Implementação e Análise de Algoritmos BSP/CGM em um *Beowulf* e no InteGrade

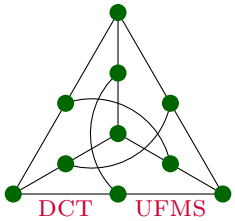
Christiane Nishibe

Dissertação de Mestrado

Orientação: Prof. Dr. Edson Norberto Cáceres

Área de Concentração: Ciência da Computação

Durante o desenvolvimento deste trabalho a autora recebeu apoio financeiro da Fundect.



Departamento de Computação e Estatística
Centro de Ciências Exatas e Tecnologia
Universidade Federal de Mato Grosso do Sul
Junho de 2009

Implementação e Análise de Algoritmos BSP/CGM em um *Beowulf* e no InteGrade

Este exemplar corresponde à redação final da dissertação devidamente corrigida e defendida por Christiane Nishibe e aprovada pela comissão julgadora.

Campo Grande/MS, Junho de 2009.

Banca Examinadora:

- Prof. Dr. Edson Norberto Cáceres (Orientador) (DCT-UFMS)
- Prof. Dr. Siang Wun Song (IME-USP)
- Prof. Dr. Henrique Mongelli (DCT-UFMS)

Agradecimentos

Ao meu orientador, professor Dr. Edson Noberto Cáceres, pela dedicação, paciência e principalmente por todo conhecimento e atenção desde a iniciação científica até a conclusão deste trabalho.

Ao Prof. Henrique Mongelli, por sempre me ajudar, compartilhando seu conhecimento nos meus momentos de dúvidas.

Ao Prof. Siang Wun Song por todas as dicas e contribuições para a conlusão do trabalho, além de toda a receptividade nas minhas passagens por São Paulo.

Aos amigos pelas divertidas conversas, pelos bons momentos de desconcentração e acima de tudo por todo apoio.

A todos que contribuíram, direta ou indiretamente, no desenvolvimento deste trabalho: Muito Obrigada!

Resumo

Com o avanço da ciência e da tecnologia nas mais diversas áreas surgiram problemas que necessitam cada vez mais de alto poder computacional. Inicialmente, para resolver esses problemas, eram utilizados computadores paralelos de grande porte e elevado custo. Em seguida, no entanto, tornou-se mais eficiente e barato montar *clusters* com PCs que trabalham em conjunto para oferecer um alto poder de processamento a um custo menor que o método anterior. Recentemente, porém, vem sendo desenvolvido a ideia de interligar *clusters* dispersos geograficamente, formando uma única grade computacional e uma dessas propostas é o *middleware* InteGrade. Visto que a utilização de grades computacionais para elevar o poder de processamento disponível para a solução dos mais diversos problemas vem se tornando mais comum, o principal objetivo do nosso trabalho é avaliar o desempenho do InteGrade em relação ao *cluster*. Para fazer essa avaliação, estudamos problemas paralelos com diferentes aspectos de computação e de comunicação e os implementamos utilizando o modelo BSP/CGM (*Bulk Synchronous Parallel/Coarse Grained Multicomputer*). Entre os problemas estudados estão o Problema da Mochila 0-1, o Problema da Árvore Geradora e por fim o Problema do Fecho Transitivo.

Todos os algoritmos foram implementados utilizando o padrão MPI (*Message Passing Interface*) e a linguagem C.

Palavras-chave: Algoritmos BSP/CGM, Algoritmos Paralelos, Problema da Mochila 0-1, Árvore Geradora, Fecho Transitivo.

Abstract

As the science and technology advanced in all diverse areas, problems which require more and more computer power to be solved were risen. In the beginning, these problems were solved by high performance parallel computers which were huge and very expensive. After that, though, making clusters with PCs which worked together to offer a higher processing capacity at a lower cost than the previous one became more efficient and inexpensive. Recently, however, the idea of linking geographically spread clusters making a single computer net, which one of the proposals is the middleware InteGrade, has been developed. Once using grid computing to improve the available processing capacity to solve the range of the most different problems has become more common, the main objective of our work is to compare the performance between InteGrade to the cluster's. To value this we have studied the parallel problems with different computer and communication aspects and implemented it using the BSP/CGM model (Bulk Synchronous Parallel/Coarse Grained Multicomputer). The studied problems were 0-1 Knapsack Problem, the Spanning Tree Problem and Transitive Closure Problem.

All the algorithms were implemented using the MPI pattern (Message Passing Interface) and C language.

Keywords: BSP/CGM Algorithms, Parallel Algorithms, 0-1 Knapsack Problem, Spanning Tree, Transitive Closure.

Conteúdo

Capítulo 1

Introdução

O conceito de paralelizar atividades para ganhar tempo existe há muito tempo. Mas, do ponto de vista computacional, a computação paralela surgiu em meados da década de 1980. Nessa época, os primeiros computadores começaram a ser programados como máquinas paralelas reais. Três razões fizeram com que a computação paralela se tornasse prática e se desenvolvesse rapidamente. A primeira delas foi o avanço nas tecnologias de hardware. A segunda, o desenvolvimento de softwares para esses sistemas. E a terceira, o desenvolvimento da área de algoritmos paralelos.

A computação paralela é utilizada, principalmente, na resolução de problemas em que o volume de cálculos e dados é grande. Além disso, a computação paralela vem permitindo que problemas complexos sejam solucionados e aplicações de alto desempenho sejam desenvolvidas.

Apesar do grande avanço tecnológico, máquinas altamente especializadas e com alto desempenho computacional são extremamente caras. Ao invés de gastar grandes quantias em *clusters* dedicados, que passam a maior parte do tempo ociosos, o *middleware* InteGrade permite a utilização de recursos computacionais já existentes para a realização de todo trabalho que demande alto poder de processamento.

A fim de avaliar o comportamento do InteGrade com diferentes problemas paralelos implementamos algoritmos com diferentes classes de rodadas de comunicação, diferentes tamanhos de dados trocados entre os processadores além de variarmos a quantidade de processadores que cada processador trocava dados. Sendo assim, para obter uma análise do comportamento do InteGrade, projetamos algoritmos paralelos e implementamos no InteGrade e num *Beowulf* para três diferentes problemas: Mochila 0-1, Árvore Geradora e Fecho Transitivo.

O Problema da Mochila 0-1 é um problema clássico de otimização combinatória sendo um problema de programação inteira com uma única restrição. Possui assim, uma enorme quantidade de aplicações [24, 32, 48]. Em princípio, qualquer problema de programação inteira pode ser transformado neste problema [24]. As dificuldades da solução do Problema da Mochila 0-1 são as dificuldades típicas da programação inteira. Bons algoritmos para a solução deste problema são interessantes para a área.

A Árvore Geradora de um grafo é um problema fundamental em Teoria dos Grafos. Tal estrutura geralmente aparece como subrotina de problemas mais complexos. Entre eles, componentes biconexos e decomposição em orelhas ou, ainda, no teste de planaridade de grafos [7].

O Fecho Transitivo de um grafo dirigido é um importante subproblema de diversas aplicações computacionais em redes de computadores, sistemas paralelos e distribuídos, banco de dados e no projeto de compiladores [40].

Os algoritmos paralelos que foram implementados nesse trabalho utilizaram o modelo BSP/CGM. O modelo BSP (*Bulk Synchronous Parallel*) foi proposto por Valiant [49] em 1990. Além de ser um dos modelos realísticos mais importantes, foi o primeiro a considerar o custo de comunicação. O modelo CGM (*Coarse Grained Multicomputer*), apresentado por Dehne *et al* [21], e o termo "granularidade grossa" vêm do fato que o tamanho do problema n é estritamente maior que o número de processadores p, ou seja, $n \gg p$.

No Capítulo 2, apresentamos conceitos em teoria dos grafos utilizados no desenvolvimento do trabalho. Descrevemos também os principais modelos computacionais paralelos e o InteGrade, descrevendo suas funcionalidades principais, além de descrever o ambiente computacional.

O Capítulo 3 descreve o Problema da Mochila 0-1 e um algoritmo sequencial para resolver esse problema. O capítulo mostra a proposta de uma solução paralela, utilizando o modelo BSP/CGM com programação dinâmica. Apresentamos, também, os principais passos do algoritmos além de analisarmos a complexidade e a corretude do algoritmo proposto.

No Capítulo 4, apresentamos a implementação do algoritmo BSP/CGM proposto por Cáceres *et al* [8] para a computação da árvore geradora de um grafo bipartido. Apresentamos, também, os tempos obtidos com a execução do programa bem como uma análise de seu comportamento no InteGrade.

O Capítulo 5 apresenta os resultados obtidos com três algoritmos BSP/CGM para o Problema do Fecho Transitivo.

No Capítulo 6, apresentamos as conclusões e comentários finais.

Capítulo 2

Fundamentos

Este capítulo apresentará uma visão geral de alguns conceitos que serão utilizados no decorrer do trabalho.

Na Seção 2.1, discutiremos sobre grafos, suas definições e representações. Na Seção 2.2, apresentaremos os principais modelos de computação paralela. Na Seção 2.3, descreveremos o InteGrade, sua arquitetura e as bibliotecas que possui. Na Seção 2.4, mostraremos o ambiente computacional que será utilizado na execução dos algoritmos.

2.1 Notacão e Terminologia

Esta seção descreve alguns conceitos de Teoria dos Grafos, utilizados neste trabalho que são encontradas em [47].

Definição 1 *Um grafo $G = (V, E)$ corresponde a um conjunto finito não-vazio V de elementos chamados vértices e um conjunto E de pares de vértices de V, chamados arestas.*

Se as arestas forem pares ordenados (v, w), o grafo é denominado **orientado** ou **digrafo**. Se as arestas forem pares não-ordenados (v, w), o grafo é dito **não-orientado**. Se (v, w) é uma aresta não-orientada, v e w são **adjacentes**. Se (v, w) é uma aresta orientada, v é **predecessor** de w e w é **sucessor** de v. Neste trabalho, exceto quando especificado, consideramos que $|V| = n$ e $|E| = m$.

Definição 2 *Em um grafo não-orientado, define-se **grau** de um vértice $v \in V$, denotado por d_v, como sendo o número de arestas $e \in E$ adjacentes à v. Em um grafo orientado, o **grau de entrada** de v é o número de arestas convergentes a v. O **grau de saída** de v é o número de arestas divergentes de v.*

Definição 3 *Uma sequência de vértices $v_1, \ldots, v_k$, tal que $(v_i, v_{i+1}) \in E$, $1 \leq i \leq k-1$ é denominado **caminho** de v_1 a v_k. Diz-se então que v_1 **alcança** ou **atinge** v_k.*

Definição 4 *Um **ciclo**, ou **circuito**, é um caminho onde $v_1 = V_k$. Um grafo que não possui ciclos é denominado **acíclico**.*

Definição 5 *Um **sub-grafo** $G_2 = (V_2, E_2)$ de um grafo $G_1 = (V_1, E_1)$ é um grafo tal que $V_2 \subseteq V_1$ e $E_2 \subseteq E_1$.*

Definição 6 *Seja $G = (V, E)$ um grafo não-orientado. Um grafo é denominado **conexo** se, para todo par de vértices v e w em V, existe um caminho entre v e w.*

Definição 7 *Seja S um conjunto e $S' \subseteq S$. Diz-se que S' é **maximal** em relação a uma certa propriedade P, quando S' satisfaz a propriedade P e não existe subconjunto $S'' \supset S'$, que também satifaz P. Ou seja, S' não está propriamente contido em nenhum subconjunto de S que satisfaz P.*

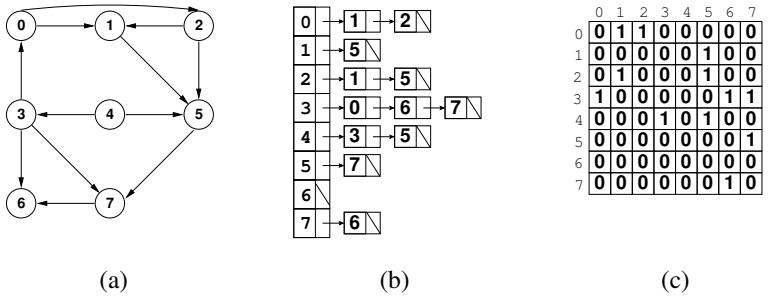
Definição 8 *Denominam-se **componentes conexos** de um grafo G aos sub-grafos maximais de G que sejam conexos. A propriedade P, neste caso, é equivalente a ser conexo.*

Definição 9 *Denomina-se **árvore** um grafo $T = (V, E)$ que seja acíclico e conexo. Se um vértice v da árvore T possuir grau igual a 1, então o v é uma **folha**. Caso contrário, v é um **vértice interno**.*

Definição 10 *Um grafo $G = (V, E)$ é **bipartido** quando seu conjunto de vértices V pode ser particionado em dois subconjuntos V_1 e V_2, tais que toda aresta de G une um vértice de V_1 a outro de V_2.*

2.1.1 Representações de Grafos

As duas formas utilizadas neste trabalho para se representar digrafos são: matriz de adjacências e lista de adjacências. A Figura 2.1 ilustra um digrafo $D = (V, E)$ (a), a lista de adjacências (b) e a matriz de adjacências (c).



