

Monografia de Programação Concorrente

Paralelização Automática

Profª Dra. Regina Helena Carlucci Santana

Debora Maria R. de Medeiros n° USP 3309225

Diego H. de Campos n° USP 3280840

Giampaolo L. Libralão n° USP 3280854

Murilo Coelho Naldi n° USP 3280990

Thiago Fernando Alves

1. Introdução

O objetivo da paralelização automática é esconder o paralelismo do programador, ou seja, livrar o programador do trabalho de desenvolver soluções paralelas para um problema. Para isso, são incorporadas a compiladores técnicas que detectem:




- Possibilidades de paralelismo
- Transformações que permitam a paralelização

O princípio básico para possibilidade de paralelismo é que dois comandos só podem ser executados em paralelo se eles produzirem os mesmos resultados se forem executados em qualquer ordem. Para isso, as entradas para um comando não podem ser dependentes de alguma saída de outro comando. E uma variável de saída de um comando não pode ser a mesma de saída de outro, pois eles modificam a mesma variável.

2. Dependência de Dados e Grafo de Dependências

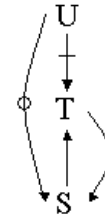
A dependência de dados representa a influência de referências à memória na estrutura de dependências do programa. O grafo de dependências ilustra a dependência de dados entre os comandos

A dependência de dados pode ser de três tipos:

- Dependência de fluxo: ocorre quando a entrada de um comando depende da saída de outro. No grafo de dependências é representada por uma seta.

- Antidependência: um comando é antidependente de outro se em alguma execução do programa, um deles modifica uma variável previamente lida pelo outro. É representada por uma seta cortada.

- Dependência por saída: ocorre quando dois comandos possuem uma mesma variável de saída. É representado por uma seta com uma bolinha.


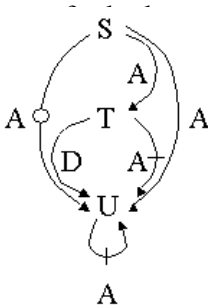
Exemplo de grafo de dependência:

```
Para i=2..200
  S: A[i] = B[i] + C[i]
  T: B[i+2] = A[i-1] + C[i-1]
  U: A[i+1] = B[2i+3] + 1
Fim Para
```



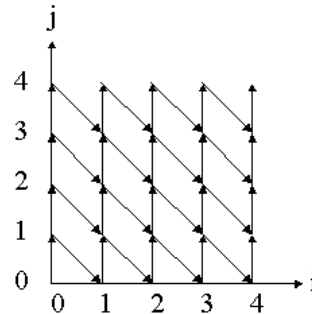
Exemplo
, laço dup

```
Para i=C
  Pa
  B[i]
  Fi
Fim Para
```



ncia entre iterações

$$= A[i][j+1]+1$$



3. Dependência de controle

Representa a influência da estrutura de controle do programa na estrutura de dependência. O grafo de dependência de controle contém apenas um *sink* que é um nó de onde não há setas saindo.

Nesse tipo de dependência, diz-se que um nó Y pós-domina um nó X se todos os caminhos de X para o *sink* incluem o Y, ou seja, para chegar no *sink* a partir de Y, é necessário passar por X. Também, um nó T é dependente de controle de S se há um caminho de S para T cujos nós internos são pós-dominados por T e T não pós-domina S., ou seja, é necessário passar por S antes de chegar em T.

É possível transformar dependência de controle em dependência de dados, como no exemplo abaixo:

```
Se A != 0
  C = C+1
  D = C/A
Senão
  D = C
Fim Se
X = C + D
```

```
b = [A != 0]
C = C+1 quando b
D = C/A quando b
D = C quando não b
X = C+D
```

4. Métodos para Eliminar Dependências

Existem alguns métodos que aplicam pequenas modificações no código, do programa sequencial, para eliminar algumas dependências.

