulares indivisíveis denominados células onde cada conjunto representa diferentes instâncias de implementação de um dos componente do circuito. Em seguida são estabelecidas as posições relativas de cada componente informando as conexões existentes entre eles. Na terceira fase são analisadas as possíveis combinações de células até encontrar a combinação de instâncias que requer área mínima de pastilha.

O problema do *Floorplan Design* esta relacionado ao último passo do processo de *lay out* de circuitos integrados e consiste na determinação das combinações de instâncias de cada componente que minimizam à área de pastilha. Para isso é necessário o conhecimento do conjunto de instâncias de cada componente e a posição relativa entre eles. Na figura a seguir (**figura 1**) são apresentados três componentes com as respectivas instâncias.

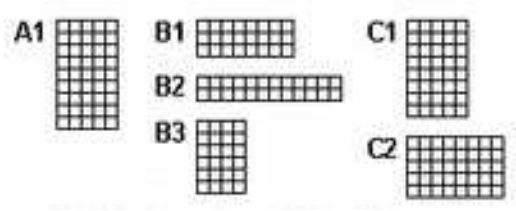

figura 1: Três conjuntos de células, cada conjunto representa um componente.

Podemos ver na **figura 1** que o componente **A** possui apenas uma instância (**A1**) enquanto o componente **B** possui três (**B1**, **B2** e **B3**) e o componente **C** possui duas (**C1** e **C2**).

O posicionamento relativo dos componentes pode ser representado por meio de dois grafos acíclicos que possuem apenas um nó fonte e um nó destino. O primeiro deles é denominado grafo **G** e estabelece a posição relativa dos componentes na direção vertical, enquanto o outro, grafo **H**, descreve o posicionamento horizontal. A **figura 2** mostra um exemplo de par de grafos de posicionamento.

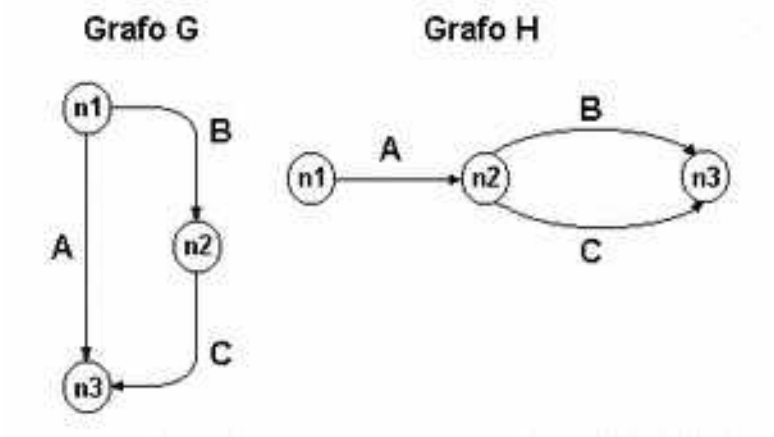

figura 2: Grafos de posicionamento relativo entre os componentes.

Nestes grafos as arestas representam os componentes enquanto os nós representam a face de contato e a comunicação entre eles. A partir do conhecimento das instâncias de cada componente e dos grafos de posicionamento a computação pode ser realizada com a avaliação de todas as combinação das instâncias dos componentes a procura da que requer menor área de pastilha.

Como podemos perceber este é um problema combinatorial onde o espaço de busca cresce exponencialmente com o número de componentes exigindo que muita computação seja feita até que se conheça o resultado. Também é importante ressaltar que este problema pode gerar mais de uma

solução, porque mais de uma combinação de instâncias podem requerer o mesmo valor de área mínima de pastilha.

Nesta seção descrevemos o problema do *Floorplan Design*. Na próxima seção apresentaremos uma solução para ele.