(a) (b) (c)

Figura 2.1: Representação computacional de um grafo.

A lista de adjacências em geral, é usada quando m é muito menor que n^2, ou seja, são utilizados na representação de grafos esparsos. Quando os grafos são densos, m

está próximo de n^2, ou quando precisamos responder com rapidez se existe uma aresta conectando dois vértices dados, uma representação utilizando matriz de adjacências é mais eficiente.

A representação de um grafo $G = (V, E)$ em uma lista de adjacências, consiste em um arranjo de n listas, uma para cada vértice em V. Para cada $u \in V$, a lista de adjacências, contém ponteiros para todos os vértices v tais que existe uma aresta $(u, v) \in E$.

Utilizando uma matriz de adjacências para representar o grafo G, supomos que os vértices são rotulados de $0, \ldots, n-1$. Então sua representação consiste em uma matriz $M_{n \times n}$, tal que $M_{i,j} = 1$ se $(i, j) \in E$ e 0 caso contrário.

2.2 Modelos de Computação Paralela

Na década de 1980 começaram a surgir os computadores paralelos. O principal objetivo da computação paralela é solucionar problemas com grande quantidade de informação no menor tempo possível. Tais problemas aparecem em diversas áreas, como na previsão do tempo, no processamento de imagens ou ainda modelagem e simulação de sistemas.

Segundo JáJá [35], um computador paralelo é simplesmente uma coleção de processadores, tipicamente do mesmo tipo, interconectados de uma certa maneira a permitir a coordenação de suas atividades e a troca de dados. No entanto, a criação de algoritmos paralelos modifica-se sensivelmente, pois, na criação de algoritmos paralelos, é preciso considerar a arquitetura paralela a ser utilizada, uma vez que esta afeta o desempenho dos algoritmos.

2.2.1 Desempenho

Um sistema de computação paralela depende de um grande conjunto de parâmetros, tais como o a quantidade de processadores, o tamanho das memórias locais, o esquema de comunicação e os protocolos de sincronização; tornando o projeto, a avaliação e a análise de algoritmos paralelos mais complexa do que no modelo sequencial. O tempo de execução de um algoritmo paralelo não depende apenas do tamanho da entrada de dados, mas também da arquitetura utilizada e da quantidade de processadores utilizados.

O processamento paralelo proporciona um aumento de capacidade e velocidade de processamento e para quantificar esse aumento, utiliza-se diferentes parâmetros, entre eles *speedup* e eficiência.

Seja T_1 o tempo gasto pelo melhor algoritmo sequencial para resolver um determinado problema e T_p o tempo gasto por um algoritmo paralelo utilizando p processadores para resolver o mesmo problema. O ***speedup*** (S_p) representa quão mais rápido é o algoritmo paralelo em relação ao sequencial.

$$S_p = \frac{T_1}{T_p}$$

O caso ótimo ocorre quando $S_p = p$, ou seja, na medida que aumentamos o número de processadores, aumenta-se diretamente a velocidade de processamento. Algumas vezes o *speedup* fica acima de p, é o chamado de *speedup* superlinear.

A **eficiência** do processador mede a utilização efetiva dos p processadores e é definida por:

$$E_p = \frac{S_p}{p}$$

Como normalmente o *speedup* é menor que p, a eficiência, nestes casos, fica entre 0 e 1. Um valor de $E_p(N)$ aproximadamente igual a 1, para algum p, indica que o algoritmo paralelo executa aproximadamente p vezes mais rápido usando p processadores do que o faria usando apenas 1 processador. Dificilmente obtém-se eficiência igual a 1 pois há perdas na paralelização do algoritmo, com sobrecarga de comunicação e sincronização entre os processadores.

2.2.2 Modelos de Computação Paralela

No modelo sequencial, podemos estabelecer uma relação entre os desempenhos das implementações e dos seus respectivos algoritmos através de suas complexidades. Na computação paralela, não temos um modelo de computação único como acontece no modelo sequencial. Isto faz com que cada algoritmo seja analisado, em termos de complexidade de tempo e uso de recursos, dentro do seu modelo. Sendo assim, há alguns modelos que servem de paradigma para simplificar a análise dos algoritmos paralelos.

Modelo PRAM

O primeiro modelo de computação paralela existente foi o *Parallel Random Access Machine* - PRAM. Os algoritmos projetados para este modelo não levam em conta a comunicação e assumem que o número de processadores, p, disponíveis é da mesma ordem do tamanho do problema, N, ou seja, é um problema de "granularidade fina". Quando esses algoritmos eram implementados nas máquinas paralelas existentes, geralmente os *speedups* obtidos eram muitas vezes desapontadores. Em muitos casos, o custo (tempo multiplicado pelo número de processadores) obtido pelos algoritmos paralelos era bastante superior ao do algoritmo sequencial. Um outro ponto, também crucial, era a falta de portabilidade das implementações (muito dependente da topologia). O mesmo não ocorre quando esses algoritmos são implementados nos *clusters* e *Beowulfs*, pois nesses sistemas temos grandes quantidades de memória local e o número de processadores p é estritamente menor que a quantidade de dados N, ou seja, $p << \frac{N}{p}$.

O modelo PRAM é formado por uma coleção de processadores, cada um com uma memória local, executando em paralelo e comunicando-se através de uma memória global compartilhada. O modelo possui variações baseadas na forma como a manipulação concorrente a uma mesma posição de memória global é efetuado:

- EREW *Exclusive Read, Exclusive Write*: leitura e escrita exclusiva, ou seja, não permite qualquer acesso simultâneo a uma mesma posição de memória.
- CREW *Concurrent Read, Exclusive Write*: permite a leitura simultânea de uma mesma posição de memória por mais de um processador, mas não a escrita simultânea.
- CRCW *Concurrent Read, Concurrent Write*: permite simultaneidade na leitura e na escrita na mesma posição de memória por diversos processadores, porém são necessários critérios para solucionar o problema de escrita concorrente. Entre as políticas adotadas para decidir qual dos processadores efetuará efetivamente a escrita temos:
  - escrita comum: todos os processadores devem escrever o mesmo valor.
  - escrita arbitrária: permite que qualquer processador obtenha sucesso na escrita.
  - escrita proprietária: define uma prioridade entre os processadores e aquele com maior prioridade terá sucesso na escrita.

Modelo BSP

No início dos anos 90, Valiant [49] introduziu um modelo simples que fornece uma previsão razoável do desempenho da implementação dos algoritmos projetados nesse modelo nas máquinas paralelas existentes, principalmente as de memória distribuída. Esse modelo de "granularidade grossa", denominado *Bulk Synchronous Parallel* - BSP, além de ser um dos modelos realísticos mais importantes, foi um dos primeiros a considerar os custos de comunicação e a abstrair as características de uma máquina paralela em um pequeno número de parâmetros.

Uma máquina BSP consiste de um conjunto de p processadores com memória local. A comunicação é feita através de algum meio de interconexão, gerenciados por um roteador, que envia mensagens ponto-a-ponto entre pares de processadores. Além disso, o modelo oferece facilidades de sincronização entre os processadores.

Um algoritmo BSP consiste numa sequência de superpassos separados por barreiras de sincronização. Em um superpasso, a cada processador é atribuído um conjunto de operações independentes, consistindo de uma combinação de passos de computação, usando dados disponibilizados localmente no início do superpasso, e passos de comunicação, através de instruções de envio e recebimento de mensagens. A Figura 2.2 ilustra os superpassos de um algoritmo BSP.

Os superpassos são separados por uma barreira de sincronização a cada L unidades de tempo. A barreira de sincronização é realizada para determinar se o superpasso foi

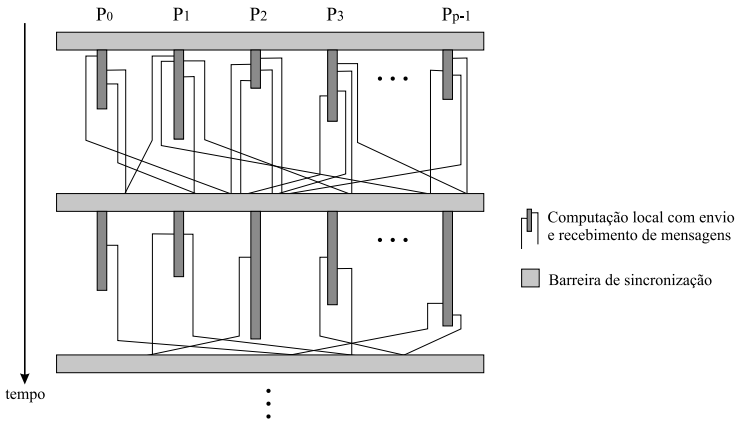


Figura 2.2: Algoritmo BSP [28].

completado por todos os processadores; assegurando que os dados recebidos pelos processadores estarão disponíveis para o próximo superpasso.

O modelo possui os seguintes parâmetros:

- N: tamanho do problema;
- p: número de processadores com memória local;
- L: periodicidade ou tempo máximo de um superpasso;
- g: taxa de eficiência de computação e comunicação; correspondente à razão entre a capacidade computacional e a capacidade de comunicação do sistema.

Neste modelo uma h-relação em um superpasso corresponde ao envio e/ou recebimento de, no máximo, h mensagens em cada processador. A resposta a uma mensagem enviada em um superpasso somente será utilizada no próximo superpasso.

Modelo CGM

O modelo *Coarse Grained Multicomputer* - CGM, proposto por Dehne *et al* [21], tinha o propósito de ser um modelo próximo às máquinas paralelas com memória distribuída existentes. Este modelo é parecido com o BSP, no entanto utiliza apenas dois parâmetros: o número de processadores p e o tamanho da entrada N.

Uma máquina CGM consiste de um conjunto de p processadores, cada um com memória local de tamanho $O(\frac{N}{p})$. Os processadores podem estar conectados por qualquer meio de interconexão. O termo "granularidade grossa" (*coarse grained*) vem do fato de que o tamanho do problema, N, é consideravelmente maior que o número de processadores, ou seja, $\frac{N}{p} >> p$.

Um algoritmo CGM consiste de uma sequência de rodadas, alternando fases bem definidas de computação local e comunicação global, separadas por uma barreira de sincronização (Figura 2.3). Normalmente, durante uma rodada de computação é utilizado o

melhor algoritmo sequencial para o processamento dos dados disponibilizados localmente. Em cada rodada de comunicação, cada processador troca no máximo $O(\frac{N}{p})$ dados com outros processadores.

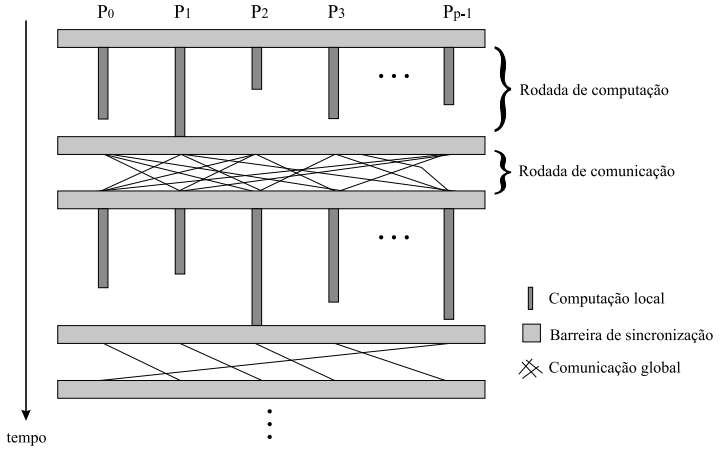


Figura 2.3: Algoritmo CGM [28].

Um algoritmo CGM é um caso especial de um algoritmo BSP onde todas as operações de comunicação de um superpasso são feitas na h-relação. Conforme observado por Dehne [20], os algoritmos CGM, quando implementados, se comportam bem e exibem *speedups* similares àqueles previstos em suas análises. Para estes algoritmos, o maior objetivo é minimizar o número de superpassos e a quantidade de computação local.

2.3 InteGrade

O Projeto InteGrade (`www.integrade.org.br`) tem como objetivo a construção de um *middleware* que permita a implantação de grades sobre recursos computacionais não dedicados, fazendo uso da capacidade ociosa normalmente disponível em instituições públicas e privadas para a resolução de problemas que demandem alto poder computacional. O InteGrade é um projeto desenvolvido em conjunto por cinco instituições: Departamento de Ciência da Computação (IME-USP), Departamento de Informática (PUC-RIO), Departamento de Informática (UFMA), Instituto de Informática (UFG) e Departamento de Computação e Estatística (UFMS).

O *middleware* InteGrade [26] permite a formação de grades computacionais oportunistas, ou seja, permite a formação de um aglomerado de computadores, a partir de máquinas já existentes em um grupo de instituições.

Os serviços administrativos que são executados nos nós de gerenciamento da grade são escritos na linguagem Java de forma a oferecer a maior portabilidade possível. Já os componentes executados nas máquinas dos usuários que compartilham parte de seus recursos com a grade são desenvolvidos nas linguagens C e Lua para minimizar o consumo de memória e assim não prejudicar a qualidade de serviço desses usuários. A comunicação entre os nós da grade é feita através do padrão CORBA [36].