O método **Renaming** atribui diferentes nomes para usos diferentes de uma mesma variável, como está acontecendo com a variável A no exemplo abaixo:

```
S: A = B + C
T: D = A + E
```

```
S': A1 = B + C
T': D = A1 + E
```

U: A = A + D

U' : A = A1 + D

O método **Forward Substitution** copia o lado direito de uma atribuição no lado direito de outra atribuição, a fim de eliminar o uso de uma variável, como no exemplo abaixo:

<p>T: D = B + C + E U: A = B + C + D</p>		<p>T' : D = B + C + E U' : A = (B + C) + (B + C + E)</p>	
		<p>T'</p> <p>U'</p>	

O método **Scalar Expansion** consiste em transformar uma variável utilizada dentro de um *loop* em um *array*, como no exemplo abaixo:

<p>Para i=1..N faça S: X = C[i] T: D[i] = X + 1 Fim para</p>		<p>Para i=1..N faça S: X[i] = C[i] T: D[i] = X[i] + 1 Fim para</p>	
		<p>S'</p> <p>T'</p>	

5. Método para Detecção de Dependências

Para detectar, em um *loop* quais iterações são dependentes de quais, vamos considerar o seguinte código:

```
Para I=2..200
  S: X[3I-5] = B[I]+1
  T: C[I] = X[2I+6]+D[I-1]
Fim Para
```

As iterações em que o comando S e T acessam a mesma posição do vetor X, considerando i o contador para S e j o contador para T, devem seguir a seguinte relação:

$$3i-5 = 2j+6$$

$$3i-2j = 11$$

Então, calcula-se o máximo divisor comum entre 3 e 2 (números que multiplicam i e j), m que é igual a 1. A solução se dá pela seguinte equação:

$$(i,j) = ((2/m)t+i_1, ((3/m)t+j_1)$$

onde t é qualquer inteiro, parâmetro m e:

$$(i_1,j_1) = ((6-5)i_0/m, (6-5)j_0/m)$$

onde 6 e 5 são os números somados para calcular a posição do vetor X nos comandos S e T, i₀ e j₀ são dois inteiros que satisfazem:

$$3i_0 - 2j_0 = m$$

e neste caso podem ser iguais a 1. Então a solução geral resulta em:

$$(i,j) = ((2/1)t+11, ((3/1)t+11)$$

Sabemos que i e j estão limitados entre 2 e 200 e, portanto:

$$2 \leq 2t+11 \leq 200 \Rightarrow -4 \leq t \leq 94 \Rightarrow \text{Intersecção}$$

$$2 \leq 3t+11 \leq 200 \Rightarrow -3 \leq t \leq 63 \quad -3 \leq t \leq 63$$

O ponto de intersecção entre $i(t) = 2t + 11$ e $j(t) = 3t + 11$ é em $t=0$, então para valores de t tal que $1 \leq t \leq 63$ $i(t)$ será menor que $j(t)$, e para os valores $t = -3, -2$ e -1 , $i(t)$ será maior do que $j(t)$.

Então T é dependente de S nas seguintes iterações:

$$\{(S(2t+11), T(3t+11)): 0 \leq t \leq 63\} = \{(S(11), T(11)) \dots (S(137), T(200))\}$$

E S é antidependente de T nas seguintes iterações:

$$\{(T(3t+11), S(2t+11)): -3 \leq t \leq -1\} = \{(T(2), S(5)), (T(5), S(7)), (T(9), S(9))\}$$

6. Transformações Serial-Paralela

Baseado nas análises de dependência (de dados ou controle), é possível empregar algumas técnicas de transformações do programa, para aumentar o paralelismo que aparece indiretamente no mesmo (dito intrínseco).

Duas são as classes de transformação possíveis, a primeira é a paralelização de código acíclico e a segunda a de laços DO.

6.1 Códigos acíclicos

Os códigos acíclicos não apresentam ciclos no grafo de fluxo do programa (ou trecho paralelizável), sendo aplicado de acordo com o nível de granulação desejado e a arquitetura destino. As instruções são divididas em blocos conforme a granulação desejada e pela avaliação das dependências determina-se quais instruções ou blocos destas serão executados paralelamente, podendo ser empregadas as construções COBEGIN/COEND e as primitivas de sincronização POST/WAIT.

