ALESSANDRO DE AZEVEDO
ISSAM IBRAHIM

HIPERGRAFOS DIRECIONADOS E COMPUTAÇÃO PARALELA

Curitiba

2009

ALESSANDRO DE AZEVEDO
ISSAM IBRAHIM

HIPERGRAFOS DIRECIONADOS E COMPUTAÇÃO PARALELA

Trabalho apresentado ao curso de Bacharelado em Ciência da Computação da Universidade Federal do Paraná, como requisito parcial à obtenção do título de Bacharel em Ciência da Computação

Orientador:
André Luiz Pires Guedes

UNIVERSIDADE FEDERAL DO PARANÁ

Curitiba

2009

Lista de Figuras

Resumo

Este trabalho tem como objetivo estudar a modelagem de problemas relacionados ao fluxo de atividades através de hipergrafos direcionados, mostrando como representar um fluxo de tarefas usando o Hipergrafo Serial-Paralelo-Disjunção (HSPD) e, além disso, demonstrar um algoritmo de força bruta para calcular o maior paralelismo do fluxo. Foi um realizado um estudo a respeito de Paralelismo, em que foram abordados tópicos a respeito de Paralelismo em Nível de Instrução, Dependência de Dados, além de outros conceitos sobre este tema. Para um pleno entendimento do trabalho, um capítulo com definições sobre Grafos, Hipergrafos e outros conceitos fundamentais foi abordado com detalhes (capítulo 3). Em seguida, foi feito um estudo detalhado do funcionamento do algoritmo A que tem o objetivo de calcular o maior paralelismo de fluxo. Esta algoritmo foi definido por Cleverson Boff, Cleverton Krenski e Juliano Lorenzet na sua monografia de graduação (”Modelagem de Fluxo de Trabalho Utilizando Hipergrafos Direcionados”) orientada pelo professor André Guedes.

Palavras chave: hipergrafos, fluxo de atividades, paralelismo.

Abstract

This paper's goal is to study the modeling of problems related to workflows through directed hypergraphs, showing how to represent a flow of tasks using Series-Parallel-Disjunction Hypergraph (HSPD) and also demonstrate a brute force algorithm to compute the largest parallel flow. It was a study done about parallelism, which take up topics concerning Parallel Instruction-Level, Addiction to Data, and other concepts on the subject. For a full understanding of the work, a chapter with definitions of graphs, hypergraphs and other key concepts were discussed in detail (Chapter 3). Then, was made a detailed study of the functioning of the A algorithm that is designed to calculate the largest parallel flow. This algorithm was defined by Cleverson Boff, Cleverton Krenski and Juliano Lorenzet in his undergraduate thesis (”Modelagem de Fluxo de Trabalho Utilizando Hipergrafos Direcionados”) supervised by Professor André Guedes.

Keywords: hypergraphs, workflow, parallelism.

Sumário

1 Introdução

Existem muitas atividades que são compostas por diversas tarefas, então podemos considerar uma atividade como um fluxo de tarefas. Colocando-as a serem executadas paralelamente, vamos obter um melhor desempenho, minimizando custos e otimizando tempo e uso dos recursos. Porém, entre essas tarefas pode existir dependência de dados, ou seja, uma tarefa necessita esperar o término da execução de uma outra tarefa para poder continuar o seu processamento.

Para modelar o fluxo, utilizamos o conceito de hipergrafos direcionados, em que cada tarefa é representada por um vértice e as dependências por hiper-arcos.

Consideremos o exemplo a seguir:

```
Load r1, r2
Add r3, r3, 1
Add r4, r4, r2
```

Podemos pensar em cada uma das instruções do código a cima como sendo um vértice. Estes 3 vértices poderão ser executados em paralelo pois não há dependência de dados entre eles. Já no exemplo a seguir, existe uma dependência, pois a segunda instrução pode ser buscada e decodificada antecipadamente, mas não pode ser executada até que seja completada a execução da primeira instrução.

```
add r1, r1, r2
move r3, r3, r1
```

Todos esses casos podem ser tratados dentro de um Hipergrafo direcionado para a obtenção do cálculo do maior grau de paralelismo, que de fato é de extrema importância para que os

recursos de uma máquina sejam usados da melhor forma possível, auxiliando na melhoria de desempenho do processo não alocando mais recursos do que o fluxo necessita.

O objetivo desse trabalho é demonstrar que o algoritmo de força bruta, implementado por Cleverson Afonso Boff, Cleverton Aparecido Krenski e Juliano Lorenzet, que tem o objetivo de encontrar o maior grau de paralelismo em um fluxo de atividades, de fato atinge o seu objetivo considerando todos os possíveis caminhos dentro desse fluxo de atividades.

2 *Paralelismo*

Em qualquer atividade que um ser humano realize, a ideia de fazê-la em grupo, implicitamente, indica um aumento de eficiência e um menor tempo para concluí-la. A construção de um prédio, a preparação de uma refeição, o desenvolvimento de um sistema são processos que, quando realizados em grupo, podem ser otimizados. O conceito matemático de duas retas que seguem a mesma direção, mas que só se encontram no infinito, o de paralelo, é empregado de forma geral para representar atividades executadas de forma cooperativa.

O paralelismo, de forma geral, está associado com aumento de desempenho. Da mesma maneira que o ser humano procura otimizar suas tarefas por meio da cooperação, os computadores, também, podem ser construídos de forma paralela, podendo ser aplicados em vários contextos da computação, como na programação concorrente, em sistemas distribuídos e multiprocessados, além de projetos de hardware. A obtenção de desempenho não depende somente em utilizar dispositivos de hardware mais rápidos, mas também em melhorias na arquitetura dos computadores e técnicas de processamento.

Devemos lembrar que o paralelismo é uma propriedade dos programas, ou seja, cada programa apresenta uma quantidade própria de paralelismo, ou grau de paralelismo. Detectar e explorar esse paralelismo são os meios de aumentar o desempenho dos computadores.

Nosso foco neste trabalho é o paralelismo em nível de instrução. A seguir, veremos uma breve introdução a este tipo de paralelismo.

2.1 Paralelismo em Nível de Instrução

Paralelismo em nível de instrução é o potencial de sobreposição na execução de várias instruções simultaneamente. Se duas instruções são paralelas, então elas podem ser executadas simultaneamente em um pipeline sem nenhuma parada, assumindo que o pipeline possui recursos suficientes (sem riscos estruturais).

Esse tipo de paralelismo pode ser executado tanto pelo compilador, o qual constitui uma abordagem estática, ou pelo processador, numa abordagem dinâmica.

A maneira mais fácil de aumentar o paralelismo de nível de instrução é por meio da exploração de paralelismo existente entre as iterações de um laço. Essa técnica pode ser aplicada tanto pelo compilador quanto pelo processador. Enquanto o compilador gera mais código para explorar esse tipo de paralelismo, o processador tenta adivinhar se a iteração irá ou não se repetir.