A unidade estrutural de uma grade InteGrade é o aglomerado (*cluster*). Um aglomerado é um conjunto de máquinas agrupadas por um determinado critério, como pertinência a um domínio administrativo. Tipicamente o aglomerado explora a localidade de rede. Entretanto tal organização é totalmente arbitrária e os aglomerados podem conter máquinas presentes em redes diferentes.

Na Figura 2.4 podemos observar alguns elementos de um aglomerado InteGrade. Estes módulos são responsáveis por diversas tarefas necessárias à grade.

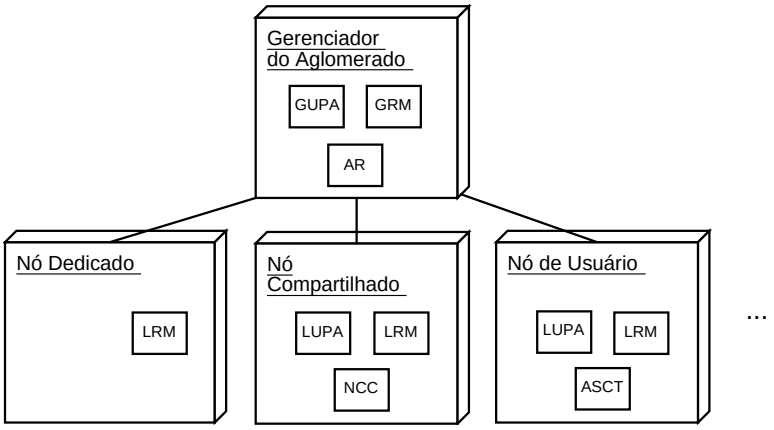


Figura 2.4: Arquitetura intra-aglomerado do InteGrade [27].

Os principais componentes da arquitetura do InteGrade são descritos a seguir.

- **LRM** *(Local Resource Manager)*: executado em todas as máquinas que compartilham seus recursos com a grade. Este componente é responsável pela coleta e distribuição das informações referentes à disponibilidade de recursos locais, permitindo a execução e controle de aplicações submetidas por usuários da grade.

- **GRM** *(Global Resource Manager)*: módulo responsável pelo gerenciamento dos recursos de um aglomerado, pela interação com outros aglomerados e pelo escalonamento da execução de aplicações. O GRM mantém uma lista dos LRMs ativos e, ao receber uma requisição para execução de uma aplicação, escolhe um LRM que atenda às necessidades da aplicação.

- **LUPA** *(Local Usage Pattern Analyzer)*: responsável pela análise e monitoramento dos padrões de uso. O LUPA é utilizado junto com o LRM nos nós provedores de recursos. O LUPA fornece subsídio às decisões de escalonamento, fornecendo uma perspectiva probabilística sobre a disponibilidade de recursos em cada nó.

- **GUPA** *(Global Usage Pattern Analyzer)*: auxilia o GRM nas decisões de escalonamento ao fornecer informações coletadas pelos diversos LUPAs.

- **AR** *(Application Repository)*: armazena de forma segura os executáveis de aplicações submetidas por usuários para execução na grade.

- **EM** *(Execution Manager)*: responsável por gerenciar os *checkpoints* das aplicações e acompanhar a execução de todas as tarefas das aplicações em execução em um dos aglomerados da grade. O LRM e GRM informam ao EM quando aplicações iniciam e terminam a sua execução e o EM coordena o processo de reinicialização de tarefas que estavam executando em nós que falharam ou se tornaram indisponíveis.
- **ASCT** *(Application Submission and Control Tool)*: é a ferramenta que permite que usuários da grade realizem requisições de execução de aplicações, controlem a execução destas aplicações e visualizem seus resultados.

Diferentemente de outras grades já existentes, o InteGrade não visa apenas aplicações *bag-of-tasks*. Mesmo considerando a grande latência existente numa grade, aplicações que utilizam o modelo BSP/CGM estão sendo projetados visando minimizar o número de rodadas de comunicação dos algoritmos e o tamanho das mensagens trocadas pelos nós do InteGrade. Entre as aplicações desenvolvidas estão:

- **Algoritmos Básicos e Algoritmos de Ordenação**: Foram desenvolvidos algoritmos básicos de soma e soma de prefixos, além de um algoritmo de ordenação, pois muitos dos algoritmos BSP necessitam de algoritmos de ordenação eficientes em suas sub-rotinas.
- **Um resolvedor SAT para o Integrade**: O Problema de Satisfabilidade-SAT e um dos problemas NP-completos mais estudados atualmente e se resume a, "dado uma fórmula do cálculo proposicional, encontrar uma atribuição de valores atômicos que tornam a fórmula verdadeira". O objetivo é implementar um resolvedor para o SAT que seja capaz de dividir o problema para otimizar o tempo de busca por uma solução usando programação distribuída, no modelo BSP.
- **Multiplicação de Matrizes**: Utilizada num grande número de problemas. Obter uma solução eficiente para esse problema no InteGrade ampliará o número de potenciais usuários. Utilizando algoritmos sistólicos para multiplicação de matrizes obteve-se bons *speedups*.
- **Programação Dinâmica e Algoritmos Gulosos**: Considerando que os algoritmos de similaridade de sequências e maior subsequência comum (LCS) de Alves *et al* [2] projetados para o modelo BSP/CGM foram implementados em um *Beowulf* e trocam um número pequeno de mensagens em cada rodada de comunicação, essas aplicações estão sendo implementadas no InteGrade. Assim como, os algoritmos para o Problema da Mochila 0-1 [11, 12].
- **Implementacão de Algoritmos FPT**: A complexidade parametrizada é um método promissor para se lidar com a intratabilidade de alguns problemas, principalmente aqueles cuja entrada pode ser dividida em uma parte principal e um parâmetro. A parte principal da entrada contribui polinomialmente na complexidade total do problema, enquanto a aparentemente inevitável explosão combinatorial fica confinada ao parâmetro. Para esta classe de problemas, há um algoritmo FPT para o problema da k-Cobertura por vértices utilizando o *middleware* do InteGrade apresentado por Mongelli e Sakamoto [39].

A versão mais recente do InteGrade permite o desenvolvimento de aplicações paramétricas (*bag-of-tasks*), MPI e BSP.

Aplicações paralelas escritas em C/C++ do tipo BSP utiliza a biblioteca BSPlib do InteGrade, que usa mesma API da implementação de Oxford (`http://www.bsp-worldwide.org/implmnts/oxtool`). Para possibilitar a execução de programas escritos em C/C++, Fortran 77 e Fortran 90 usando MPI, o InteGrade implementa uma versão adaptada da biblioteca MPICH2 (`http://www.mcs.anl.gov/research/projects/mpich2`).

A implementação do MPI pelo InteGrade facilita sua utilização pois não há a necessidade de modificação dos códigos-fontes já existentes.

2.4 Ambiente Computacional

Todas as experiências foram realizadas no *cluster* do DCT-UFMS, que possui 11 máquinas com diferentes velocidades de processamento e memória. A Tabela 2.1 mostra detalhadamente a configuração de cada máquina. A comunicação entre os processadores é realizada através de um *switch* Gigabit Ethernet. Cada nó executava o sistema operacional Fedora 6, gcc 4.1.2, LAM/MPI 7.1.2 e InteGrade 0.4.

Máquinas	Processador	Cache	Memória
01	AMD 1.5 GHz	256 KB	1.5 GB
02	P4 2.8 GHz	1024 KB	2 GB
03	P4 2.6 GHz	512 KB	2 GB
04	P4 2.8 GHz	512 KB	2 GB
05	P4 2.6 GHz	512 KB	2 GB
06	P4 1.8 GHz	512 KB	2 GB
07	P4 2.6 GHz	512 KB	2 GB
08	AMD 1.6 GHz	256 KB	1 GB
09	AMD 1.6 GHz	256 KB	1 GB
10	AMD 1.6 GHz	256 KB	1 GB
11	AMD 1.6 GHz	256 KB	1 GB

Tabela 2.1: Configuração das máquinas do *cluster*.

Os algoritmos descritos neste trabalho foram implementados utilizando a linguagem C e a biblioteca MPI para troca de mensagens. O MPI foi escolhido por se tratar de um padrão na programação por troca de mensagens, garantido a portabilidade do programa.

Os resultados obtidos, em segundos, não abrangem os tempos gastos com a distribuição inicial dos dados e o envio dos resultados ao final do programa. Consideram apenas as etapas de processamento local e comunicação.

Capítulo 3

Problema da Mochila 0-1

Neste capítulo, apresentaremos um algoritmo paralelo para solucionar o Problema da Mochila 0-1 que utiliza programação dinâmica em sua solução e baseia-se na ideia do algoritmo apresentado por Alves *et al* [2]. Na Seção 3.1 definiremos o problema. Na Seção 3.2, descreveremos um algoritmo sequencial com programação dinâmica para solucionar o problema. Na Seção 3.3, apresentaremos o algoritmo de frente de onda que utiliza programação dinâmica para solucionar o Problema da Mochila 0-1 gastando $O(p)$ rodadas de comunicação, onde p é a quantidade de processadores. Por fim, mostraremos, na Seção 3.4, os resultados obtidos com a implementação do algoritmo frente de onda.

3.1 O Problema

O Problema da Mochila 0-1 consiste em um conjunto $S = \{1, 2, \ldots, n\}$ de n itens distintos, cujo i-ésimo item possui um valor v_i, digamos em reais e um peso w_i, digamos em quilos, onde v_i e w_i são inteiros. Seja W um inteiro que representa a capacidade máxima da mochila que será utilizada para transportar os itens. O problema a ser resolvido é: quais itens devem ser escolhidos a fim de encher a mochila com os itens mais valiosos sem exceder a capacidade máxima?

$$\max\{\sum_{i=1}^{n} v_i z_i : \sum_{i=1}^{n} w_i z_i \leq W, z_i \in \{0, 1\}\}.$$

3.2 Algoritmo Sequencial

O Problema da Mochila 0-1 pertence à classe dos problemas NP-*completos* [23]. Contudo, este problema pode ser resolvido sequencialmente em tempo $O(nW)$. Este limite não é polinomial para o tamanho da entrada visto que $\lg W$ bits são necessários para codificar a entrada W. Esta solução é chamada de *pseudo-polinomial* [23].

Existem basicamente duas abordagens para encontrar a solução exata do Problema da Mochila 0-1: programação dinâmica (PD) e *branch-and-bound* (B&B). Quando os parâmetros v_i e w_i são gerados independentemente e temos um problema muito grande, a aproximação (B&B) é, na média, mais eficiente quando implementada em máquinas sequenciais [38]. Quando os parâmetros estão interligados, a aproximação PD comporta-se melhor do que B&B [15, 16]. O primeiro algoritmo para o problema da mochila baseado em programação dinâmica foi desenvolvido por Gilmore e Gomory [25].

A programação dinâmica é uma técnica para a solução de vários problemas de decisão e otimização. A metodologia da programação dinâmica decompõe o problema que está sendo tratado numa sequência de passos de decisão ou otimização inter-relacionados, que são solucionados um após o outro. A solução ótima para um problema é obtida através da decomposição do problema em subproblemas, computando a solução ótima para cada subproblema e recombinando essas soluções para obter o resultado ótimo para a solução global do problema.

Diferentemente dos outros métodos de otimização, tais como programação linear ou *branch-and-bound*, a programação dinâmica não é uma técnica geral. Todo problema de otimização dever ser primeiramente traduzido numa forma mais adequada para que a abordagem de programação dinâmica possa ser utilizada. Essa adequação pode ser bastante difícil e a definição da formulação (em programação dinâmica) que soluciona o problema eficientemente aumenta ainda mais a dificuldade da tarefa.

Utilizando PD, o Algoritmo 1 soluciona sequencialmente o Problema da Mochila 0-1 em tempo $O(nW)$. Denotaremos $f(r, c)$, com $1 \leq r \leq n$ e $0 \leq c \leq W$, os valores da solução ótima para o Problema da Mochila 0-1 com um conjunto de objetos $[1, r]$ e peso c. Consequentemente, $f(n, W)$ é o valor da solução ótima. A relação de recorrência é:

$$f(r, c) = \max\{f(r-1, c), f(r, c-w_r) + v_r\}, c \leq W, 1 \leq r \leq n$$

Algoritmo 1 MOCHILA 0-1

Entrada: (1) v_i e w_i, $1 \leq i \leq n$; e (2) W.
Saída: $f(n, W)$

```
1: for c ← 1 to W do
2:    f(0, c) ← 0;
3: end for
4: for r ← 1 to n do
5:    for c ← 1 to W do
6:       if c < w_k then
7:          f(r, c) ← f(r − 1, c);
8:       else
9:          f(r, c) ← max{f(r, c − w_r) + v_k, f(r − 1, c)};
10:      end if
11:   end for
12: end for
```

Na bibliografia, vários algoritmos paralelos foram propostos para o Problema da Mo-

chila 0-1. A seguir, apresentamos os algoritmos paralelos que estão mais relacionados ao nosso trabalho.