A figura 6.1 indica um exemplo de códigos acíclicos

```

S1 : A = 1
      cobegin
S2 :   B = A + 1
      post (e)
S3 :   C = B + 1
      ||
S4 :   D = A + 1
      wait (e)
S5 :   E = D + B
      coend

```

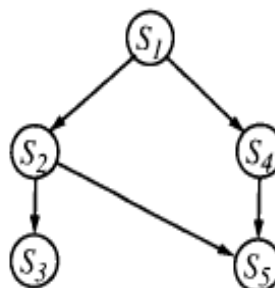


Figura 6.1: *Exemplo de Códigos Acíclicos.*

6.2 Métodos de definição da ordem de execução (scheduling)

Apesar da simplicidade da geração de código paralelo, a definição da ordem de execução (*scheduling*) é extremamente complexo e se enquadra como um processo de otimização combinatória. A seguir são citados alguns métodos heurísticos que auxiliam neste processo.

6.2.1 Paralelização pela granulação

Varia de acordo com a arquitetura paralela empregada. Geralmente, pode-se empregar uma granulação mais grossa para posterior balanceamento de carga, em tempo de execução, entre os processadores utilizados.

6.2.2 Compactação de código

Nesta é empregado o conceito de macronós (estruturas que contém um número qualquer de operações lógico-aritméticas e são delimitadas por COBEGIN/COEND). Esta técnica é importante quando do uso de processadores superescalares e VLIW (*very large instruction word*).

O princípio consiste na independência de um macronó perante os demais, sendo esta condição garantida pelo emprego apenas de instruções de atribuição (valendo-se das operações matemáticas já citadas). Uma das componentes do macronó é uma árvore de instruções condicionais (*if*) e instruções de salto (*go to*), determinando o próximo macronó a ser executado, o que garante a manutenção da dependência de controle e/ou de dados. Este salto somente é realizado quando as demais instruções da macronó tenham sido realizadas.

Procura-se evitar a atribuição de uma variável duas ou mais vezes num mesmo macronó, assim como a dependência de variáveis entre macronós.

6.2.3 Trace Scheduling

É uma técnica de paralelização bastante refinada. Para realizar a paralelização é importante existir um grande conjunto de instruções que sejam executadas sequencialmente para identificação das dependências, o agrupamento destas em blocos e posterior execução em paralelo.

Um dos problemas encontrados é o dos saltos condicionais, pois estes dificultam a geração de um alto grau de paralelismo entre os blocos de instruções. Esta técnica avalia a probabilidade de execução dos saltos, pré-avaliando os e obtendo um longo caminho de instruções que pode ser paralelizável, considera-se que a maioria dos saltos em um programa é preditível.

Em caso de falha na avaliação são especificadas condições de contorno que estão relacionadas com um salto não corretamente predito. De modo a garantir uma recuperação destes saltos, os códigos do programa original são colocados juntamente ao código paralelo sempre que este tipo de movimento de execução ocorra.

Como uma alternativa a esta técnica é o emprego de técnicas de fluxo de dados (em pequena escala) para o despacho dinâmico das instruções, ao passo que é realizado uma reorganização de pequenos blocos de instruções para elevar o número de instruções independentes (paralelizáveis).

6.2.4 Filtragem de *Scheduling*

Baseada em três transformações elementares (*moveop*, *movecj* e *unify*), envolvendo macronós próximos (aqueles com relação pai-filho).

Transformação moveop: desloca uma atribuição de um macronó para o macronó anterior, no grafo de fluxo de controle, considerando as dependências de dados.

Transformação movecj: desloca qualquer subárvore de uma árvore condicional de um macronó para o macronó anterior.

Transformação unify: efetua a movimentação para um macronó pai de instruções repetitivas em seus macronós sucessores.