2.2 Dependência de Dados Verdadeira

Consideremos a seguinte sequência de instruções:

```
add r1, r1, r2
move r3, r3, r1
```

A segunda instrução pode ser buscada e decodificada antecipadamente, mas não pode ser executada até que seja completada a execução da primeira instrução. A razão é que a segunda instrução requer dados produzidos pela primeira instrução. Essa situação é conhecida como depedência de dados verdadeira (também chamada dependência de fluxo ou dependência de escrita-leitura).

Caso não exista dependência de dados, duas instruções podem ser buscadas e executadas em paralelo. Se existir uma dependência de dados entre a primeira e a segunda instruções, então a execução da segunda instrução deve ser atrasada. De um modo geral, uma instrução deve ser atrasada até que todos os seus dados de entrada tenham sido produzidos.

O paralelismo no nível de instruções existe quando as instruções de uma sequência são independentes e podem, portanto, ser executadas em paralelo, por sobreposição.

Consideremos os seguintes exemplos:

Exemplo de código assembly 1:

```
Load r1, r2
Add r3, r3, 1
Add r4, r4, r2
```

Exemplo de código assembly 2:

```
Add r3, r3, 1
Add r4, r3, r2
Store [r4], r0
```

As três instruções do exemplo 1 são independentes e, em tese, podem ser executadas paralelamente. As três instruções do exemplo 2, diferentemente do exemplo 1, não podem ser executadas em paralelo, porque a segunda instrução usa o resultado da primeira, e a terceira instrução usa o resultado da segunda, logo existe uma dependência de dados.

2.3 Limites no Paralelismo

A ideia de que o aumento de recursos nos computadores, como o número de unidades funcionais e de acesso a memória ou mesmo o número de processadores, aumenta o paralelismo nos leva à noção de um limite para isso, pois de forma contrária o paralelismo seria ilimitado. Logo, o paralelismo é um propriedade dos programas, sendo limitado por eles. Isto significa que não é possível obter maior paralelismo do que o próprio programa fornece. Além disso, existem limitações de paralelismo na arquitetura da máquina onde o programa é executado e na linguagem de programação utilizada.

2.4 Fluxo de Dados

O fluxo de dados de um programa é definido pela dependência entre a computação dos dados de um programa. Um programa sem dependências de dados poderia ser executado em um único ciclo em uma máquina com um número de unidades funcionais maior ou igual ao número de instruções desse programa.

Assim, o fluxo de dados acaba sendo o único limitador real de paralelismo de um programa quando abstrai-se a linguagem de programação utilizada e a arquitetura alvo no qual o programa será executado. Devido ao fato de um programa ser um conjunto de instruções que descreve a solução de um problema, o paralelismo consiste simplesmente em executar o máximo de instruções simultaneamente sem afetar a corretude de um programa. O fluxo de dados de um programa é o limite superior de paralelismo que pode ser encontrado neste programa. Entretanto, é necessário que o programa seja escrito numa linguagem de programação e executado numa máquina, incluindo, dessa forma, mais restrições de paralelismo, devido as imposições criadas pela arquitetura.

Para efeito de ilustração, consideremos o fluxo de dados da figura 2.2 que considera a avaliação de um exercício feito por um aluno. Neste exemplo, é simples perceber a dependência de dados entre as tarefas a serem executadas, pois antes de se avaliar o exercício, é necessário que ele seja concluído pelo aluno.

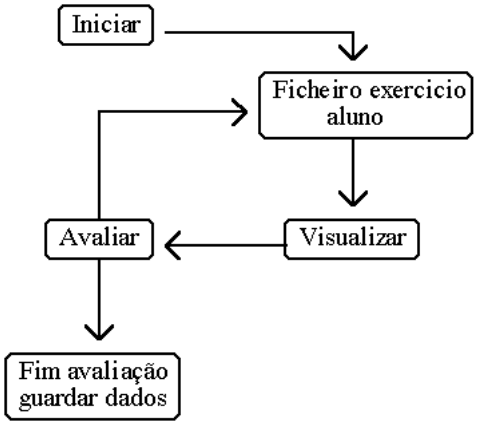

Figura 2.1: Fluxo de dados simples

2.5 Fluxo de Controle

Em ciência da computação, o fluxo de controle é o que determina a sequência de execução das instruções de um programa. Nem todas as instruções de um programa são executadas sequencialmente. Um tipo de instrução especial, chamada de instrução de desvio, altera o fluxo de execução.

Os tipos de estruturas de controle disponíveis diferem de linguagem para linguagem, mas podem ser cruamente caracterizados por seus efeitos. O primeiro é a continuação da execução em uma outra instrução, como na estrutura sequencial ou em uma instrução de desvio. O segundo é a execução de um bloco de código somente se uma condição é verdadeira, uma estrutura de seleção. O terceiro é a execução de um bloco de código enquanto uma condição é verdadeira, ou de forma a iterar uma coleção de dados, uma estrutura de repetição. O quarto é a execução de instruções distantes entre si, em que o controle de fluxo possivelmente volte para a posição original posteriormente, como chamadas de subrotinas. O quinto é a parada do programa de computador.

Interrupções e sinais são mecanismos de baixo nível que podem alterar o fluxo de controle de forma similar a uma subrotina, mas geralmente em resposta a algum estímulo externo ou um evento ao invés de uma estrutura de controle em uma linguagem.

O fluxo de controle é um grande limitador dentro daquilo que chamamos de paralelismo. Ele é quem determina a sequência de execução das instruções de um programa.

Da mesma maneira que o fluxo de dados está relacionado com as dependências de dados, o fluxo de controle relaciona-se com as dependências de controle. Uma instrução X é dependente por controle de uma instrução Y, quando a execução de X está condicionada a algum resultado da execução de Y.

Para efeito de ilustração, a seguir temos um exemplo que demonstra a sequência de instruções de um código simples:

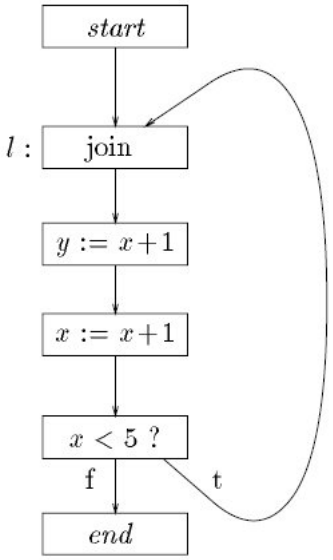

Figura 2.2: Exemplo de fluxo de controle

2.6 Pipeline

Pipelines são conjuntos de elementos que processam dados, os quais são conectados em série, sendo que a saída de um elemento é a entrada de outro. Pipelines são normalmente utilizados para aumentar o throughput de dados, ou seja, aumentar a vazão de dados em, por exemplo, um processador, não aumentando o tempo de execução de um dado vizinho.

3 *Fundamentação Teórica*

Nesta seção serão abordados conceitos fundamentais para o entendimento do trabalho como um todo.

3.1 Grafo e grafo direcionado

Em matemática e ciência da computação, grafo é o objeto básico de estudo da teoria dos grafos. Tipicamente, um grafo é representado como um conjunto de pontos (vértices) ligados por retas (as arestas). Um grafo, expresso frequentemente por G = (V,A), é composto por um conjunto não vazio e finito V(conjunto de vértices) e por um conjunto A (conjunto de arestas) composto por pares não ordenados de elementos distintos de V, ou seja, cada elemento de A é um subconjunto de dois elementos de V.