Este problema clássico de otimização combinatorial possui uma enorme quantidade de aplicações [24, 32, 48]. Além disso, este é um problema de programação inteira com uma única restrição. Em princípio, qualquer problema de programação inteira pode ser transformado neste problema [24]. As dificuldades da solução do Problema da Mochila 0-1 são as dificuldades típicas da programação inteira. Bons algoritmos para o Problema da Mochila 0-1 são interessantes para a área de pesquisa em programação inteira.

3.3 Algoritmos Paralelos

Andonov *et al* [4] apresentaram um algoritmo sistólico para solucionar o problema da mochila. A complexidade do algoritmo é $\Theta(nW/p + n)$, utilizando uma topologia em anel com p processadores. Além disso, o algoritmo possui uma fase de *backtracking* para solucionar o problema.

Almeida *et al* [1] proposuseram um algoritmo reduzindo o problema da mochila integral ao problema de caminhos máximos. Os autores também mostraram que algoritmos desenvolvidos para solucionar o Problema da Mochila 0-1 podem ser transformados em soluções eficientes para o problema da mochila integral. Apresentando um novo algoritmo com complexidade $O(W^2/p + n)$ com p processadores em um anel.

Arruda [5] estudou a relação entre paralelismo e redução no espaço de busca através da eliminação de objetos dominados.

Estamos interessados em projetar um algoritmo paralelo que possa ser implementado em máquinas paralelas disponíveis e obter tempos de execução compatíveis aos previstos no modelo BSP/CGM, independentemente do tipo de interconexão de rede utilizado. Apresentamos um algoritmo BSP/CGM para o Problema da Mochila 0-1 que está baseado na idéia *wavefront* (frente de onda) ou comunicação sistólica de Alves *et al* [2]. Sua principal vantagem é que cada processador comunica-se com poucos processadores, o que o torna potencialmente mais conveniente para aplicações em grades computacionais. Nosso algoritmo gasta $O(\frac{nW}{p})$ de computação local e $O(p)$ rodadas de comunicação, onde p é o número de processadores.

3.3.1 O Algoritmo de Frente de Onda

Nesta seção apresentamos um algoritmo BSP/CGM que gasta $O(p)$ rodadas de comunicação para computar a solução do Problema da Mochila 0-1 com n itens e peso máximo W. Utilizaremos p processadores, cada um com memória local $O(W\frac{n}{p})$.

Primeiro daremos a ideia principal para computar a matriz com a solução ótima f utilizando p processadores. Para cada conjunto $S = \{1, 2, \ldots, n\}$ de itens, o vetor w, onde $w[i]$ é o peso de cada item i, e o vetor v, onde $v[i]$ é o valor do item i, são divididos

em p partes, de tamanho $\frac{n}{p}$, e cada processador P_i, $1 \leq i \leq p$, recebe a i-ésima parte de w ($w[(i-1)\frac{n}{p}+1\,..\,i\frac{n}{p}]$) e v ($v[(i-1)\frac{n}{p}+1\,..\,i\frac{n}{p}]$).

Esta ideia está ilustrada na Figura 3.1. A notação P_i^k significa o trabalho do processador P_i na rodada k. Assim, inicialmente P_1 começa a computar na rodada 0. Então P_1 e P_2 podem trabalhar na rodada 1; P_1, P_2 e P_3 na rodada 2, e assim sucessivamente. Em outras palavras, após a computação da k-ésima parte da submatriz f_i (denotada por f_i^k), o processador P_i envia para o processador P_{i+1} os elementos da fronteira direita (coluna mais a direita) de f_i^k. Estes elementos são denotados por R_i^k. Utilizando R_i^k, o processador P_{i+1} pode computar a k-ésima parte da submatriz f_{i+1}. Após $p-1$ rodadas, o processador P_p recebe R_{p-1}^1 e computa a primeira parte da submatriz f_p. Na rodada $2p-2$, o processador P_p recebe R_{p-1}^p e computa a p-ésima parte da submatriz f_p e termina a computação.

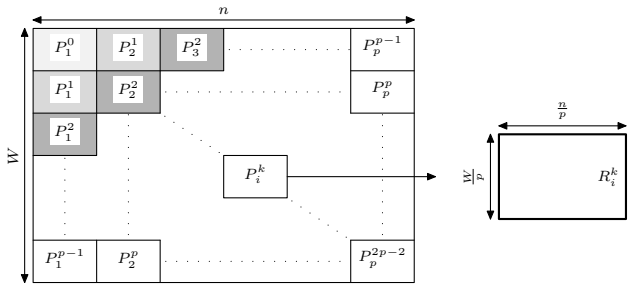


Figura 3.1: Um escalonamento de $O(p)$ rodadas de comunicação.

É fácil verificar na Figura 3.1 que o processador P_p só inicia seu trabalho quando o processador P_1 termina sua computação, na rodada $p-1$. Portanto, temos um balanceamento de carga muito ruim.

Visando uma melhor distribuição de carga, tentamos fazer com que cada processador inicie seu processamento o quanto antes. Isto pode ser feito diminuindo o tamanho da mensagem que o o processador P_i envia para o processador P_{i+1}. Em vez de considerarmos mensagens de tamanho $\frac{W}{p}$, consideramos mensagens de $\alpha\frac{W}{p}$ e testamos diversos tamanhos de α.

O Algoritmo 2 trabalha da seguinte maneira: depois de calcular f_i^k, o processador P_i envia R_i^k para o processador P_{i+1}. O processador P_{i+1} recebe R_i^k de P_i e calcula f_{i+1}^{k+1}. Após $p-2$ rodadas de comunicação, o processador P_p recebe R_{p-1}^{p-2} e calcula f_p^{p-1}. Se utilizarmos $\alpha < 1$ todos os processadores estarão trabalhando simultaneamente depois da $(p-2)$-ésima rodada. Testamos diferentes valores para α a fim de encontrar um bom equilíbrio entre o trabalho de cada processador e o número de rodadas do algoritmo. A Figura 3.2 mostra como o algoritmo funciona quando $\alpha = 1/2$.

Algoritmo 2 α-0-1-KNAPSACK

Entrada: (1) A quantidade p de processadores; (2) O identificador i de cada processador, onde $1 \leq i \leq p$; e (3) A capacidade W da mochila e os subvetores v_i e w_i de tamanho $\frac{n}{p}$; (4) A constante α.

Saída: $f(r,c) = \max\{f[r, c - w[r]] + v[r], f[r-1, c]\}$, onde $1 \leq c \leq W$ e $(j-1)\frac{n}{p} + 1 \leq r \leq j\frac{n}{p}$.

1: **for** $1 \leq k \leq \frac{p}{\alpha}$ **do**
2: **if** $i = 1$ **then**
3: **for** $\alpha(k-1)\frac{W}{p} + 1 \leq r \leq \alpha k \frac{W}{p}$ **and** $1 \leq c \leq \frac{n}{p}$ **do**
4: `compute` $f(r,c)$;
5: **end for**
6: `send`$(R_i^k,\ P_{i+1})$;
7: **end if**
8: **if** $i \neq 1$ **then**
9: `receive`(R_{i-1}^k, P_{i-1});
10: **for** $\alpha(k-1)\frac{W}{p} + 1 \leq r \leq \alpha k \frac{W}{p}$ **and** $1 \leq c \leq \frac{n}{p}$ **do**
11: `compute` $f(r,c)$;
12: **end for**
13: **if** $i \neq p$ **then**
14: `send`(R_i^k, P_{i+1});
15: **end if**
16: **end if**
17: **end for**

Utilizando o esquema da Figura 3.2, podemos notar que na primeira rodada, apenas o processador P_1 trabalha. Na segunda rodada, os processadores P_1 e P_2 trabalham. Na rodada k, todos os processadores P_i trabalham, onde $1 \leq i \leq k$. Dependendo do valor de $\alpha < 1$ os processadores iniciam seu trabalho mais cedo.

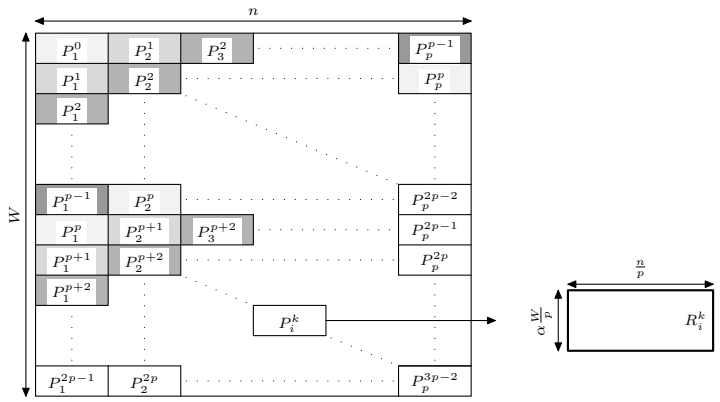


Figura 3.2: Um escalonamento de $O(p)$ rodadas de comunicação e $\alpha = 1/2$.

Teorema 3.1 *O Algoritmo 2 utiliza* $O(\frac{Wn}{p})$ *computação local com* $(1 + \frac{1}{\alpha})p - 2$ *rodadas de comunicação.*

Prova. O processador P_1 envia R_1^k para o processador P_2 após computar o k-ésimo bloco $\alpha\frac{W}{p}$ de linhas da $\frac{Wn}{p}$ submatriz f_1. Após $\frac{p}{\alpha}-1$ rodadas de comunicação, o processador P_1 termina seu trabalho. Similarmente, o processador P_2 acaba seu trabalho após $\frac{p}{\alpha}$ rodadas de comunicação. Assim, depois de $\frac{p}{\alpha}-2+i$ rodadas de comunicação, o processador P_i finaliza seu trabalho. Visto que temos p processadores, após $(1+\frac{1}{\alpha})p-2$ rodadas de comunicação, todos os p processadores terão acabado seu trabalho.

Cada processador utiliza um algoritmo de programação dinâmica sequencial para computar a solução ótima da submatriz f_i para o Problema da Mochila 0-1. Consequentemente este algoritmo utiliza tempo $O(\frac{Wn}{p})$ de computação local em cada processador p. ■

Teorema 3.2 *No final do Algoritmo 2, $f(n,W)$ armazenará a solução ótima para o Problema da Mochila 0-1 com n itens, valores v_i e pesos w_i, $1 \leq i \leq n$ e capacidade W.*

Prova. O Teorema 3.1 prova que após $(1+\frac{1}{\alpha})p-2$ rodadas de comunicação, o processador P_p termina seu trabalho. Essencialmente, estamos computando um algoritmo de programação dinâmica sequencial para o Problema da Mochila 0-1 e enviando as fronteiras para o processador da direita, a corretude do algoritmo aparece naturalmente com a corretude do algoritmo sequencial. Portanto, após $(1+\frac{1}{\alpha})p-2$ rodadas de comunicação, $f(n,W)$ armazenará a solução ótima para o Problema da Mochila 0-1 com n itens, valores v_i e pesos w_i, $1 \leq i \leq n$, capacidade W. ■

3.4 Resultados Experimentais

Nesta seção, tratamos dos resultados obtidos com o algoritmo BSP/CGM para o Problema da Mochila 0-1 descrito anteriormente.

Os dados de entrada usados em nossos testes foram gerados de forma aleatória. O programa recebe como entrada o número de itens n que serão gerados e capacidade da mochila W. De maneira aleatória o programa gera para cada item i um valor v_i e um peso w_i, ambos inteiros. Depois de gerada a lista com os valores e pesos dos n itens, o programa armazena em um arquivo, o número de itens, a capacidade da mochila e a lista dos itens com o peso e o valor de cada item.

A Tabela 3.1 mostra o número de itens e a capacidade da mochila para cada teste utilizado nos experimentos.

Instância	n	W
I_1	1024	4096
I_2	2048	8192
I_3	4096	16384
I_4	8192	32768

Tabela 3.1: Número de itens e a capacidade da mochila.

Na Tabela 3.2, podemos avaliar o desempenho do algoritmo no *Beowulf*. Nas instâncias I_1 e I_2, observamos que quando aumentamos o número de processadores até 4, conseguimos uma melhora no tempo de execução, mas quando utilizamos 8 processadores a comunicação acaba sendo maior que a computação local, aumentando o tempo de execução. Nas instâncias I_3 e I_4, os tempos do algoritmo também decrescem à medida que aumentamos o número de processadores.

P	I_1	I_2	I_3	I_4
1	0.068719	0.276378	1.104469	4.415288
2	0.064763	0.211221	0.958066	3.831723
4	0.031791	0.125380	0.581249	2.319630
8	0.036714	0.142538	0.578370	1.891723

Tabela 3.2: Tempo em segundos para o Problema da Mochila 0-1 utilizando o *Beowulf*.

A Tabela 3.3, apresenta o tempo execução do algoritmo no InteGrade com MPI. À medida que aumentamos o número de processadores, conseguimos uma melhora exceto na instância I_1, quando temos 8 processadores a comunicação acaba sendo maior que a computação local e o tempo de execução aumenta. Nas demais instâncias sempre há uma melhora no tempo a medida que aumentamos o número de processadores.