Exemplos:

Moveop:

<pre> M: cobegin S₁ ... S_n if goto N coend </pre>	\Rightarrow	<pre> M': cobegin S₁ ... S_n S'_i if goto N' coend </pre>
--	---------------	--

Unify

<pre> M: cobegin S₁ ... S_n if goto N₁ goto N_m coend </pre>	\Rightarrow	<pre> M: cobegin S₁ ... S_n X_{n'} if goto N₁ goto N_m coend </pre>
<pre> N₁: cobegin S'₁ ... S'_{n'} X if ... coend </pre>		<pre> N₁: cobegin S'₁ ... S'_{n'} if ... coend </pre>
<pre> ... </pre>		<pre> ... </pre>
<pre> N_m: cobegin S''₁ ... S''_{n''} X if ... coend </pre>		<pre> N_m: cobegin S''₁ ... S''_{n''} if ... coend </pre>

Nas técnicas de transformação por *Trace Scheduling* e por *Filtragem de Scheduling* as informações de dependência são empregadas apenas à medida que as transformações necessitem. Isto produz um inter-relacionamento entre estas duas técnicas possibilitado seu uso conjunto em compiladores paralelos.

6.3 Laços Heterogêneos

As técnicas para dependência uniforme para laços DO podem ser classificadas em dois grupos distintos: as que geram código heterogêneo (códigos paralelos em que os códigos seriais são distintos) e as que geram código homogêneo (nos quais os códigos seriais são idênticos).

Serão discutidos dois casos para geração de *código heterogêneo*. No primeiro as instruções no corpo do laço são paralelizadas considerando suas dependências. Em laços nos quais as instruções são independentes, cada uma delas é executada em paralelo, no entanto, no caso de dependência de dados é necessário alterar os laços para obter estruturas independentes.

Sendo assim, a técnica de *skewing* pode ser empregada na qual os índices do laço são modificados, permitindo que o código paralelo opere de tal forma que o uso de acessos à memória sejam realizados com o deslocamento espacial para garantir a independência das instruções. Abaixo segue um exemplo desta técnica.

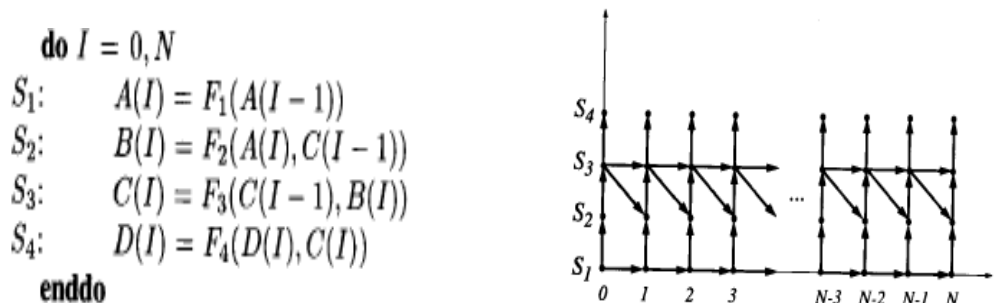


Figura 6.2: Exemplo de *Skewing*.

Outra modificação no laço para permitir o paralelismo das instruções baseia-se em um desdobramento parcial (*partial enrolling*) do próprio laço. Neste método, pode-se repetir o corpo do laço em um número de vezes igual às distâncias de dependência entre os acessos à

memória, estes blocos repetidos podem ser então executados em paralelo. Para permitir uma ampliação do paralelismo alcançado, pode ser empregado uma substituição à frente (*forward substitution*), ou seja, considerando uma instrução de atribuição $v = expressão$, pode-se substituir toda ocorrência de v por *expressão*.

Uma outra maneira de alterar o laço para produção de código paralelo heterogêneo é empregando *pipeline de software*, que visa transformar os laços DO em código paralelo de fina granulação. Existe uma variante chamada *pipeline perfeito*, que corresponde a um mecanismo de *greedy*, neste os macronós sem dependência de dados/controle são executados primeiro, seguidos pelos macronós que tenham dependência direta, e assim sucessivamente. Vale lembrar que, para os casos em que não existem instruções condicionais, o código gerado é ótimo (mais eficiente).