Dependendo da aplicação, as arestas podem ser direcionadas, e são representadas por ”setas”, sendo chamada de grafo direcionado. Um grafo direcionado D = (V, A) é, de forma análoga, composto por um conjunto não vazio e finito V de vértices e por um conjunto A finito de arcos, em que ”a”$\in$ A é um par ordenado de elementos de V, ou seja, ”a”$\in$ VxV.

Como podemos observar no grafo a seguir, temos que:
G = (V,A), sendo V = {1, 2, 3, 4, 5, 6, 7, 8} e A = { {1,2},{1,5},{2,5},{3,4},{6,7} }.

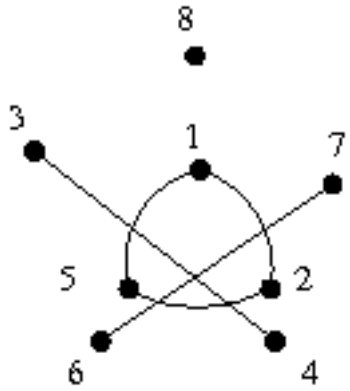

Figura 3.1: Exemplo de grafo comum

Um exemplo de grafo direcionado pode ser percebido no exemplo a seguir, onde nota-se claramente a direção das arestas, representada pelas setas em cada umas das arestas do grafo:

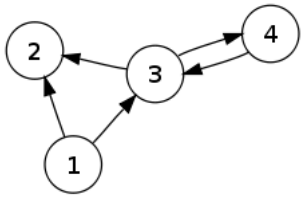

Figura 3.2: Exemplo de grafo direcionado

3.2 Hipergrafo e hipergrafo direcionado

O hipergrafo é uma generalização do conceito de grafos. Em um hipergrafo a cardinalidade das arestas pode ser diferente de dois. Um hipergrafo H = (V,A) onde V é um conjunto finito de vértices e A é um conjunto finito de hiperarestas, em que uma hiperaresta a $\in$ A é um subconjunto não vazio de V.

Um exemplo de hipergrafo pode ser visto claramente na figura 3.3 onde temos V = {v1, v2, v3, v4, v5, v6, v7}, E = {e1, e2, e3, e4} = {{v1, v2, v3}, {v2, v3}, {v3, v5, v6}, {v4}}.

Avançando e especializando ainda mais o objeto de estudo, podemos definir hipergrafos como sendo um par (H = (V,A)) em que o conjunto de vértices (V) é finito e conjunto de hiperarestas é formado por pares ordenados (X,Y), sendo que X e Y são subconjuntos disjuntos de elementos de V.

Aqui definimos que o conjunto de arestas de saída de S de um vértice v é:

$S(v) = \{(X,Y) \in A \mid v \in X\}$

E como vértices de entrada:

$E(v) = \{(X,Y) \in A \mid v \in Y\}$

Percebemos, observando atentamente os hipergrafos das figuras 3.3 e 3.4, a diferença entre um hipergrafo e um hipergrafo direcionado: o conjunto de vértices é semelhante, porém ao utilizarmos hiperarestas direcionadas, definimos o caminho que se pode ser tomado dentro do hipergrafo.

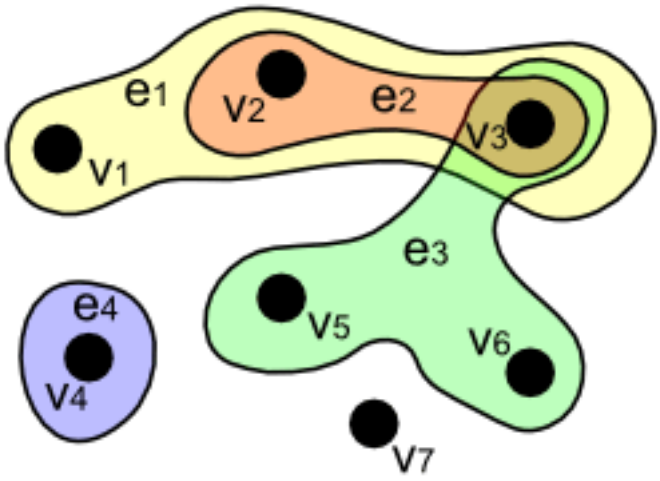

Figura 3.3: Exemplo de um hipergrafo

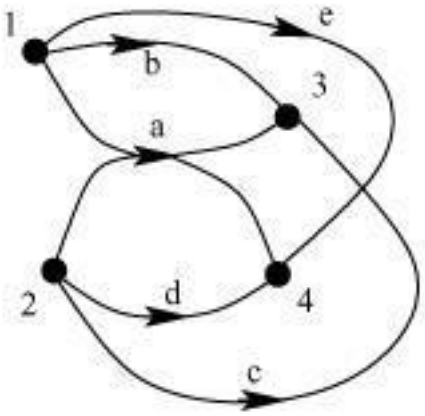

Figura 3.4: Exemplo de um hipergrafo direcionado

3.3 Hiper-Caminho

Dado um Hipergrafo direcionado H = (V,E), e dois vértices s e t, um hiper-caminho de s a t, de tamanho k, é uma sequência de hiper-arcos C = $(e_{i1}, e_{i2}, ..., e_{ik})$ onde s pertence a Org(e_{i1}) e t pertence a Dest(e_{ik}), e para cada hiper-arco e_{ip} de C com $1 \leq p \leq k$ temos que:

- Org(e_{ip}) $\cap$ (Dest($\{e_{i1}, e_{i2}, ..., e_{ip-1}\}$) $\cup$ $\{s\}$) <> $\{\}$;
- Dest(e_{ip}) $\cap$ (Org($\{e_{ip+1}, e_{ip+2}, ..., e_{ik}\}$) $\cup$ $\{t\}$) <> $\{\}$.

Onde Org(e_{i1}) e Dest(e_{ik}) são respectivamente Org(C) e Dest(C).

Na figura a seguir, vemos um exemplo de hiper-caminho. O hiper-caminho C = $\{a,c,d\}$, de 1 a 7, tem origem $\{1,2\}$ e destino $\{7\}$.

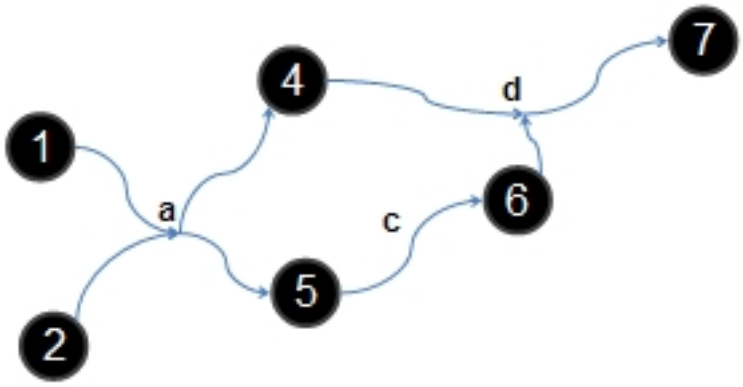

Figura 3.5: Hiper-caminho

3.4 Conexão

Dizemos que um vértice u está hiper-conectado ou simplesmente conectado ao vértice x se existe um hiper-caminho de x a y.