P	I_1	I_2	I_3	I_4
1	0.070851	0.307094	1.124227	4.425235
2	0.066116	0.265557	0.997288	3.885723
4	0.063857	0.244886	0.766450	2.368154
8	0.094035	0.243241	0.679454	2.009194

Tabela 3.3: Tempo em segundos para o Problema da Mochila 0-1 utilizando InteGrade.

Apesar do *overhead* gerado pelos módulos do InteGrade, os tempos de execuções ficaram próximos aos tempos gastos na execução do algoritmo no *Beowulf*. Ou seja, em aplicações onde há pouca comunicação entre os processadores o InteGrade apresenta resultados positivos.

A Figura 3.3 apresenta o gráfico gerado utilizando os resultados contidos nas Tabelas 3.2 e 3.3.

A fim de melhorar o balanceamento de carga do algoritmo, testamos diferentes valores de α, $1/2 \leq \alpha \leq 1/32$. As Tabelas 3.5 e 3.4 mostram o desempenho no *Beowulf* e no InteGrade, respectivamente. Nesse experimento utilizamos a instância I_4. Nos dois, casos podemos observar a melhora no tempo de execução do algoritmo à medida que aumentamos o número de processadores e diminuímos o valor de α. O melhor resultado foi obtido quando utilizamos $p = 8$ e $\alpha = 1/16$. Quando utilizamos $\alpha = 1/32$, o tempo de execução aumenta independente da quantidade de processadores. Isto ocorre porque, embora o processador inicie seu trabalho mais cedo, a quantidade de rodadas de comunicação aumenta, aumentando o tempo de execução do algoritmo.

Mesmo variando o valor de α, o comportamento do algoritmo no InteGrade permanece

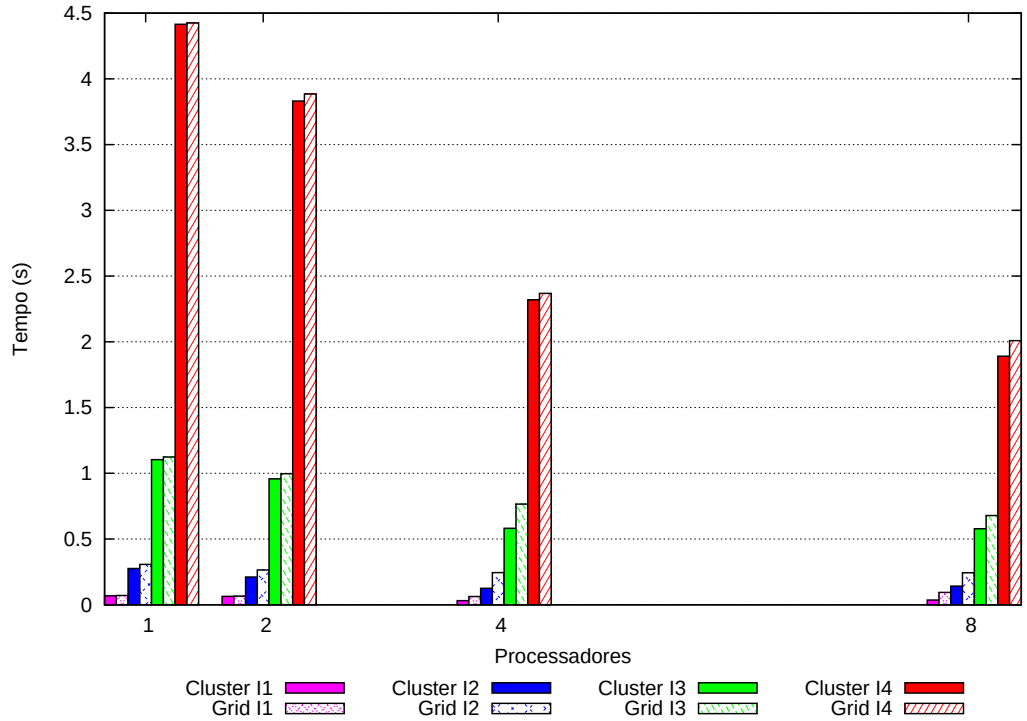


Figura 3.3: Gráfico de comparação entre o *Beowulf* e o InteGrade.

P	$\alpha = 1/2$	$\alpha = 1/4$	$\alpha = 1/8$	$\alpha = 1/16$	$\alpha = 1/32$
2	3.000317	2.272610	2.132903	2.082754	2.092594
4	1.846830	1.616668	1.520640	1.500600	1.552780
8	1.341998	1.137590	1.090617	1.030019	1.256126

Tabela 3.4: Tempo de execução para diferentes valores de α no *Beowulf*.

o mesmo, ou seja, o tempo se mantém próximo aos resultados obtidos no *Beowulf* apesar do aumento na quantidade de rodadas de comunicação.

P	$\alpha = 1/2$	$\alpha = 1/4$	$\alpha = 1/8$	$\alpha = 1/16$	$\alpha = 1/32$
2	3.284433	2.858701	2.792207	2.674303	2.438954
4	1.903860	1.717341	1.689433	1.588095	1.721421
8	1.541998	1.237598	1.119017	1.032134	1.567617

Tabela 3.5: Tempo de execução para diferentes valores de α no InteGrade.

Capítulo 4

Árvore Geradora

Este capítulo trata do Problema da Árvore Geradora. Na Seção 4.1 definiremos o problema. Na seção seguinte, discutiremos diferentes algoritmos paralelos existentes, apresentando o algoritmo de Cáceres *et al* [8] no modelo BSP/CGM que utiliza $O(\log\ p)$ rodadas de comunicação. Por fim, na Seção 4.3, mostraremos os resultados obtidos com a implementação do algoritmo.

4.1 O Problema

Seja $G = (V, E)$ um grafo com $n = |V|$ vértices e $m = |E|$ arestas. Uma árvore geradora $T = (V', E')$, é um subgrafo de G que é uma árvore e contém todos os vértices de G, ou seja, $V' = V$ e $E' \subset E$.

Na Figura 4.1, podemos observar, em (a), o grafo G e, em (b), sua árvore geradora T.

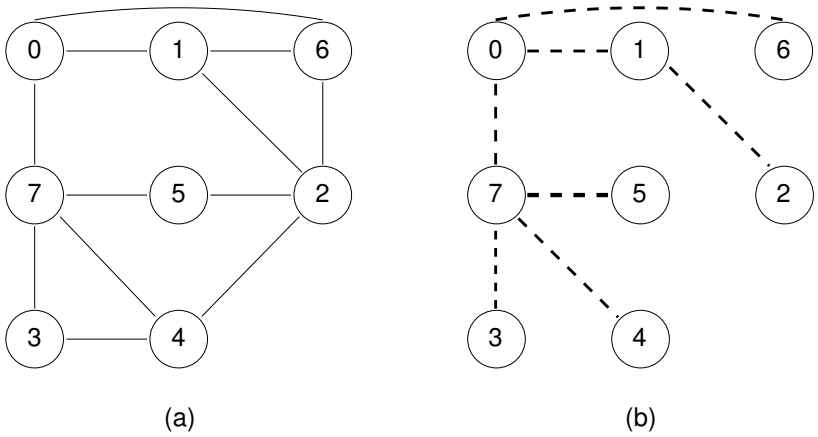


Figura 4.1: O grafo G e sua árvore geradora T.

O Problema da Árvore Geradora de um grafo é um problema básico em teoria dos grafos e serve como base de outros subproblemas para muitas aplicações. Sequencialmente, este problema é resolvido eficientemente aplicando-se uma busca em profundidade ou uma

busca em largura. O algortimo de busca em profundidade não possui uma implementação paralela eficiente [43].

4.2 Algoritmo Paralelo

Hirschberg *et al* [31] apresentaram no modelo PRAM CRCW um algoritmo que calcula os componentes conexos de um grafo com n vértices em tempo $O(\log^2 n)$ e utilizando n^2 processadores. Utilizando o mesmo modelo, Shiloach e Vishkin [45] apresentaram um algoritmo que gasta tempo $O(\log\ n)$ e utiliza $m + n$ processadores.

Melhorando trabalhos já apresentados, Halperin e Zwick [30] mostraram um algoritmo aleatório PRAM EREW que encontra componentes conexos do grafo em tempo $O(\log\ n)$ e utilizando $O((m + n)/\log\ n)$ processadores.

Bader e Cong [7] implementaram os algoritmos de Shiloach e Vishkin [45] e Hirschberg *et al* [31] e desenvolveram um novo algoritmo randômico para computar a árvore geradora em multiprocessadores simétricos.

Baseado no algoritmo PRAM proposto por Shiloach e Vishkin [45], Dehne *et al* [22] propuseram um algoritmo BSP/CGM que computa uma árvore geradora e os componentes conexos do grafo gastando $O(\log\ p)$ rodadas de comunicação e $O(\frac{m+n}{p})$ computação local em cada rodada, onde p é o número de processadores, esse algoritmo utiliza a solução do *Euler tour* no grafo de entrada, que é baseada na solução do problema do *list ranking*.

Cáceres *et al* [8] apresentaram um outro algoritmo paralelo no modelo BSP/CGM para calcular a árvore geradora. Este algoritmo também gasta $O(\log\ p)$ rodadas de comunicação e $O(\frac{n+m}{p})$ de computação local por rodada, mas não depende da solução do *Euler tour*, nem do *list ranking*. Ao invés disso, ele utiliza uma ordenação de número inteiros, o que pode ser implementado eficientemente no modelo BSP/CGM por [13].

4.2.1 Algoritmo de Cáceres *et al*

O algoritmo proposto por Cáceres *et al* [8] encontra uma árvore geradora em um grafo bipartido. Apesar da entrada do algoritmo ser um grafo bipartido, os autores mostram que o algoritmo funciona para qualquer grafo, pois todo grafo pode ser transformado em um grafo bipartido.

Antes de descrevermos o algoritmo, defininiremos algumas estruturas que utilizaremos no algoritmo. Considere um grafo bipartido $H = (V_1, V_2, E)$, sendo $V_1 = \{u_1, u_2, \ldots, u_{n_1}\}$ e $V_2 = \{v_1, v_2, \ldots, v_{n_2}\}$ o conjunto de vértices e E seu conjunto de arestas.

Um *strut* ST em V_1 é uma floresta geradora de H tal que cada $v_i \in V_2$ incide em ST com exatamente uma aresta de E e (u_j, v_i) é uma aresta de ST se e somente se (u_k, v_i) não é uma aresta de H, para qualquer $v_k \in V_1, k < j$.

Um vértice $u \in V_1$ é chamado de zero-diferença se o grau de u em H é igual ao grau de u em ST.

O Algoritmo 3 descreve os passos do algoritmo de Cáceres *et al* [8]. Dado um grafo bipartido $H = (V_1, V_2, E)$ com $|V_1| = n_1$, $|V_2| = n_2$ e $|E| = m$, a entrada do algoritmo são as arestas ordenadas lexicograficamente e então distribuídas entre os p processadores.

O primeiro passo do algoritmo é obter uma floresta geradora através de um *strut* ST em H. Em seguida, calcula-se a quantidade de vértices zero-diferença. Se o número de vértices zero-diferença for um, então o problema é facilmente resolvido adicionando-se uma aresta qualquer de $H - ST$ incidente em cada vértice não zero-diferença de ST.

Caso existam dois ou mais vértices zero-diferença é necessário fazer uma compactação no grafo. Para cada vértice zero-diferença $u \in V_1$ compacte todos os vértices $v_i \in V_2$ incidente em u juntando todos os vértices v_i no menor vértice de v_i. Isto é feito até que exista apenas um vértice zero-diferença.

Algoritmo 3 Árvore Geradora

Entrada: Um grafo bipartido $H = (V_1, V_2, E)$ onde $V_1 = \{u_1, \ldots, u_{n_1}\}$, $V_2 = \{v_1, \ldots, v_{n_2}\}$ e $\mid E \mid = m$. Uma aresta(u_i, v_i) de E possui o vértice u_i em V_1 e o vértice v_i em V_2. As m arestas estão igualmente distribuídas através dos p processadores.
Saída: Uma árvore geradora de H.

1: `Fase I`
2: Inicialize $\overline{V_1} \leftarrow V_1; \overline{V_2} \leftarrow V_2; \overline{E} \leftarrow E$;
3: **for** $\log p$ vezes **do**
4: **for** cada $v_i \in V_2$ **do**
5: Escolha o menor vértice u_j dentre todas as arestas (u, v_i) e marque a aresta (u_j, v_i). Seja ST o conjunto de arestas marcadas;
6: **end for**
7: Calcule o grau de cada vértice $u \in \overline{V_1}$ em $H(\overline{V_1}, \overline{V_2}, \overline{E})$;
8: Calcule o grau de cada vértice $u \in \overline{V_1}$ em H_{ST} $(\overline{V_1}, \overline{V_2}, \overline{E})$;
9: Utilizando os graus dos vértices computados anteriormente, conte o número de vértices zero-diferença;
10: **if** o número de vértices zero-diferença $= 1$ **then**
11: O algoritmo acaba;
12: **end if**
13: Compacte o grafo produzindo o grafo $H(\overline{V_1}, \overline{V_2}, \overline{E})$;
14: **end for**
15: `Fase II`
16: Compute uma floresta geradora com as arestas de $H(\overline{V_1}, \overline{V_2}, \overline{E})$ que não pertencem a ST e remova aquelas cujo `grau(`$\overline{u}$`)`$= 1$, onde $\overline{u} \in \overline{V_1}$;
17: **for** $i \leftarrow 0$ **to** $\log p$ **do**
18: P_{2i+1} envia sua floresta geradora para o o processador P_{2i};
19: O processador P_{2i} calcula uma nova floresta geradora;
20: **end for**

Teorema 4.1 *O Algoritmo 3 computa a árvore geradora de* $H = (V_1, V_2, E)$ *gastando* $O(\log p)$ *rodadas de comunicação e* $O(\frac{m+n}{p})$ *de computação local em cada rodada.*

Prova. Prova em [8]. ■

4.3 Resultados Experimentais

Nesta seção, mostramos os resultados obtidos com a implementação do algoritmo paralelo no modelo BSP/CGM para computar a árvore geradora previamente descrito. Durante a implementação, por simplicidade, algumas rotinas foram executadas sequencialmente.