No segundo caso, para a produção de código paralelo heterogêneo é aplicada, sobre laços múltiplos, a técnica chamada *distribuição de laço*, caracterizada pela execução em paralelo de laços internos. Neste sentido, a existência de dependência de dados implica na utilização de instruções de sincronização.

6.4 Laços Homogêneos

Quando são considerados laços DO paralelos, é possível a construção de blocos sequenciais homogêneos, sendo que podem ou não existir dependências de interação cruzada.

No caso de não existirem, as instruções podem ser executadas em paralelo. Se existirem, estas devem ser tratadas, três são as técnicas que serão citadas que empregam transformação do código sequencial em paralelo.

A primeira envolve a separação de parte do código em que não haja dependência cruzada de dados e sua paralelização direta, enquanto as demais partes continuam executando de forma sequencial. Os compiladores e tradutores empregam reconhecimento de padrões de instrução, reduções e outras técnicas de recorrência linear para aplicar este tipo de transformação a códigos sequenciais. Outro método proposto é o emprego da instrução PARALLEL DO e elementos de sincronização para manter a dependência de dados. Nota-se que as duas técnicas mencionadas tem a performance relacionada com a organização do fluxo de execução dos macronós.

Outra maneira para evitar toda a dependência cruzada de dados em alguns macronós é utilizar a replicação (cópia) de algumas instruções, mas isso exige pequenas modificações nos índices de acesso à memória (wait and post).

A última técnica seria particionar os laços paralelos homogêneos em múltiplos outros que serão executados em paralelo. Estudos indicam que o máximo divisor comum (m.d.c.) entre as distâncias de dependência corresponde ao número máximo de partições possíveis para os sublaços rodarem de modo paralelo.

As técnicas mencionadas envolviam o uso apenas de um único laço contendo macronós paralelos, porém para o caso de existirem múltiplos destes, um conjunto adicional de técnicas pode ser empregado.

A primeira transformação seria a troca na ordem de execução dos múltiplos laços, possibilitando uma maior eficiência na computação paralela. A avaliação da ordem pode ser realizada através da análise do grafo de fluxo do bloco a ser transformado. O uso de *skewing* também é uma alternativa.

Considerando que alguns laços podem ter a ordem de execução alterada, é possível reverter a ordem de acesso à memória na instruções internas aos laços (*reversão*), esta técnica

permite que alguns laços múltiplos possam ser paralelizados pois possibilita eliminar dependência de dados nas instruções dos laços.

Vale lembrar que todas estas técnicas podem ser aplicadas em conjunto, de acordo com as características de cada porção de código, possibilitando ao código final paralelo uma maior eficiência que o código original serial.

6.5 Localidade e Overhead de Processadores

Existem algoritmos seriais de difícil paralelização, neste caso, ao invés de aplicar as técnicas mencionadas, são empregadas outras modalidades de transformações em que se minimizam o problema do *overhead* e da falta de localidade das subsoluções.

Para minimizar o *overhead* é utilizada a *fusão de laços*, pois somente um laço paralelo será disparado, além do que a localidade da solução é incrementada. O balanceamento de carga também se torna mais simples, já que um número menor de tarefas é executado em paralelo. Neste mesmo âmbito, a técnica de *colapso de laços* visa reduzir laços múltiplos a um único, resultando nos ganhos de localidade e redução do *overhead* quando transformados em algoritmos paralelos.

Outra técnica conhecida é o *tiling*, ou *strip-mining* para o caso de um único laço. Seu objetivo é possibilitar a geração de elementos paralelos, permitindo a distribuição destes entre os processadores.

O *tiling* permite a redução do sincronismo através de instruções de bloqueio condicional e melhoramento da localidade (na tentativa de manter a independência entre os macronós), além de explorar vários níveis de granularidade no paralelismo. A figura 6.3 tem um exemplo desta técnica.

```

                                do  $I_1 = J_1, \min(J_1 + IB - 1, N)$ 
                                  do  $I_2 = J_2, \min(J_2 + IB - 1, N)$ 
S1:       $B(I_2, I_1) = F_1(A(I_1, I_2))$ 
                                  enddo
                                enddo
                                enddo
                                enddo
                                enddo

```

Figura 6.3: Exemplo da técnica de *tiling*.