3. *FLOORPLAN*: APRESENTAÇÃO DA SOLUÇÃO

Podemos adicionar alguns símbolos aos grafos de posicionamento de modo à simplificar a compreensão das informações que eles armazenam. No grafo **G** adicionaremos os símbolos **T** (de topo) à origem de cada aresta e **B** (de base) à extremidade final delas e o grafo **H** será modificado com a adição dos símbolos **E** (de esquerda) para à origem e **D** (de direita) para o final de cada aresta. A **figura 3** apresenta os grafos de posicionamento após a modificação com a adição destas informações.

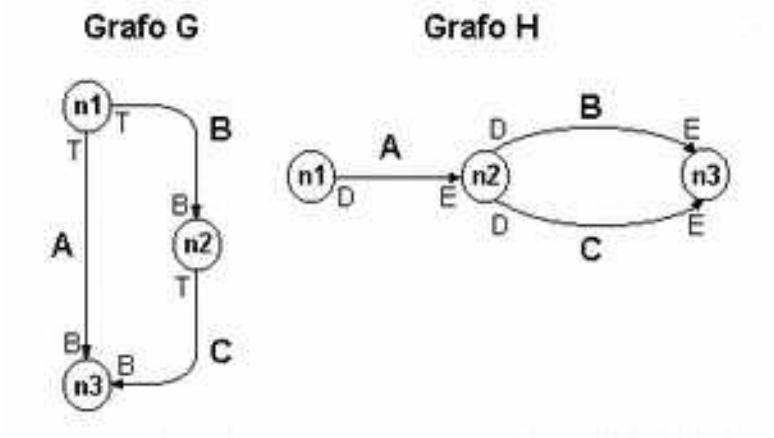

figura 3: Grafos de posicionamento após adição das informações auxiliares.

Observando os grafos de posicionamento após as modificados conseguimos extrair informações visuais adicionais que permitem compreender melhor o problema. Estes grafos agora informam claramente como as faces de cada célula se correspondem e assim é possível determinar as regiões em que cada componente esta contido. A **figura 4** mostra estas regiões para o arranjo definido pelos grafos.

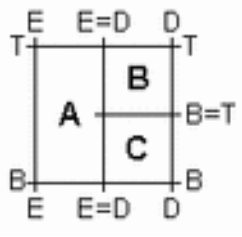

figura 4: Regiões de ocupação dos componentes.

Através da **figura 4** podemos perceber que o posicionamento de cada componente se encontra bem definido em relação aos componentes em sua volta. Isto é, o componente **A** se localiza à esquerda de **B** e **C**, enquanto **B** está sobre **C**.

Observe que as regiões apresentadas são esquemáticas e fornecem apenas uma idéia de posicionamento relativo entre os componentes.

Na figura abaixo (**figura 5**) apresentamos todas as combinações possíveis de instâncias dos componente sobre à região definida na **figura 4**.

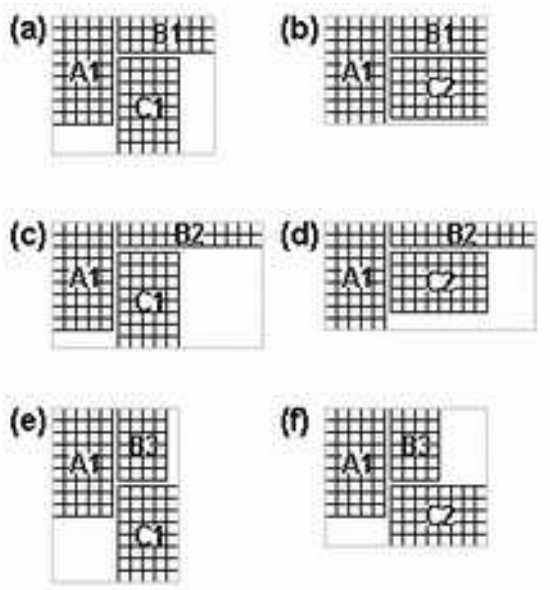

figura 5: As combinações das instâncias aplicadas às regiões.