Para os testes foram gerados grafos bipartidos aleatórios. A Tabela 4.1 mostra o número de vértices e de arestas para cada grafo utilizado como entrada do algoritmo. Como o número de arestas é estritamente maior que o número de vértices, assumimos que o grafo possui o mesmo número de vértices nas duas partições.

Instância	V	E
G_1	1024	262.144
G_2	2048	1.048.576
G_3	4096	4.194.304
G_4	8192	8.388.608

Tabela 4.1: Quantidade de vértices e arestas do grafo bipartido.

Na Tabela 4.2, observamos o tempo de execução do algoritmo no *Beowulf* para diversos tamanhos de grafos. Na instância G_1, como o número de arestas é pequeno, a execução com um único processador é mais rápida. Quando aumentamos de 2 para 4 processadores conseguimos uma pequena melhora, mas com 8 processadores a comunicação acaba sendo maior que a computação local e os tempos de execução aumentam. Com esta instância, fica difícil analisarmos o comportamento do algoritmo pois temos um número pequeno de arestas. Na instância G_2 também podemos observar que a execução em único processador é mais eficiente, mas a partir de $p = 2$, sempre que aumentamos o número de processadores diminuímos o tempo de execução do algoritmo. Nas instâncias G_3 e G_4, os tempos do algoritmo paralelo decrescem à medida que aumentamos o número de processadores.

P	G_1	G_2	G_3	G_4
1	0.217107	0.950689	5.240836	11.789449
2	0.235769	1.161584	4.836589	10.615158
4	0.230011	1.133226	4.350016	9.687900
8	0.240889	1.117800	3.360593	8.812110

Tabela 4.2: Tempo em segundos para o Problema da Árvore Geradora utilizando o *Beowulf*.

A Tabela 4.3 mostra a execução do algoritmo no InteGrade. Aqui o comportamento do algoritmo é contrário ao que ocorre no *Beowulf*. Na grade, à medida que aumentamos o número de processadores, aumentamos também o tempo de execução do algoritmo, um comportamento que não é esperado quando trabalhamos com algoritmos paralelos.

Utilizando apenas 1 processador, o tempo de execução com o InteGrade é similar ao tempo no *Beowulf*, sendo ligeiramente melhor para as maiores instâncias. Quando o número de processadores aumenta e inicia-se a troca de dados entre os processadores há um aumento considerável no tempo de processamento.

P	G_1	G_2	G_3	G_4
1	0.251695	1.099503	5.090014	11.643704
2	1.677849	5.432931	18.379176	29.222423
4	2.361967	8.001241	25.201421	37.953199
8	2.600342	10.314014	29.228178	43.632104

Tabela 4.3: Tempo em segundos para o Problema da Árvore Geradora utilizando *middleware* InteGrade e MPI.

No *Beowulf*, observamos um ganho pouco significativo no tempo de processamento conforme aumentamos o número de processadores. Isso ocorre porque dificilmente é necessário utilizar a `Fase II` do algoritmo. Além disso, a compactação do grafo na `Fase I` exige a troca de dados entre todos os processadores, pois é necessário redistribuir as arestas compactadas para a próxima iteração do algoritmo. A quantidade e o tamanho das mensagens trocadas entre os processadores contribui negativamente no desempenho do algoritmo no InteGrade.

A Figura 4.2 apresenta o gráfico gerado utilizando os resultados contidos nas Tabelas 4.2 e 4.3.

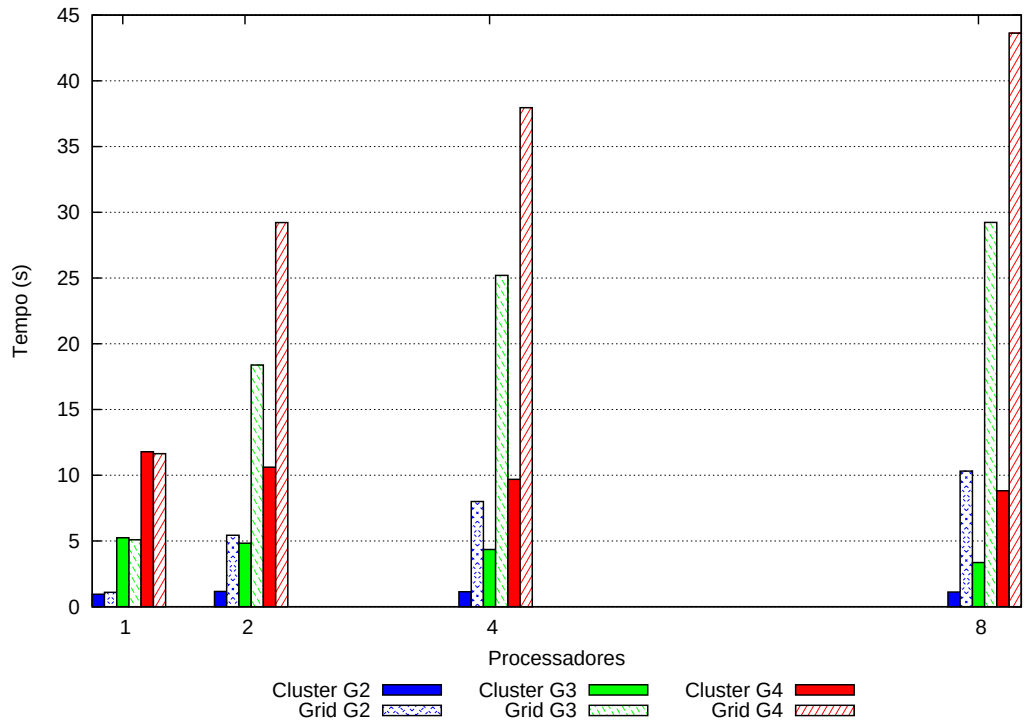


Figura 4.2: Gráfico de comparação dos resultados obtidos pelo *Beowulf* e pelo InteGrade.

Capítulo 5

Fecho Transitivo

Neste capítulo, discutiremos o Problema do Fecho Transitivo. Primeiro, defineremos o problema e apresentaremos dois algoritmos sequenciais que são usados como base dos algoritmos parelos apresentados neste capítulo. Os algoritmos paralelos possuem diferentes classes de rodadas de comunicação. Assumindo que p seja a quantidade de processadores e n o tamanho da entrada de dados, o algoritmo apresentado por Alves *et al* [3] possui $O(p)$ rodadas de comunicação, por outro lado, Vieira [50] apresentou um algoritmo que não possui nenhuma rodada de comunicação e, por fim, um algoritmo com $O(n)$ rodadas de comunicação, proposto por Jenq e Sahni [33]. Na Seção 5.4 descreveremos os resultados obtidos com a implementação de cada algoritmo paralelo.

5.1 O Problema

Em muitas aplicações, desejamos determinar se dois vértices em um grafo estão conectados. Isto, normalmente, é feito encontrando o fecho transitivo do grafo.

Seja $D = (V, E)$ um digrafo, onde $|V| = n$ é o conjunto de vértices e $|E| = m$ é o conjunto de arestas. O **fecho transitivo** de D é definido como o digrafo $D^t = (V, E^t)$, tal que, para todos os pares de vértices (v_i, v_j) de V, existe uma aresta (i, j) em E^t se e somente se existe um caminho de i até j. A Figura 5.1 ilustra, em (a), o digrafo D e, em (b), seu fecho transitivo D^t.

O fecho transitivo de um digrafo é um importante subproblema de diversas aplicações computacionais em redes de computadores, sistemas paralelos e distribuídos, banco de dados e no projeto de compiladores [40].

5.2 Algoritmos sequenciais

Muitos algoritmos foram propostos para solucionar o problema do fecho transitivo de forma eficiente. Em 1959, Roy [44] estudou a computação do fecho transitivo em digrafos.

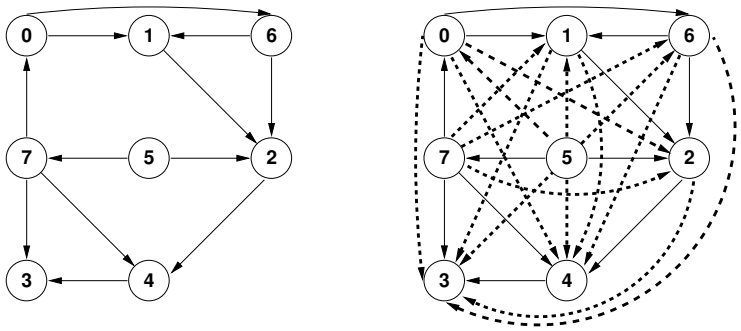


Figura 5.1: Digrafo D e o fecho transitivo D^t.

Warshall [51] apresentou um algoritmo com complexidade $O(n^3)$ que utilizava uma matriz de adjacências para armazenar o digrafo. Através de uma matriz de adjacência de bits para representar o digrafo, Baase [6] descreveu uma variação do algoritmo de Warshall com complexidade de $O(\frac{n^3}{\alpha})$, onde α é quantidade de bits que podem ser armazenados em um tipo de dado primitivo.

Uma busca em largura ou em profundidade encontra todos os vértices que são alcançáveis a partir de um dado vértice v. Portanto, aplicando uma busca para cada um dos vértices do digrafo, computamos o fecho transitivo. Cada busca pode ser realizada com complexidade de tempo $O(n + m)$. Sendo assim, o fecho transitivo pode ser computado em $O(n(n + m))$.

Para grafos esparsos, o melhor algoritmo sequencial para computar o fecho transitivo gasta $O(nm)$ e foi proposto por Koubková e Koubek [37], Habib *et al* [29] e Simon [46]. Para computar o fecho transitivo em grafos densos, Coppersmith e Winograd [17] basearam seu algoritmo em multiplicação de matrizes, gastando $O(n^{2.376})$. No entanto, este algoritmo não apresenta bons resultados na prática, devido a algumas constantes agregadas à sua complexidade.

Embora possua complexidade $O(n^3)$, o algoritmo de Warshall pode ser facilmente implementado e possui uma característica que favorece sua paralelização, o que pode ser verificado nos trabalhos de Alves *et al* [3], Castro Jr. [34] e Vieira [50].

5.2.1 Algoritmo de Warshall

Seja um digrafo $D = (V, E)$, representado pela matriz de adjacências M, onde cada posição $M_{i,j}$ representa uma aresta (i, j). Se $M_{i,j} = 1$, então a aresta existe no digrafo, caso contrário $M_{i,j} = 0$.

Dado dois vértices v_i e v_j, o algoritmo de Warshall verifica se existe um caminho entre eles. Este caminho é construído de maneira gradativa com a inclusão de todos os vértices entre v_i e v_j.

O Algoritmo 4 apresenta os passos do algoritmo de Warshall. A entrada é o digrafo D

representado pela matriz de adjacências M e o número de vértices n do digrafo. A saída será a matriz de adjacências M^t representando o fecho transitivo do digrafo D.

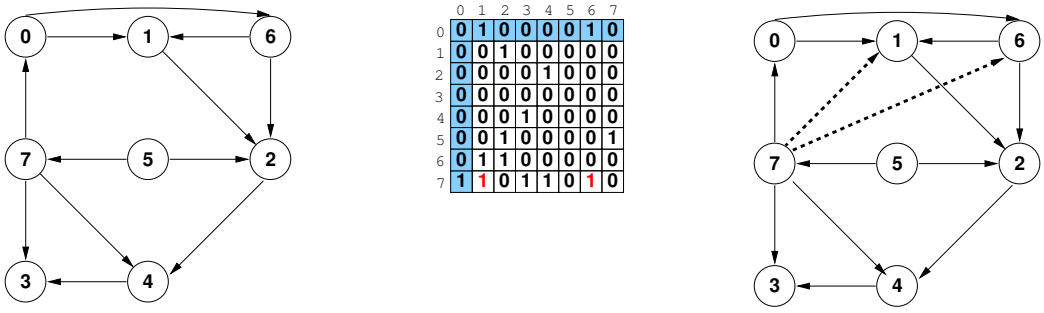
Algoritmo 4 Algoritmo de Warshall

Entrada: (1) A quantidade n de vértices; (2) A matriz de adjacências M do digrafo.
Saída: O Fecho Transitivo representado pela matriz de adjacências M^t.

```
1: M^t ← M;
2: for 0 ≤ k < n do
3:    for 0 ≤ i < n do
4:       for 0 ≤ j < n do
5:          if M^t_{i,k} ∧ M^t_{k,j} then
6:             M^t_{i,j} ← 1;
7:          end if
8:       end for
9:    end for
10: end for
```

No primeiro estágio, verifica-se a existência de um caminho entre v_i e v_j passando pelo vértice v_0. No segundo estágio, verifica-se a existência de um caminho entre v_i e v_j passando pelo vértice v_1. E assim sucessivamente até o n-ésimo estágio, onde todos os caminhos de v_i para v_j, passando pelos n vértices, foram verificados.