6.6 Dependências Cruzadas

A fim de reduzir ou mesmo eliminar as dependências cruzadas de dados, diversas técnicas foram desenvolvidas, algumas delas serão abordadas neste tópico.

Pode-se empregar o conceito de variável de indução. Neste sentido, uma determinada variável é identificada em um conjunto de laços e eliminada do mesmo, pois torna-se invariante nos laços através de uma análise de dependência. Uma vez identificada e eliminada dos laços, o macronó resultante disto pode se tornar paralelo pela eliminação da dependência cruzada.

Uma outra maneira é redefinir algumas variáveis ou elementos n-dimensionais de tal forma que estas sejam acessadas antes de serem modificadas na mesma iteração. Estes elementos

são removidos através do emprego de elementos temporários, eliminando-se assim as dependências existentes nas instruções remanescentes no laços.

Um método denominado *privatização* faz com que todas as referências à variável original sejam substituídas para uma variável temporária local ao macronó, de onde a variável original é removida. Sua vantagem é a menor quantidade de memória necessária para sua manutenção, podendo ser armazenado no próprio processador no qual o macronó será executado, garantindo a independência do macronó.

6.7 Avaliações Dinâmicas

Algumas decisões no processo de compilação são impossíveis, pois dependem da determinação exata das dependências existentes, valores de variáveis que devem ser conhecidos, e isso não ocorre em muitos algoritmos seriais. Estas informações geralmente são conhecidas somente em tempo de execução.

Uma alternativa seria o compilador inserir alguns testes em relação a pontos dúbios no código paralelo e poder executar subtarefas alternativas e mais eficientes levando em conta alternativas geradas em tempo de execução.

Outro problema está na análise de ponteiros e no paralelismo de construções recursivas, e cuja possibilidade, no caso de ponteiros, seria inferir sobre a relação entre os ponteiros e os elementos apontados. Avaliações automáticas sobre ponteiros possibilitam uma análise de dependência mais precisa, porém são necessárias técnicas de identificação de padrões algorítmicos ao longo do código sequencial. Alguns compiladores permitem que lhe sejam passadas informações sobre a estrutura de programação desenvolvida, auxiliando no processo de produção de código automático e avaliação das dependências existentes.

Para solucionar os problemas da recursão existe uma técnica denominada *quebra de recursão*, na qual a função recursiva é expandida a partir de seu valor de entrada inicial e então reduzida pela aplicação de uma função adicional (passível de paralelização) no processo recursivo sobre os resultados parciais da expansão.

Comparação de Performance

Testes foram realizados em uma máquina Cedar, que consiste de uma máquina composta por 4 clusters de 8 processadores cada. Parte da memória do Cedar é compartilhada entre todos os processadores e parte é compartilhada apenas com os processadores do mesmo cluster. Esta configuração é interessante justamente por mesclar os tipos de memória, se tornando um ambiente híbrido que pode simular melhor um computador paralelo “genérico”.

Para os testes, foi usado um “Benchmark perfeito”, que é uma coleção de programas que tentam simular o máximo de aplicações possíveis, nas mais diversas tarefas e foram usadas as técnicas de paralelização automáticas mais usadas nos anos 80 (os testes em questão foram feitos em 1993).