3.5 Propriedades sobre os conjuntos de hiper-arcos

3.5.1 Propriedade S

Seja C um conjunto de hiper-arcos e H_c o hipergrafo direcionado induzido por C. O conjunto C terá a propriedade S se, para todo vértice x de H_c, $|E(X)| \leq 1$ e $|S(X)| \leq 1$, ou seja, cada vértice aparece no máximo em um conjunto origem e um destino de C.

Um hiper-caminho com esta propriedade é um S-caminho.

3.5.2 Propriedade B

Um hiper-caminho de C, de s a t, de tamanho K, tem a propriedade B se, para cada hiper-arco e_{ip} de C, com $1 \leq p \leq$ k, temos que: Org(e_{ip}) C = (Dest($e_{i1}, e_{i2}, ..., e_{ip-1}$) $\cup \{s\}$).

Um hiper-caminho com esta propriedade é um B-caminho.

3.5.3 Propriedade F

Um hiper-caminho de C, de s a t, de tamanho k, tem a propriedade F se, para cada hiper-arco e_{ip} de C, com $1 \leq p \leq$ k, temos que: Dest(e_{ip}) C = (Org($\{e_{ip+1}, e_{ip+2}, ..., e_{ik}\}$) $\cup \{t\}$).

Um hiper-caminho com esta propriedade é um F-caminho.

3.5.4 Ciclos

Um ciclo é um hiper-caminho C onde Org(C) $\cap$ Dest(C) <> $\{\}$.

3.6 Aplicação de Hipergrafos

Com esse conceito, podemos imaginar algumas situações em que hipegrafos direcionados se aplicam. Já faz algum tempo que cientistas da computação do mundo inteiro pesquisam sobre a programação paralela, e o conceito de hipergrafos aplica-se perfeitamente para as representações de programas que apresentem paralelismo. Imagine que podemos representar

por um vértice 'u' de um hipergrafo direcionado um processo que necessite de outros 'n' processos para atingir o resultado esperado na saída do programa. Podemos propor que para um hipergrafo direcionado H = (V,A) representando os processos de um programa, para cada 'v' $\in$ V, temos que para cada aresta em E(v) temos um ou mais processos dos quais 'v' depende e para cada aresta em S(v) temos um ou mais processos que dependem de 'v'.

Trazendo esse conceito para situações administrativas e de planejamento, podemos imaginar que cada vértice pode ser uma tarefa a ser executada e as hiperarestas seriam dependências destas.

Com as informações de fluxo contidas nas hiperarestas, podemos calcular o número de processadores para a execução em menor tempo e otimizar a utilização dos recursos.

Há, ainda, a situação de instruções sendo jogadas no pipeline de um processador. O hipergrafo seria a modelagem da situação, sendo que cada vértice é uma instrução, e as hiperarestas representam as dependências de dados. A otimização consiste em manter um dado dependente do outro o mais longo possível na execução; assim, esta distância diminuiria o número de bolhas pelo processador.

Em resumo, os hipegrafos direcionados 'se apresentam como uma alternativa para a modelagem de dados em que relações binárias usuais não são adequadas' (Guedes, Hipergrafos Direcionados 2001).

3.7 HSPD - Hipergrafos Serial-Paralelo-Disjunção

Da mesma maneira como os fluxos de tarefas sequenciais podem ser modelados utilizando-se de grafos direcionados, seria interessante que os problemas que contenham paralelismos pudessem ser modelados de maneira computacional. Para isso utilizamos os Hipergrafos direcionados. Nesta estrutura, podemos modelar as condições E e as condições OU. Para apresentarmos os problemas que possuem paralelismos de uma maneira mais estruturada, podemos utilizar a classe dos Hipergrafos Serial-Paralelo-Disjunção (HSPD), que definiremos na sequência. Antes disso, é necessário que o conceito de Hipergrafo de fluxo esteja bem sólido.

Um Hipergrafo de fluxo é um tripla H = (V,E,s), em que (V,E) é um hipergrafo direcionado, S pertence a V, sendo V o vértice de origem, e existe um B-caminho de s para qualquer outro vértice em V. [GUEDES, 2001]

Utilizando-se deste conceito podemos definir um hipergrafo de controle.

Um hipergrafo de controle Hp = (V,E,s,t) associado a um programa paralelo P é um hipergrafo de fluxo (v,E,s) e um vértice t pertencente a V, em que:

1. cada instrução (ou conjunto de instruções elementar) de P é um vértice de V;
2. cada dependência de execução de P é um hiper-arco de E;
3. s é o vértice início e t o vértice final. [GUEDES, 2001]

Abaixo segue um de hipergrafo direcionado que modela o seguinte fluxo de controle:

```
int a = 0,   b = 0;        (P1)
if (a < 10){               (P2)
        a++;               (P3)
        b–;                (P4)
}
else{                      (P5)
        a–;                (P6)
        b++;               (P7)
}
```

Figura 3.6: Código

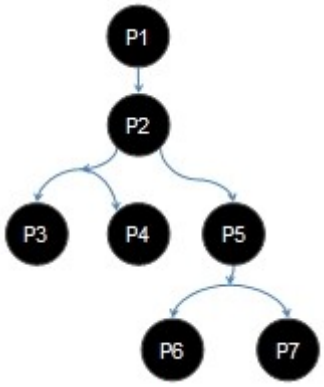

Figura 3.7: Hipergrafo direcionado que modela o código acima

Com base na figura 3.7, podemos descrever como ocorre a execução do fluxo sobre o grafo apresentado. A execução inicia-se pelo passo 1 (P1). Quando a execução chega em P2, uma decisão deverá ser tomada. Executam-se, então, os passos 3 ou 4 OU os passos 6 e 7. Caso a condição de P2 for satisfeita, então executam-se os passos 3 e 4, caso contrário os passos 6 e 7 serão então executados. Pode-se concluir que em P2 gere-se um OU-Exclusivo, pois se P3 e P4 forem executados, então P6 e P7 não serão, e vice-versa.

Após essas definições, podemos então definir os hipergrafos Serial-Paralelo-Disjunção. Esta família de hipergrafos se faz bastante útil para representarmos fluxos de controle de programas escritos em linguagens que suportam estruturas de controle como if-then-else, case, while, etc.