Observando as possíveis combinações apresentadas na **figura 5** percebemos que a área a ser calculada é o produto entre a maior dimensão horizontal e a maior dimensão vertical. Por exemplo, na **figura 5a** a área resultante será o produto entre a soma das larguras de **A1** e **B1** (13 un), com a soma das alturas de **B1** e **C1** (11 un), produzindo uma área total requerida de 143 un.

A maior dimensão vertical pode ser obtida à partir do grafo de posicionamento **G**, percorrendo-o à procura do caminho crítico para ir do nó fonte ao nó destino. De forma análoga a maior dimensão horizontal é obtida percorrendo o grafo **H** à procura do caminho crítico para ir de um extremo à outro. Uma vez que obtemos os caminhos críticos nos grafos para uma combinação de instâncias podemos calcular à área requerida fazendo o produto dos valores obtidos. Desta forma a solução do problema do *Floorplan Design* se reduz ao problema de pesquisa do caminho crítico em um grafo para cada combinação de instâncias dos componentes.

Nesta seção foi apresentada uma solução para o problema do *Floorplan Design*. Na próxima seção mostraremos como foram realizadas às implementações desta solução para cada versão do programação.

4. IMPLEMENTAÇÕES PARALELAS

Para facilitar a implementação paralela do problema foram construídas duas versões seqüenciais com diferentes graus de otimização e a partir destes modelos seqüenciais é que foram construídos os programas paralelos.

Uma vantagem de possuir o modelo seqüencial é que os resultados obtidos pelos modelos paralelos puderam ser comparados facilitando o diagnóstico de problemas. A outra vantagem se caracteriza pelo aproveitamento do código que agilizou bastante o trabalho.

A implementação do problema consiste basicamente de duas etapas. Na primeira etapa são realizadas leituras aos arquivos de definição de células e de definição dos grafos. Enquanto a segunda etapa é responsável pelo processamento propriamente dito.

O ponto chave da implementação são as definições das estruturas de dados que armazenam as informações das células e dos grafos de posicionamento. Estas estruturas se relacionam de forma à evitar ao máximo a redundância de informações e a necessidade de pesquisa por dados.

Na **figura 6** apresentamos a estrutura de armazenamento das células e grafos de posicionamento.

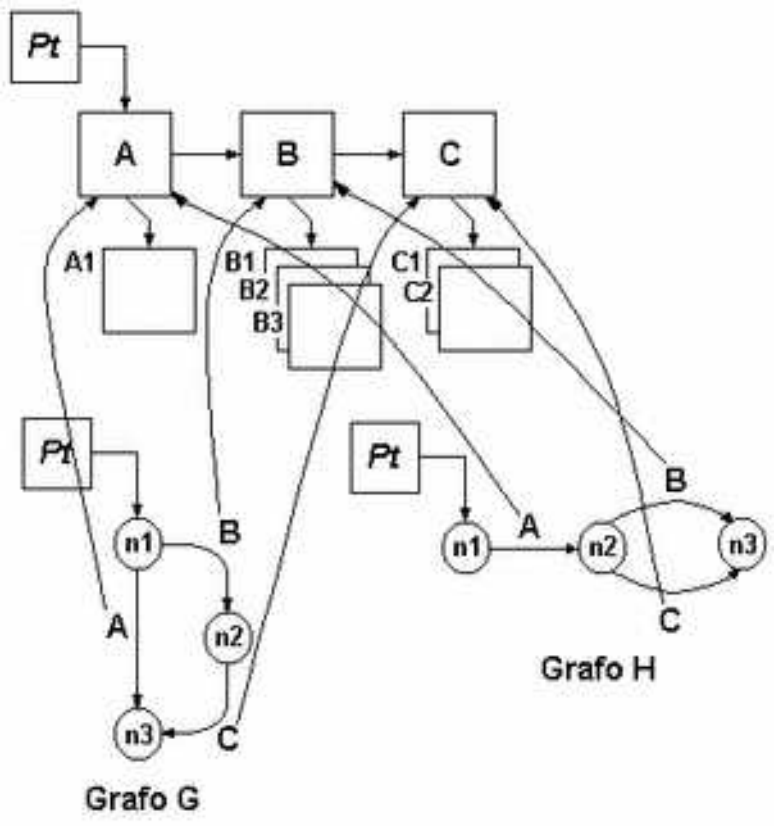

figura 6: Estruturas de dados básicas do programa.