A estrutura do algoritmo permite que, durante o k-ésimo estágio, todas as posições da matriz de adjacências possam ser utilizadas na geração de novas arestas, entretanto apenas a k-ésima linha e a k-ésima coluna são utilizadas na geração das arestas. Tal característica pode ser observada na Figura 5.2, onde para gerar as duas arestas, $(7, 1)$ e $(7, 6)$, foram utilizados apenas a coluna zero e a linha zero.



(a) (b) (c)

Figura 5.2: Primeira iteração do Algoritmo 4.

5.2.2 Algoritmo de Busca

O Algoritmo 4 utiliza uma matriz de adjacência para representar o digrafo e computar o fecho transitivo de maneira gradativa. Uma outra estratégia para computar o fecho

transitivo é verificar quais vértices são alcançados a partir de um determinado vértice. Utilizando uma busca no digrafo, é possível determinar todos os vértices v_j que um vértice v_i alcança, $0 \leq i, j < n$.

A busca em largura ou em profundidade no digrafo, a partir de um determinado vértice, pode ser realizada gastando-se $O(n + m)$ [18]. Como é necessário realizar uma busca a partir de cada vértice do digrafo, a complexidade para se calcular o fecho transitivo, utilizando buscas, será de $O(n(n + m))$.

O Algoritmo 5 descreve os passos para computar o fecho transitivo de um digrafo D utilizando uma busca em profundidade. A entrada do Algoritmo 5 é uma lista de adjacências L que representa o digrafo D. A cada iteração, o algoritmo determina quais vértices são alcançados a partir do vértice v_i e armazena na árvore T^{v_i}, em seguida estes vértices são adicionados a lista L^t que representa o fecho transitivo.

Algoritmo 5 FT com Busca

Entrada: (1) A quantidade n de vértices; (2) A lista de adjacências L do digrafo.
Saída: O fecho transitivo do digrafo D.

1: **for** $i \leftarrow 0$ **to** $n - 1$ **do**
2: Compute T^{v_i};
3: Adicione ao fecho transitivo L^t os vértices de T^{v_i};
4: **end for**

A Figura 5.3 ilustra a primeira iteração do Algoritmo 5. O digrafo representado em (a) é o digrafo inicial do algoritmo. A Figura 5.3b mostra a árvore, T^{v_0}, construída a partir da busca em profundidade iniciada no vértice v_0. E o digrafo representado em (c) é o fecho transitivo parcial após a computação da primeira iteração do algoritmo, as setas em vermelho são as novas arestas criadas a partir da busca iniciada em v_0.

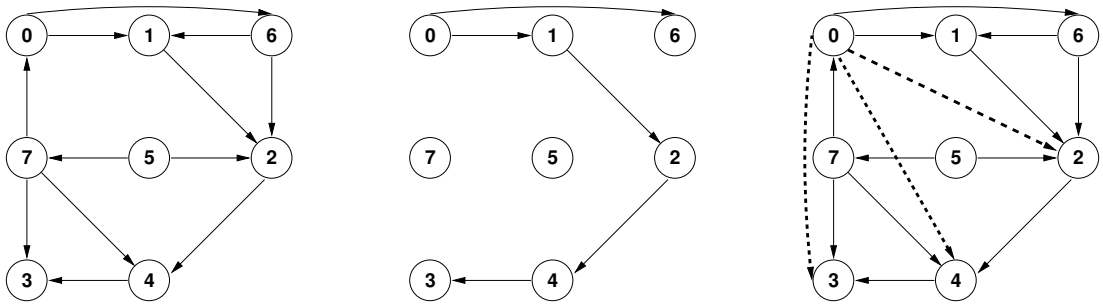


Figura 5.3: Primeira iteração do Algoritmo 5.

5.3 Algoritmos Paralelos

Utilizando o modelo BSP/CGM, Cáceres *et al* [19] apresentaram um algoritmo para computar o fecho transitivo de um digrafo acíclico usando $log\ p + 1$ rodadas de comunicação,

onde p é o número de processadores, $O(\frac{nm}{p})$ em computação local e $O(\frac{nm}{p})$ de espaço em cada processador, sendo n o número de vértices e m a quantidade de arestas.

Baseado no algoritmo sequencial de Warshall [51] e no algoritmo de Cáceres *et al* [19], Alves *et al* [3] e Castro Jr. [34] apresentaram uma nova solução no modelo BSP/CGM para computar o fecho transitivo. Este algoritmo distribui os dados de acordo com a numeração inicial dos vértices do digrafo G e não computa a extensão linear.

Vieira [50] avaliou as implementações de Cáceres *et al* [19], Alves *et al* [3] e Castro Jr. [34], verificando os tempos relacionados à computação local e à troca de mensagens; melhorando o desempenho dessas implementações e diminuindo o tamanho das mensagens trocadas entre os processadores, a computação local e a quantidade de rodadas de comunicação entre os processadores.

Jenq e Sahni [33] apresentaram um algoritmo paralelo para calcular o caminho mínimo entre todos os pares de vértices de um grafo. A mesma idéia pode ser utilizada para se calcular o fecho transitivo. O algoritmo gasta $\theta(\frac{n^2}{\sqrt{p}}\log p)$ em comunicação e $\theta(\frac{n^3}{p})$ em processamento.

Pagourtzis *et al* [41] apresentaram um algoritmo de granularidade grossa desenvolvido a partir de um algoritmo de granularidade fina para se computar o fecho transitivo. O algoritmo foi implementado em PVM (*Parallel Virtual Machine*). Existem também outros trabalhos implementados em PVM [14, 42].

5.3.1 Algoritmo de Alves *et al* e Castro Jr.

Alves *et al* [3] e Castro Jr. [34] apresentaram uma nova abordagem utilizando o modelo BSP/CGM para computar o fecho transitivo. Este algoritmo divide a matriz de adjacências M entre p processadores. Cada processador recebe uma faixa horizontal com $\frac{n}{p}$ linhas e uma faixa vertical com $\frac{n}{p}$ colunas. Por simplicidade, mas sem perda de generalidade, assume-se que n é divisível por p. O processador P_i receberá a i-ésima faixa horizontal e a i-ésima faixa vertical, ou seja, $M[\frac{in}{p}..\frac{(i+1)n}{p}-1][0..n-1]$ e $M[0..n-1][\frac{in}{p}..\frac{(i+1)n}{p}-1]$. A Figura 5.4 mostra a divisão do digrafo da Figura 5.1 entre $p = 4$ procesadores.

Esta distribuição de dados permite a utilização do algoritmo sequencial de Warshall em cada processador. Em cada rodada, o algoritmo verifica se o vértice k pertence ao caminho de i até j. Para isso, é necessário apenas a k-ésima linha e a k-ésima coluna. Depois de calculado o fecho transitivo local, as arestas geradas nesta fase de computação que pertençam a um outro processador são enviadas na próxima rodada de comunicação. Em seguida, o processador que recebe a aresta, atualiza o fecho transitivo local que possui. O algoritmo acaba quando nenhuma nova aresta é gerada ou atualizada. O Algoritmo 6 especifica os passos do algoritmo.

O algoritmo apresentado por Alves *et al* [3] e Castro Jr. [34] não garante que o algoritmo gaste $O(\log p)$ rodadas de comunicação, pois, no pior caso, podem ser necessários $p-1$ rodadas. No entanto, os nossos experimentos e os testes realizados pelos autores para digrafos gerados aleatoriamente mostraram que apenas $\log p+1$ rodadas foram suficientes para se computar o fecho transitivo.

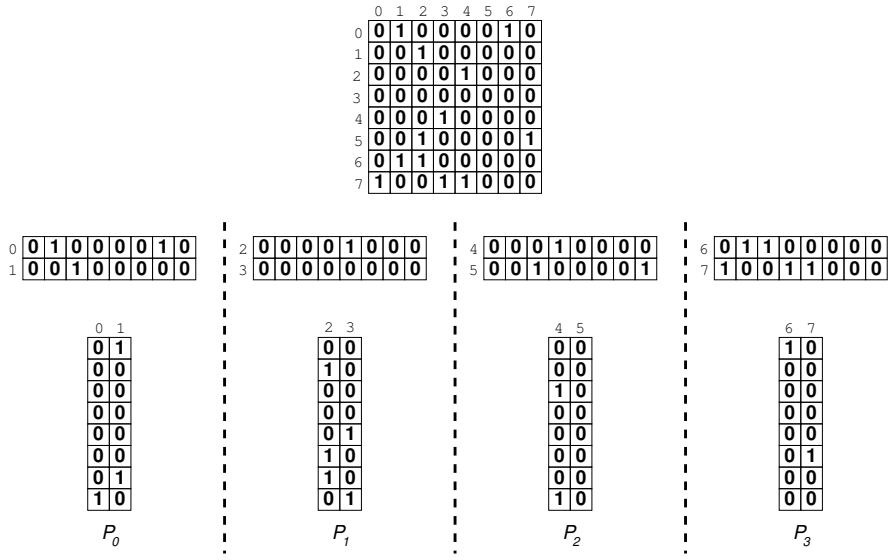


Figura 5.4: Distribuição de dados utilizado pelo Algoritmo 6.

Algoritmo 6 Algoritmo de Alves

Entrada: (1) A quantidade p de processadores; (2) O identificador q de cada processador, onde $0 \leq q < p$; (3) As submatrizes $M[(q-1)\frac{n}{p}..q\frac{n}{p}][0..n-1]$ e $M[0..n-1][(q-1)\frac{n}{p}..q\frac{n}{p}]$.

Saída: O Fecho Transitivo representado pela matriz de adjacências M distribuído pelos p processadores.

1: **repeat**
2: **for** $(q-1)\frac{n}{p} \leq k \leq q\frac{n}{p}$ **do**
3: **for** $0 \leq i \leq n-1$ **do**
4: **for** $0 \leq j \leq n-1$ **do**
5: **if** $Mi,k \wedge Mk,j$ **then**
6: $Mi,j \leftarrow 1$;
7: **end if**
8: **end for**
9: **end for**
10: **end for**
11: Envie as arestas pertencentes a outros processadores;
12: Receba as arestas e atualize seu fecho local;
13: **until** que nenhuma aresta seja criada

5.3.2 Algoritmo de Vieira

Vieira [50] observou nas implementações de Alves *et al* [3] e Castro Jr. [34] que a troca de mensagens comprometia o desempenho do algoritmo e apresentou um algoritmo BSP/CGM com complexidade $O(\frac{n}{p}(n+m))$ de computação local e sem rodada de comunicação. O algoritmo utiliza as ideias contidas no algoritmo sequencial de busca (Algoritmo 5).

O algoritmo proposto utiliza uma lista de adjacências para representar o grafo de entrada, que em seguida é distribuída inteiramente para todos os processadores. Assim, cada processador possui os dados necessários para computar o fecho transitivo. Vieira denota Φ_q como o conjunto de $\frac{n}{p}$ vértices que cada processador q utiliza como vértice origem. Através de uma busca em profundidade no digrafo, encontra-se todos os vértices j alcançáveis a partir de um vértice i, como $i \in \Phi_q$ cria-se a aresta (i, j). A complexidade de espaço é $O(n + m)$. O Algoritmo 7 apresenta os passos do algoritmo proposto por Vieira [50].

Durante a i-ésima iteração do Algoritmo 7, todos os p processadores armazenam em Γ^i todos os vértices que foram alcançados a partir de v_i, em seguida adicionam estes vértices ao fecho transitivo final. Depois de $\frac{n}{p}$ iterações, o fecho transitivo foi calculado e está distribuído pelos p processadores.

Algoritmo 7 Algoritmo de Vieira

Entrada: (1) A quantidade p de processadores; (2) O identificador q de cada processador, onde $0 \leq q \leq p - 1$; e (3) A lista de adjências.
Saída: O fecho transitivo armazenado na lista de adjacências L^t.

```
1: for i ← q n/p to (q + 1) n/p do
2:     Compute Γ^i;
3:     Adicione ao fecho transitivo L^t os vértices de Γ^i;
4: end for
```

O Algoritmo 7 não necessita de comunicação entre os processadores, pois as arestas geradas a partir de Γ^i não são utilizadas na computação por outros processadores, podendo provocar um desbalanceamento na computação local. Contudo, o algoritmo possui uma boa escalabilidade.

5.3.3 Algoritmo de Jenq e Sahni

Utilizando as ideias do algoritmo paralelo apresentado por Jenq e Sahni [33] para o problema de caminho mínimo entre todos os pares de vértices de um grafo, implementamos um algoritmo BSP/CGM com $O(n)$ rodadas de comunicação para o Problema do Fecho Transitivo. A computação local e a memória gasta dependem da distribuição de dados utilizada.