Um hipergrafo Serial-Paralelo-Disjunção (HSPD) H = (V,E,s,t) é um hipergrafo direcionado e dois vértices, s e t, que satisfazem uma das condições abaixo:

- Trivial: V = s. E = 0 e t = s.
- Serial: V = V1 $\bigcup$ V2 , E = E1 $\bigcup$ E2 $\bigcup$ $\{(\{t1\},\{s2\})\}$, s = s1 e t = t2 , onde $H_1 = (V_1, E_1, s_1, t_1)$ e $H_2 = (V_2, E_2, s_2, t_2)$ são HSPDs (não necessariamente distintos).
- Paralelo: V = $\bigcup_{i=1}^{k} V_i \bigcup \{s,t\}$, E = $\bigcup_{i=1}^{k} E_i \bigcup \{(\{s\},\{s_1,...,s_k\}),(\{t_1,...,t_k\},\{t\})\}$, onde Hi = (V, E_i, s_i, t_i), com i = 1...k, são HSPDs distintos, e s e t são vértices novos.

- Disjunção: V = $\bigcup_{i=1}^{k} V_i \bigcup \{s,t\}$, E = $\bigcup_{i=1}^{k} (E_i \bigcup \{(\{s\},\{S_i\}),(\{t_i\},\{t\})\})$, onde Hi = (V, E_i, s_i, t_i), com i = 1...k, são HSPDs distintos, e s e t são vértices novos.

Em todas estas condições, quando os HSPDs H_i forem distintos, também devem ser disjuntos. [GUEDES, 2001]

3.8 Modelagem de Fluxo de Tarefas com Hipergrafos Direcionados

Num hipergrafo direcionado, temos que os vértices representam as atividades a serem executadas e os hiper-arcos as dependências entre essas atividades, pois tarefas podem ter que ser executadas antes de outras devido a dependência de dados.

No exemplo a seguir, ilustramos um fluxo de controle. Cada vértice (vértices nomeados de A a H) indica uma tarefa a ser realizada. Os hiper-arcos que saem indicam as próximas tarefas que necessitam ser realizadas após a conclusão da tarefa atual. Os que chegam indicam dependência para o início da realização da nova tarefa gerada. Veja o fluxo a seguir:

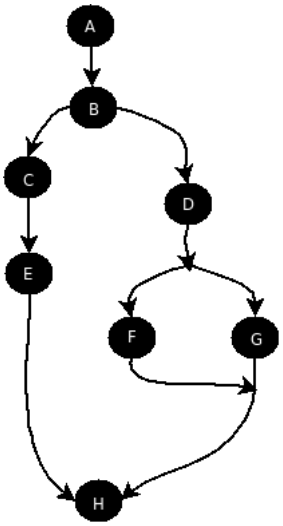

Figura 3.8: Exemplo de fluxo de tarefas

3.9 Informações pré-existentes na estrutura de dados

Com os hipergrafos direcionados, é possível modelar e enxergar no nível de estrutura de dados, uma série de informações relevantes sem a necessidade de anexos. Assim é possível visualizar e modelar com facilidade as dependências entre as tarefas. Podemos descrever e obter vários tipos de condições e dependências por meio da estrutura básica do hipergrafo-direcionado, sendo possível verificar as seguintes condições:

3.9.1 OU-F-Condição

Sejam u, v e w vértices distintos de um hipergrafo, e E1, E2,...,En, conjuntos distintos de hiper-arcos que partem de u, formando hiper-caminhos entre u e v e entre u e w. Assim, podemos definir uma OU-F-Condição como sendo duas ou mais sequências de hiper-arcos distintos que ligam dois extremos de um hiper-caminho. Porém, somente um dos hiper-arcos deve ser percorrido para que uma determinada condição seja satisfeita. Veja a ilustração a seguir:

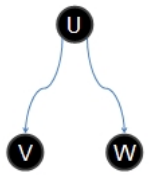

Figura 3.9: Um exemplo de OU-F-Condição

3.9.2 OU-B-Condição

Sejam u, v e w vértices distintos de um hipergrafo, e E1, E2,...,En, conjuntos distintos de hiper-arcos que chegam em u, formando hiper-caminhos entre v e u e entre w e u. Assim, podemos definir uma OU-B-Condição como sendo duas ou mais sequências de hiper-arcos distintos que ligam os dois extremos de um hiper-caminho. Porém, somente um dos hiper-caminhos possíveis precisa ser tomado para que uma determinada condição seja satisfeita. Veja a ilustração a seguir:

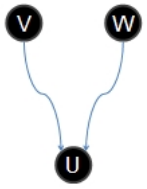

Figura 3.10: Um exemplo de OU-B-Condição

3.9.3 E-F-Condição

Sejam u, v e w vértices distintos de um hipergrafo, e E1, E2,...,En, conjuntos distintos de hiper-arcos que partem de u, formando hiper-caminhos entre u e v e entre u e w. Assim, podemos definir uma E-F-Condição como sendo duas ou mais sequências de hiper-arcos distintas que ligam dois extremos de um hiper-caminho, sendo que todos os hiper-caminhos possíveis precisam ser percorridos para que uma determinada condição seja satisfeita. Veja o exemplo a seguir:

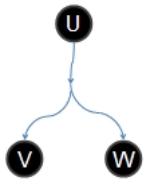

Figura 3.11: Um exemplo de E-F-Condição

3.9.4 E-B-Condição

Sejam u, v e w vértices distintos de um hipergrafo, e E1, E2,...,En conjuntos distintos de hiper-arcos que chegam em u, formando hiper-caminhos entre v e u e entre w e u. Assim, podemos definir uma E-B-Condição como sendo duas ou mais sequências de hiper-arcos distintas que ligam os dois extremos de um hiper-caminho, sendo que todos os hiper-caminhos possíveis precisam ser percorridos para que uma determinada condição seja satisfeita. Veja o exemplo a seguir:

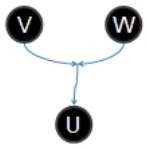

Figura 3.12: Um exemplo de E-B-Condição

3.9.5 Detalhamento

Voltando ao exemplo da figura 3.8, temos como o vértice inicial o vértice A e o vértice final o H, pois, como o vértice A não tem pai, ou seja, não depende de nenhuma outra tarefa para ser executado, ele acaba por ser o primeiro vértice a ser executado e o vértice H acaba por ser o vértice final, já que não há nenhum vértice a ser executado após ele. Os passos de execução na figura 3.8, seguem a seguinte ideia:

1. A é executado passando direto para B (B depende de A para poder executar);
2. a partir do vértice B, saem dois hiper-arcos que conectam B com C e D representando um OU-F-Condição; logo, apenas um dos hiper-arcos deve ser executado;
3. Caso o hiper-arco da esquerda seja executado, os vértices C, E serão executados na sequência um após o outro e por último o vértice H será executado, finalizando o fluxo
4. No caso do hiper-arco da direita ser executado, temos um E-F-Condição representado pelos hiper-arcos que conectam D a F e G. Isso significa que F e G dependem de D e ambos F e G serão executados. Além disso, F e G criam uma conexão E-B-Condição com H, o que significa que H depende de ambos F e G para executar. Uma vez que F e G são executados H poderá executar e assim o fluxo estará terminado.

3.10 Algoritmos de Força Bruta

Para a continuação de nossos estudos é necessário entendermos o conceito de algoritmo de Força Bruta. Em ciência da computação, força bruta (ou busca exaustiva) é uma algoritmo

trivial mas de uso muito geral que consiste em enumerar todos os possíveis candidatos de uma solução e verificar se cada um satisfaz o problema.