Conforme pode ser observado na **figura 6** a estrutura básica do programa consiste de uma lista encadeada dos componentes, onde cada componente possui um vetor com todas as instâncias assumidas por ele. Desta forma os grafos são construídos armazenando em cada aresta ponteiros para o elemento correspondente na lista. Esta otimização elimina o tempo de pesquisa pelas instâncias das células durante o calculo do caminho crítico no grafo.

Estas estruturas são visíveis à todos os nós de processamento, sendo que nas implementações com **IPC *Share Memory***, **TreadMarks** e **PVM** cada nó de processamento é responsável por inicializa-las.

Após a criação das estruturas inicia-se a etapa de processamento. Nesta etapa são criadas as tarefas responsáveis por resolver o problema.

A solução paralela consiste na implementação de um modelo mestre-escravo onde cada nó de processamento acessa um processo mestre que é responsável por distribuir tarefas. Este conceito só é diferente para as aplicações que implementam memória compartilhada pois elas possuem acesso direto à estrutura de gerenciamento de tarefas sendo o controle efetuado de forma descentralizada.

Na listagem à seguir (**listagem 1**) apresentamos o pseudocódigo base para todas as implementações do programa.

```
1. Lê o arquivo de células;
2. Lê o grafo de posicionamento GRAFO_H;
3. Lê o grafo de posicionamento GRAFO_G;
4. Cria os NPROC processos paralelos;
      1. Em paralelo cada processo avalia a área mínima;
            1. Enquanto existirem combinações à processar;
                  1. Obtém combinação de instâncias corrente;
                  2. Avalia largura da área de integrado;
                  3. Avalia altura da área de integrado;
                  4. Calcula à área requerida;
                  5. Se área requerida < que área mínima local;
                        1. atualiza o valor da área mínima local;
                        2. armazena as instâncias utilizadas;
6. Processo mestre aguarda pelo término das tarefas;
7. Processo mestre seleciona a menor das áreas locais;
8. Processo mestre apresenta o resultado.
```

listagem 1: Listagem do pseudocódigo base dos programas.

Nas subseções seguintes analisaremos os detalhes de implementação e problemas encontrados em cada versão do programa.

4.1. THREADS

A implementação com tarefas foi imediata pois a implementação seqüencial já possuia os procedimentos bem definidos sendo necessário apenas adicionar os mecanismos de controle das *threads* e os mecanismos de sincronização.

Threads é um recurso muito simples de implementar e que poderia ser empregado por todas as versões do programa. Ficando responsáveis pelo processamento de inicialização das estruturas de dados, distribuição do processamento entre os processos em **PVM** e **TreadMarks**, e pela computação propriamente dita quando a granularidade das tarefas aumenta gerando um *pool* de *threads* responsáveis pelo conjunto de tarefas.

4.2. IPC (*Share Memory*)

A utilização do mecanismo de compartilhamento de memória através do **IPC** não fornece ao programador a simplicidade encontrada na implementação utilizando tarefas sendo seu emprego atraente apenas quando o compartilhamento de memória é realizado por aplicações diferentes e mesmo nestes casos a utilização de tarefas para otimizar o processamento interno é aconselhável.

O desenvolvimento utilizando **IPC** modifica um pouco a estrutura do programa e o armazenamento das estruturas de dados, mas foi muito útil pois seu código pode ser reaproveitado mais tarde na versão em **TreadMarks**.

4.3. PVM

Optei pela implementação com **PVM** e não **MPI** pelas dificuldades de instalação desta última. A dificuldade foi provocada por incompatibilidade entre as versões do compilador e *kernel* do sistema operacional **Linux** com a biblioteca.