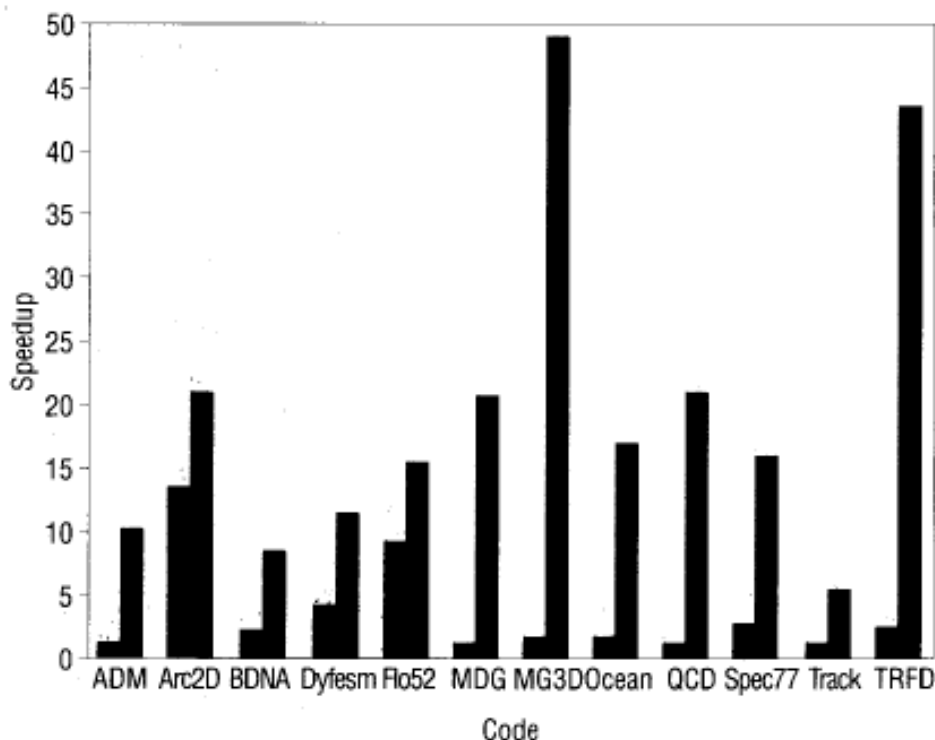


Figure 1. Speedups of automatically and manually parallelized versions of the Perfect Benchmarks on the Cedar machine.

O gráfico acima indica o speedup atingido com paralelização manual (barras da direita) e paralelização automática (barra da esquerda) para cada um dos algoritmos listados. É notória a discrepância entre os 2 métodos de paralelização, em apenas 2 dos algoritmos (Arc2D e Ro52) o speedup alcançado foi minimamente comparável. De uma forma geral, os testes mostraram que houve um ganho significativo no speedup apenas em programas pequenos com o uso da paralelização automática. Nas aplicações reais o ganho foi muito pequeno.

Estudo de Caso: Polaris

Com base nestes estudos, nasceu a idéia de se implementar as técnicas de paralelização manual dentro dos compiladores, de forma que estes compiladores abrangessem boa parte das técnicas manuais de paralelização. Assim nasceu a idéia do Polaris.

Inicialmente, com poucas exceções, apenas as transformações que um compilador paralelizador fazem foram implementadas, baseadas nas técnicas manuais de paralelização. Além destas transformações, apenas as baseadas no código fonte foram feitas, de forma que

as paralelizações que dependem do conhecimento da aplicação e as paralelizações específicas de cada algoritmo ficaram de fora.

Essas técnicas de transformação foram implementadas num compilador chamado Polaris, que numa versão preliminar conseguiu paralelizar metade dos programas mostrados na figura acima tão bem quanto a paralelização manual

, utilizando técnicas não muito rebuscadas. Em pouco tempo, implementando mais algumas técnicas, se conseguiu resultados semelhantes para mais 2 dos algoritmos da figura acima.

Conclusões

As técnicas de paralelização surgiram inicialmente para que se tirasse um maior proveito das arquiteturas paralelas em busca de instruções e operações que pudessem ser executadas em paralelo. Logo notou-se que os computadores poderiam ajudar no trabalho de busca e paralelização de código.

Durante o decorrer dos anos os algoritmos de paralelização automática evoluíram bastante, principalmente nos anos 90, de forma que alguns deles hoje em dia tem um speedup comparável aos de uma paralelização manual. Porém a paralelização manual conta com o cérebro humano, e técnicas que sejam aplicáveis a alguns algoritmos pela sua natureza não podem ser tão facilmente inferidos pelo computador atualmente.