Por exemplo, um algoritmo para encontrar os divisores de um número natural n é enumerar todos os inteiros de 1 a n, e verificar para cada um se ele dividido por n resulta em resto 0.

Esse algoritmo possui uma implementação muito simples, e sempre encontrará uma solução se ela existir. Entretanto, seu custo computacional é proporcional ao número de candidatos a solução, que, em problemas reais, tende a crescer exponencialmente. Portanto, a força bruta é tipicamente usada em problemas cujo tamanho é limitado, ou quando há uma heurística usada para reduzir o conjunto de candidatos para uma espaço aceitável. Também pode ser usado quando a simplicidade da implementação é mais importante que a velocidade de execução, como nos casos de aplicações críticas em que os erros de algoritmo possuem em sérias consequências.

3.11 Definições Adotadas

Nesta seção trataremos de algumas definições que são necessárias para o completo entendimento do trabalho. Aqui serão considerados apenas fluxos inicializáveis e compatíveis.

Chamaremos S(v), definida na seção 3.2, de BS[v]. Temos que BS[v] contém os hiper-arcos de chegada do vértice v. Da mesma Forma, chamaremos E(v), também definida na seção 3.2, de FS[v]. Temos que FS[v] contém os hiper-arcos que saiem do vértice v.

Seja a seguinte proposição:
Proposição: Seja o vértice v, se BS(v) = $\{\}$, então v é o vértice inicial do fluxo. Dizemos que o fluxo F é inicializável, se, para todo vértice v pertencente a F, existe somente um vértice v, em que BS(v) = $\{\}$. Se existir mais de um vértice que satisfaça esta condição (BS(v) = $\{\}$), então o fluxo é denotado multi-inicializável. Se não existir nenhum vértice tal que BS(v) = $\{\}$, o fluxo é denominado não-inicializável.

Um fluxo no qual apareçam ramificações em OU a qualquer tempo, e abaixo, na sua execução, apareçam convergências do tipo E, tendo como ancestral uma OU, é qualificado

como incompatível, pois suas condições nunca poderão ser satisfeitas. Seja o exemplo a seguir:

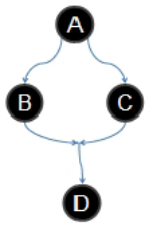

Figura 3.13: Um exemplo de um hipergrafo incompatível

Um fluxo pode ser qualificado da seguinte forma:

Quanto à inicialização:

- mono-inicializável (para simplificar, chamaremos apenas de inicializável);
- multi-inicializável;
- não-inicializável.

Quanto ao seu percurso:

- compatível;
- incompatível.

Dizemos que um fluxo não-inicializável ou incompatível é intratável.

Outras definições importantes para o entendimento do trabalho são a questão da dependêndia de vértices e a utilização da mesma:

- Dependência: Definimos como dependência a relação entre os vértices de destino e os vértices de origem de um hiper-arco, ou seja, para a execução dos vértices do conjunto de vértices de destino de um hiper-arco é necessário que todos os vértices do conjunto de origem estejam finalizados.
- Utilização de vértices dependentes: Nos hipergrafos utilizados neste documento, existem vários exemplos em que ocorrem dependências. Nestes casos, as tarefas dependentes só

podem ser executadas depois que a dependência for resolvida. Além disso, quando uma determinada tarefa é executada, então ela passa a ser considerada finalizada durante a execução de todo o fluxo, ou seja, tarefas que dependam desta execução não precisam fazê-la novamente. Assim, tarefas que já foram finalizadas podem ser usadas quantas vezes forem necessárias durante uma execução. Há casos em que poderia ser interessante que o vértice dependente pudesse ser usado apenas umas vez durante a execução. Porém, este caso não será abordado aqui.

4 Estudo de um Algoritmo de Força Bruta sobre Hipergrafos

Nesta seção iremos definir o Algoritmo A referente ao trabalho de Modelagem de Fluxos de Trabalho Utilizando Hipergrafos Direcionados de autoria de Cleverson A Boff, Cleverton A Krenski e Juliano Lorenzet. O objetivo do algoritmo é cálculo do maior paralelismo em um hipergrafo direcionado.

4.1 Estrutura de Dados Utilizada

Foram definidas três classes: classe Hipergrafo, classe Vértice e a classe Aresta. Para a classe Vértice foram utilizados dois vetores, um para guardar as arestas de origem e outro para guardar as arestas de destino. Para a classe Aresta, também foram utilizados dois vetores, como foi feito para a classe Vértice, porém para guardar os vértices anteriores e de destino. Se a aresta tem mais de um vértice de origem ou destino, então essa aresta é um OU; em caso contrário é um E. A classe Hipergrafo é composta por uma lista de vértices e outra de arestas.

```
 1 class TVertice:
 2         dado = None
 3         visita = None
 4         rotulo = “ ”
 5         tempo = 1
 6         fs = []
 7         bs = []
 8         b = None
 9         f = None
10         def init (self):
11                 self.fs = []
12                 self.bs = []
13
14 class THiperArco:
15         rotulo = “ ”
16         origens = []
17         destinos = []
18         p = None
19         def init (self):
20                 self.origens = []
21                 self.destinos = []
22
23 class THipergrafo:
24         lista vertices = []
25         lista arestas = []
26         def init (self):
27                 self.lista vertices = []
28                 self.lista hiperarcos = []
```

Figura 4.1: Classes usadas no algoritmo

No algoritmo também foi usado um Vetor de Listas. Um vetor de listas é uma lista de listas, o VETOR é a lista principal, onde cada nodo possui outra lista, que por sua vez cada nodo possui um vértice. A ideia é que cada lista, neste vetor, represente um possível caminho na execução do hipergrafo.

A ideia geral é que, a partir do vértice inicial (I), escolhe-se uma hiperaresta pertencente ao conjunto de hiperarestas de saída. Em seguida, se não houver nenhuma dependência, o vértice em que a hiperaresta corrente incide é executado, e posteriormente, colocado em uma lista auxiliar (L). Em seguida, escolhe-se um vértice pertencente a (L) e repete-se o procedimento de

execução. Este processo se repete até que o vértice final do fluxo seja alcançado e executado. Caso exista dependência, o vértice não pode ser executado até que todas as dependências tenham sido resolvidas.

No algoritmo são utilizadas quatro listas:

- Lista de Execução: Armazena os vértices que estão prontos para serem executados;
- Lista de Recém-finalizados: Armazena os vértices que foram finalizados na última iteração do algoritmo;
- Lista de Finalizados: Armazena os vértices que já foram concluídos;
- Lista de Pendentes: Armazena os vértices que ainda não podem ser executados sem razão de alguma dependência.