PVM é uma biblioteca de troca de mensagens para o desenvolvimento de aplicações paralelas distribuídas. A implementação com **PVM** produz uma grande mudança na forma como os dados são armazenados que passaram a ser tratados como vetor para facilitar a comunicação entre os nós.

Nas implementações com memória distribuída o grande gargalo está nos processos que não podem mais gerar tarefas sozinhos e precisam requisita-las a um processo mestre. Este processo mestre é responsável por enviar um vetor contendo as instâncias que serão utilizadas para o processamento.

Nesta implementação a utilização de *threads* otimizaria o processamento adicionando múltiplas portas de comunicação entre as aplicações e várias tarefas de processamento permitindo maior granularidade à aplicação que poderia trabalhar com um *pool* de tarefas ao invés de uma.

4.4. TREADMARKS

A versão de **TreadMarks** analisada foi desenvolvida para o *kernel* 1.2.13 do **Linux** e gerou alguns problemas de compatibilidade com a versão de *kernel* utilizada 2.0.33. As principais modificações foram feitas nas bibliotecas de comunicação com *sockets*.

A implementação com **TreadMarks** é bastante similar a implementação com **IPC** e o código anterior pode ser reaproveitado. Apesar de **TreadMarks** fornecer um pouco de simplicidade as

aplicações distribuídas esta simplicidade não supera o desempenho obtido por este mecanismo. O *overhead* de comunicação e requisitos de sistemas exigidos degradam muito o desempenho do sistema.

4.5. SR: MEMÓRIA COMPARTILHADA

A dificuldade encontrada em **SR** foi motivada por não existir documentação detalhada da linguagem e a implementação para **Linux** foi realizada através de adaptação do compilador.

SR é uma linguagem bastante semelhante à linguagem Pascal e possui recursos para processamento de instruções em paralelo e para criação de múltiplas tarefas para processamento paralelo. Esta linguagem também possui características de orientação à objetos través da definição de recursos de processamento.

O compilador da linguagem permite a construção de código executável ou código intermediário em **C** sendo esta uma linguagem de programação leve que possui os recursos essenciais para a exploração de paralelismo de forma distribuída por troca de mensagens ou através de memória compartilhada centralizada.

5. CONCLUSÕES

Embora não tenha sido feita nenhuma comparação de desempenho entre as diferentes versões do programa o trabalho proporcionou conhecimento prático de desenvolvimento das aplicações paralelas. As dificuldades encontradas na adaptação do código paralelo entre as diferentes implementações e os problemas de sincronização enfrentados nas aplicações foram de grande valor como complemento as aulas de programação concorrente.

Pela comparação das implementações pudemos avaliar as características de cada mecanismo destacando a utilização de múltiplas tarefas como fundamental para otimizar o processamento de qualquer aplicação paralela ou não. O mecanismo de **IPC** embora não tenha a simplicidade das tarefas ele ainda é muito útil na comunicação entre processos diferentes. A implementação com troca de mensagem utilizando **PVM** simplificou bastante a comunicação entre os processos distribuídos, embora as operações de redução centralizadas provocassem um gargalo na aplicação. Desta forma muitas otimizações poderiam ser realizadas com a aplicação direta de mecanismos de comunicação entre as tarefas.

TreadMarks é uma forma bastante interessante de implementação de aplicações distribuídas embora o *overhead* de controle e gerenciamento dos dados e processos seja muito grande. Eu

considero os mecanismos de desenvolvimento baseados em *software* **DSM** promissores mas muito trabalho ainda precisa ser feito nesta área.

SR é completamente diferente dos outros mecanismos sendo uma linguagem de programação e não uma biblioteca. Desta forma **SR** consegue reunir muitos recursos interessantes para o desenvolvimento de aplicações paralelas. Esta linguagem também introduz algumas das características de programação orientada a objeto que se exploradas em maior grau fornecerão um potencial muito grande à esta linguagem.