Ao longo da execução do algoritmo as listas vão sendo atualizadas de modo que as dependências sejam respeitadas e todo o fluxo seja percorrido. No primeiro passo o vértice inicial (I) é executado e os vértices pertencentes à FS(I) são colocados na lista de tarefas pendentes. Em seguida, executa-se um laço até que a lista de tarefas que podem ser executadas esteja vazia. Esta execução consiste em percorrer a lista de tarefas pendentes e para cada vértice (v) escolhesse uma hiperaresta e pertencente à FS(v). Se não houver dependência a lista de próximas tarefas que podem ser executadas é incrementada com os vértices nos quais a aresta e incide. Após percorrer a lista de pendências a lista de tarefas que podem ser executadas é incrementada com a lista de próximas tarefas que foi construída durante a última iteração. O algoritmo termina quando a lista de tarefas que podem ser executadas estiver vazia.

4.2 Algumas das Funções Utilizadas

- RealizaTrabalho: recebe uma lista de tarefas (vértices), e diminui o tempo em 1 a cada iteração. Quando o tempo for igual a zero significa que a tarefa foi executada. A tarefa é, então, removida da Lista de Tarefas em Execução e incluída na lista de tarefas finalizadas.

- VerificaDependencias: verifica se todos os vértices de origem da aresta pela qual o vértice foi alcançado foram finalizadas, verificando se estas tarefas pertencem a lista de tarefas finalizadas. Se foram finalizadas, inclui na lista de tarefas a gerar e retorna esta lista.

4.3 Funcionamento do algoritmo

A ideia por trás do algoritmo é que dado um determinado vértice 'a' do hipergrafo direcionado, analisamos se as hiperarestas que saem deste vértice formam um E ou um OU.

Se a aresta for OU, então criamos uma cópia da LISTA em que estamos trabalhando para cada hiperaresta, e em cada uma delas substituímos as origens por seus respectivos destinos. Em seguida removemos a LISTA original. Assim teremos uma cópia da LISTA para a hiperaresta OU. Seja o seguinte exemplo:

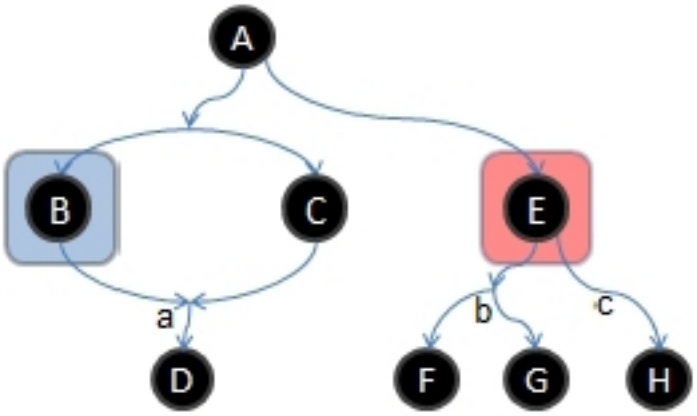

Figura 4.2: Exemplo de hipergrafo para ilustração

Podemos observar que de B parte apenas uma hiperaresta, chamada a. Portanto trata-se de uma E, que por definição exige B e C para seguir para D. Neste caso, devemos analisar se todas as origens da hiperaresta a estão marcadas como PRONTO. Para tanto, é marcado B como pronto e verificamos se estão todos prontos. Se sim, remove todos os vértices origem de a e colocamos no lugar de B todos os vértices destinos de a. Senão, não podemos seguir até estarem todos prontos. Devemos partir então para o próximo vértice da LISTA. Considere agora o vértice E. Podemos observar que de E partem mais de uma hiperaresta, no caso as hiperarestas

b e c. Portanto trata-se de uma OU. Nesta situação, para cada hiperaresta verificamos se suas origens estão prontas. Se todas as origens de todas as hiperarestas estiverem prontas, então criamos uma cópia da LISTA em que estamos trabalhando para cada hiperaresta, e em cada uma delas substituímos as origens por seus respectivos destinos. Em seguida removemos a LISTA original. Assim teremos uma cópia da LISTA para a hiperaresta b, substituindo E por F e G; e outra cópia para a hiperaresta c, substituindo E por H. Se alguma das origens das hiperarestas que saem de E não estiver pronta, então apenas marcamos E como pronto e seguimos para o próximo vértice da LISTA. Esperamos todas as arestas estarem prontas mesmo quando se trata de um OU, pois como consideramos todos os tempos igual a 1, se um caminho andar e o outro não, então pode ocorrer de o cálculo do paralelismo para aquele nível se dar de forma errada. O maior paralelismo é determinado pelo tamanho da maior das LISTAS do VETOR em qualquer momento da execução do algoritmo.

4.4 Pseudo-código do Algoritmo

```
1 nivel = 0
2 maiorParalel = 0
3 Enquanto Vetor não está vazio:
4       nivel = nivel + 1
5             Para cada Lista do Vetor:
6                   Para cada nodo da Lista:
7                         Se nodo.nivel <= nivel:
8                               Se quantidade aresta que saem <= 1:
9                                     Se VerificaProntos(nodo):
10                                          RemoveOrigens(Lista, nodo, aresta)
11                                          remove nodo da Lista
12                                          Para cada vértice destino da aresta que sai do nodo:
13                                                adiciona vértice na Lista, na posição do nodo
14                                    Senão:
15                                          marca nodo como pronto
16                              Senão:
17                                    Se VerificaProntos(nodo):
18                                          Para cada aresta que sai de nodo:
19                                                ReplicaLista(Lista)
20                                                RemoveOrigens(Lista, nodo, aresta)
21                                                remove nodo da nova_lista
22                                                Para cada vértice destino da aresta:
23                                                      adiciona vértice na nova lista na posição do nodo
24                                          Remove Lista do Vetor
25                                    Senão:
26                                          marca nodo como pronto LimpaVetor()
27      VerificaMaiorParalel(maiorParalel)
28 imprime: "Maior paralelismo encontrado: " + maiorParalel
```

Figura 4.3: Pseudo-código do algoritmo estudado

4.5 Simulação e demonstração

Nesta seção demonstraremos um exemplo de simulação do algoritmo de força bruta estudado. Para isso vamos considerar passo a passo da execução do algoritmo. Por exemplo, para um passo X teremos a descrição do que acontece e o vetor de listas. No vetor de listas serão indicados todos os vértices ali presentes e o tempo da execução dos mesmos. Por exemplo, o vetor de listas V (3): o vetor possui apenas V e 3 é o tempo de execução do vértice V.

4.5.1 Exemplo de simulação de um hipergrafo

Vamos simular o seguinte hipergrafo:

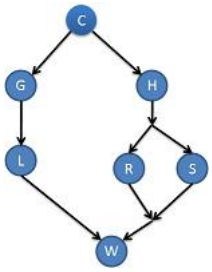

Figura 4.4: Exemplo de um hipergrafo para a simulação do algoritmo de Força Bruta

Passo 1: Inicia pelo vértice inicial C. Vetor de Listas:
C (1)

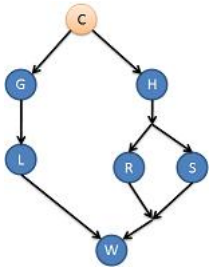

Figura 4.5: Passo 1

Passo 2: Na saída do vértice C temos duas arestas, temos um OU com dois destinos, então vamos trabalhar com duas lista para representar as duas escolhas. Vetor de listas:

H (2)

G (2)

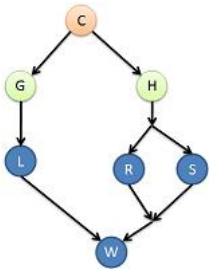

Figura 4.6: Passo 2

Passo 3: Do vértice H vai uma aresta E com dois vértices, então substitui os dois vértices no lugares de H. Vetor de listas:

S (3) R (3)

G (2)

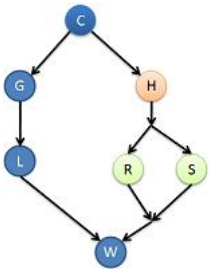

Figura 4.7: Passo 3

Passo 4: Com o vértice G chegamos em L. Vetor de listas:

S (3) R (3)

L (3)

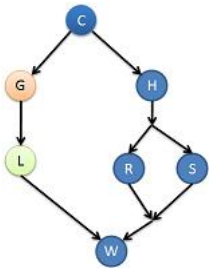

Figura 4.8: Passo 4

Passo 5: Com os vértices S e R chegamos em W. Vetor de listas:

W (4)

L (3)

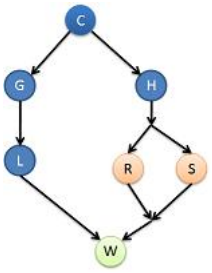

Figura 4.9: Passo 5

Passo 6: Com o vértice L chegamos em W. Vetor de listas:

W (4)

W (4)

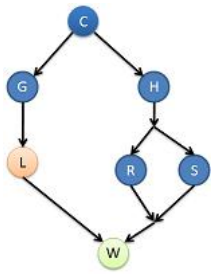

Figura 4.10: Passo 6

Passo 7: O vértice W na primeira lista é finalizado. Vetor de listas:

(vazio)

W (4)

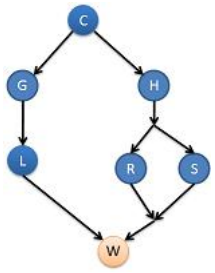

Figura 4.11: Passo 7

Passo 8: O vértice W na segunda lista é finalizado. O algoritmo finaliza, pois a lista de vértices pendentes está vazia assim como a lista de prontos para executar.

Vetor de listas:

(vazio)

(vazio)

O maior paralelismo é dado pelo maior tamanho que alguma lista chegou ao longo de sua execução. Neste exemplo o maior paralelismo é 2, verificado entre os passos 3 e 4 e no tempo de execução 3 do algoritmo, quando a segunda lista atinge esse tamanho.

O vetor de lista tem duas listas, então o fluxo usado como exemplo tem dois caminhos possíveis para execução, seguem abaixo figuras representando os caminhos.

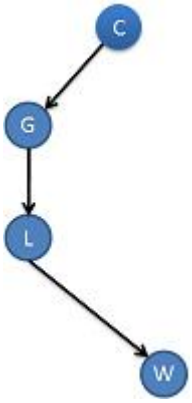

Figura 4.12: Caminho 1

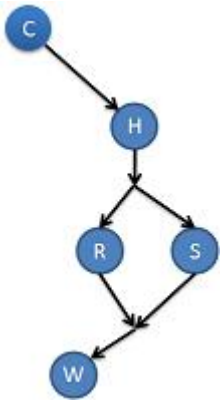

Figura 4.13: Caminho 2

4.6 Justificativa

O algoritmo vai criar uma lista para cada possível caminho de execução. Cada vez que a execução chegar a um vértice OU, a lista será replicada de acordo com a quantidade de arestas que estão saindo do vértice de origem. Com isso, o vetor de lista vai conter todos os possíveis caminhos de execução. Ao passar para o(s) próximo(s) vértice(s), o vértice de origem vai sair da lista para que nela tenha somente os vértices que estão sendo executados em paralelo. Para verificar o maior paralelismo, a cada vértice executado é verificado o tamanho das listas. O maior paralelismo é dado pelo maior tamanho de uma dessas listas ao longo da execução do algoritmo. De acordo com a execução descrita acima, chegamos à conclusão que esse algoritmo é válido.

5 *Conclusão*

Após o estudo feito neste trabalho, é fácil de enxergar como Hipergrafos direcionados podem ser extremamente úteis na modelagem de fluxo de tarefas. Com esta estrutura é possível modelar e enxergar, no nível de estrurturas de dados, uma série de informações relevantes quanto a paralelismo sem a necessidade de estruturas adicionais.

Mostramos um algoritmo de força bruta que explora todos os caminhos possíveis dentro do Hipergrafo e que de fato encontra o maior grau de paralelismo. Percebemos, então, a necessidade de um algoritmo como este, pois, todos nós sabemos a extrema importância do desempenho/otimização de um processo, que pode ser obtida através do paralelismo.

Características dentro da estrutura de dados que foram descritas no trabalho, como as condições E-F-Condição ou OU-B-Condição, permitem que problemas de dependências de dados possam ser resolvidos e que o cálculo do maior paralelismo seja obtido. Assim, é possível calcular a quantidade de recursos dentro de uma máquina para a exploração do paralelismo.

Em resumo, temos o paralelismo com uma grande área de estudo dentro da ciência da computação que ainda necessita de muita pesquisa e avanço, e Hipergrafos direcionados como uma importante alternatica para a modelagem do paralelismo.

Referências Bibliográficas

[1] GUEDES, A. L. P. Hipergrafos direcionados. 2001. Tese (Doutorado) Universidade Federal do Rio de Janeiro.

[2] BOFF, C.; KRENSKI, C.; LORENZET, J. Modelagem de fluxos de trabalho utilizando hipergrafos direcionados. 2009. Monografia de Graduação Universidade Federal do Paraná.

[3] PEDRASSANI, B.; RIBAS, F.; DOI, R. N. Hipergrafos dirigidos à computação paralela. 2008. Monografia de Graduação Universidade Federal do Paraná.

[4] GOMES, E. Transformação de grafo de fluxo de controle para grafo de fluxo de dados para análise e exploração de paralelismo. 2007. Monografia de Graduação Universidade Federal do Paraná.

[5] STALLINGS, W. *Organização e Arquitetura de Computadores*. [S.l.]: Pearson Education, 2003.

[6] PILLAY, V. Hypergraph. Acessado em dezembro de 2009. Disponível em: <http://en.wikipedia.org/wiki/Hypergraph>.

[7] LANG, Y. Paralelismo. Acessado em dezembro de 2009. Disponível em: <http://pt.wikipedia.org/wiki/Paralelismo>.

[8] LANG, Y. Força bruta. Acessado em dezembro de 2009. Disponível em: <http://pt.wikipedia.org/wiki/Força_bruta>.

[9] FELTRIM, D. Estruturas de controle (fluxo). Acessado em dezembro de 2009. Disponível em: <http://pt.wikipedia.org/wiki/Estrutura_de_controle>.

[10] DONADELLI, J. Algoritmos e teoria dos grafos. Acessado em dezembro de 2009. Disponível em: <http://www.inf.ufpr.br/jair/ci065>.