Os autores apresentaram duas formas para dividir uma matriz $M_{n\times n}$ entre p processadores:

- Faixa: a matriz é dividida em p faixas, criando p submatrizes com n linhas e $\frac{n}{p}$ colunas.
- Retângulo: a matriz é dividida em p blocos, cada submatriz de ordem $\frac{n}{2^{\lfloor d/2 \rfloor}} \times \frac{n}{2^{\lceil d/2 \rceil}}$, onde $p = 2^d$. Portanto, se d for par teremos quadrados, e cada submatriz terá $\frac{n}{\sqrt{p}}$ linhas e colunas.

Para facilitar a compreensão do algoritmo, assuma que os p processadores estão organizados logicamente como uma grade, ou seja, o processador $P_{i,j}$ pertence à linha i e à coluna j da grade. Isto é apenas uma representação visual e não reflete, necessariamente, o tipo de interconexão entre os processadores. Na distribuição por faixas, a grade possui 1 linha e p colunas. Na divisão por retângulo, assuma que a grade possui $\sqrt{p}$ linhas e $\sqrt{p}$ colunas.

A Figura 5.5 ilustra a distribuição da matriz em faixas e retângulos entre 4 processadores. Em (a), observamos a matriz que representa o grafo de entrada. Em (b), a distribuição por faixas dos dados e a organização dos processadores e, em (c), a distribuição por blocos e a disposição dos processadores na grade.

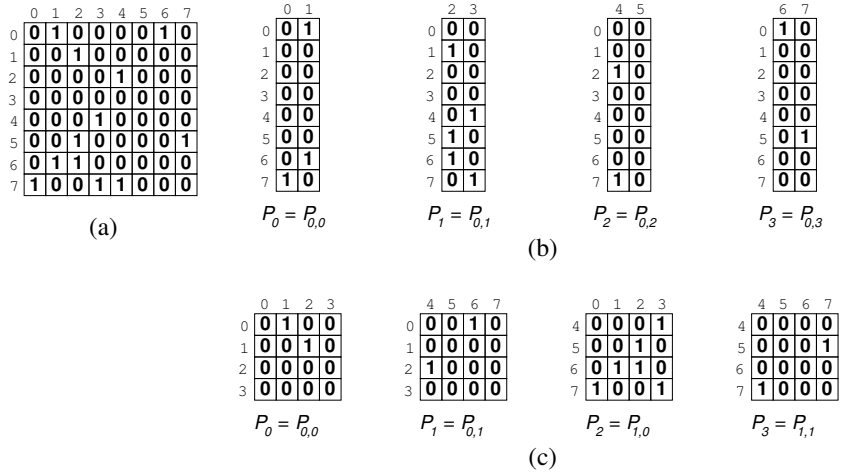


Figura 5.5: Distribuição de dados utilizado pelo Algoritmo 8.

Como mencionado anteriormente, para se calcular o fecho transitivo, utilizando o Algoritmo de Warshall, é necessário apenas a k-ésima linha e a k-ésima coluna para se calcular todos os caminhos de i até j, passando por k. Logo, cada processador $P_{i,j}$ necessita de uma faixa da k-ésima linha e da k-ésima coluna da matriz M. Se estivermos utilizando a distribuição por faixas, cada processador já possui o pedaço da k-ésima linha que necessita, no entanto, precisará receber de outro processador a k-ésima coluna. Mas, se a distribuição de dados utilizada for a retângulo, cada um dos $\sqrt{p}$ processadores que possui um pedaço da k-ésima linha envia este para os $\sqrt{p} - 1$ processadores na mesma coluna. Similarmente, cada um dos $\sqrt{p}$ processadores que possui um pedaço da k-ésima coluna envia este para os $\sqrt{p} - 1$ processadores na mesma linha.

O Algoritmo 8 é uma modificação no algoritmo apresentado por Jenq e Sahni [33] para que este compute o fecho transitivo de um digrafo. O algoritmo utiliza, como entrada de dados, uma matriz de adjacências M para representar o digrafo. Feita a escolha pelo método de distribuição de dados, o primeiro passo do algoritmo é determinar qual processador possui a k-ésima linha da matriz, para que este possa enviá-la a todos os processadores que necessitam da informação. Similarmente, descobre-se quem possui a k-ésima coluna. De posse de todos os dados necessários, cada processador $P_{i,j}$ calcula seu fecho local e armazena em sua submatriz. O processo se repete n vezes até que, no final, o fecho transitivo foi calculado e encontra-se distribuído entre os p processadores.

Algoritmo 8 Algoritmo de Jenq e Sahni Modificado

Entrada: (1) A quantidade p de processadores; (2) O identificador q de cada processador, onde $0 \leq q \leq p-1$; (3) A submatriz de acordo com a distribuição de dados utilizada.
Saída: O fecho transitivo distribuído entre os p processadores.

1: **for** $k \leftarrow 0$ **to** $n-1$ **do**
2: Cada processador $P_{i,j}$ que possui a k-ésima linha a envia para todo processador $P_{*,j}$;
3: Cada processador $P_{i,j}$ que possui da k-ésima coluna envia esta para todo processador $P_{i,*}$;
4: Cada processador recebe os dados que necessita;
5: Todo processador $P_{i,j}$ computa seu fecho local;
6: **end for**

5.4 Resultados Experimentais

Nesta seção, descrevemos os resultados obtidos a partir das implementações dos três algoritmos paralelos descritos anteriormente.

Para obtermos os resultados apresentados nesta seção, foram gerados digrafos acíclicos. A Tabela 5.1 apresenta a quantidade de vértices e de arestas de cada digrafo gerado.

Instância	V	E
D_1	1024	210000
D_2	2048	800000
D_3	4096	2000000
D_4	6144	4000000

Tabela 5.1: Número de vértices e arestas dos digrafos.

5.4.1 Resultados do Algoritmo de Alves

A Tabela 5.2 apresenta os resultados obtidos quando executamos o algoritmo de Alves (Algoritmo 6) no *Beowulf*. Nesta tabela, podemos observar que o tempo diminui, conforme aumentamos o número de processadores.

P	D_1	D_2	D_3	D_4
1	5.009932	39.482493	312.919694	1052.727073
2	2.950086	22.594351	177.560120	598.062114
4	1.749157	13.213336	104.150085	349.281700
8	1.211288	8.934399	69.405486	233.526661

Tabela 5.2: Tempos obtido pelo Algoritmo de Alves *et al* [3] no *Beowulf*.

A Tabela 5.3 mostra os resultados para a execução do Algoritmo de Alves *et al* [3] no InteGrade. Neste caso, há uma diferença nos resultados obtidos. No InteGrade, o

tempo decai apenas quando temos no máximo 4 processadores sendo utilizados, em todas os testes realizados sempre que 8 processadores são utilizados, o tempo aumenta. Para entradas pequenas, o tempo de processamento já aumenta para 4 processadores. Ou seja, quando passamos de 4 para 8 processadores, tempo gasto com processamento é superado pelo tempo gasto com comunicação.

P	D_1	D_2	D_3	D_4
1	5.308890	42.762307	350.070205	1065.645218
2	4.241723	33.282511	232.072663	736.942952
4	6.007912	29.553537	176.509916	518.176995
8	9.747264	45.267509	239.355367	665.504264

Tabela 5.3: Tempos obtido pelo Algoritmo de Alves *et al* [3] utilizando o InteGrade.

Na Figura 5.6, podemos observar os tempos obtidos pelo Algoritmo de Alves *et al* [3] na execução dos digrafos D_3 e D_4 das Tabelas 5.2 e 5.3.

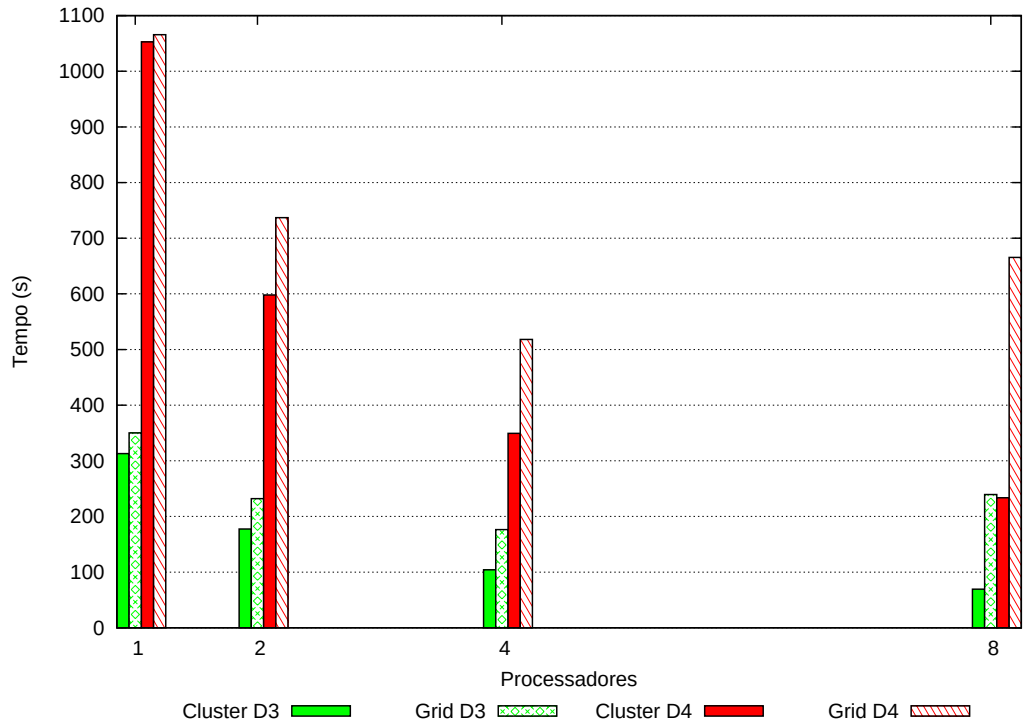


Figura 5.6: Gráfico de comparação entre o *Beowulf* e o InteGrade para o Algoritmo de Alves *et al* [3] para os digrafos D_3 e D_4.

5.4.2 Resultados do Algoritmo de Vieira

Os resultados obtidos com a implementação do Algoritmo 7 comprovam que, embora possa existir um desbalanceamento na computação local, a ausência de comunicação entre os processadores e a boa complexidade do algoritmo de busca produz bons resultados.

A Tabela 5.4 apresenta os resultados obtidos no *Beowulf* na execução do Algoritmo de Vieira [50].

P	D_1	D_2	D_3	D_4
1	0.801448	6.015528	29.216829	85.614550
2	0.719707	5.195078	25.418925	74.433189
4	0.472781	3.408277	16.640012	48.920923
8	0.271171	1.939994	9.469316	27.813640

Tabela 5.4: Resultados obtidos pelo Algoritmo de Vieira [50] usando o *Beowulf*.

A Tabela 5.5 mostra o tempo de execução do algoritmo no InteGrade. Assim como no *Beowulf*, o tempo de processamento diminui à medida que aumentamos o número de processadores, porém a redução é menor que no *Beowulf*. Esse menor desempenho pode estar relacionado com o desbalanceamento da computação local, além dos módulos do próprio InteGrade que fazem o monitoramento das aplicações.

P	D_1	D_2	D_3	D_4
1	0.802943	6.242818	30.639646	90.007240
2	0.757624	5.417648	26.592451	83.120324
4	0.515882	3.559516	25.921029	78.245178
8	0.286561	2.021061	23.213637	69.365300

Tabela 5.5: Resultados obtidos pelo Algoritmo de Vieira [50] usando o InteGrade.

Na Figura 5.7 podemos observar o gráfico obtido a partir das Tabelas 5.4 e 5.5 para os digrafos D_3 e D_4.

5.4.3 Resultados dos Algoritmos de Jenq

Por questões de infraestrutura, optamos em nossos testes utilizarmos a distribuição por faixas para avaliarmos melhor o comportamento do algoritmo à medida que aumentávamos o número de processadores. No entanto, antes de escolher a distribuição por faixas, realizamos testes com a distribuição por retângulo utilizando 1 e 4 processadores, e, em ambos, os tempos obtidos foram similares aos obtidos pelo uso da distribuição por faixas.

Na Tabela 5.6, podemos observar o tempo de execução do Algoritmo 8. Apesar de gastar n rodadas de comunicação, o algoritmo apresentou bons resultados utilizando até 4 processadores. O ganho mais significativo ocorreu quando passamos de 1 para 2 processadores. Mas, quando passamos de 4 para 8 processadores, o desempenho caiu para todas instâncias, ou seja, o tempo gasto em comunicação é maior que o tempo gasto em processamento local. Neste caso, a quantidade de processadores que devem receber a informação em cada uma das n rodadas interfere no desempenho do algoritmo.

A Tabela 5.7 mostra o desempenho do InteGrade ao executar o Algoritmo 8. Neste caso, a grande quantidade de rodadas de comunicação e o número de processadores que recebem informações em cada rodada de comunicação influenciaram negativamente no

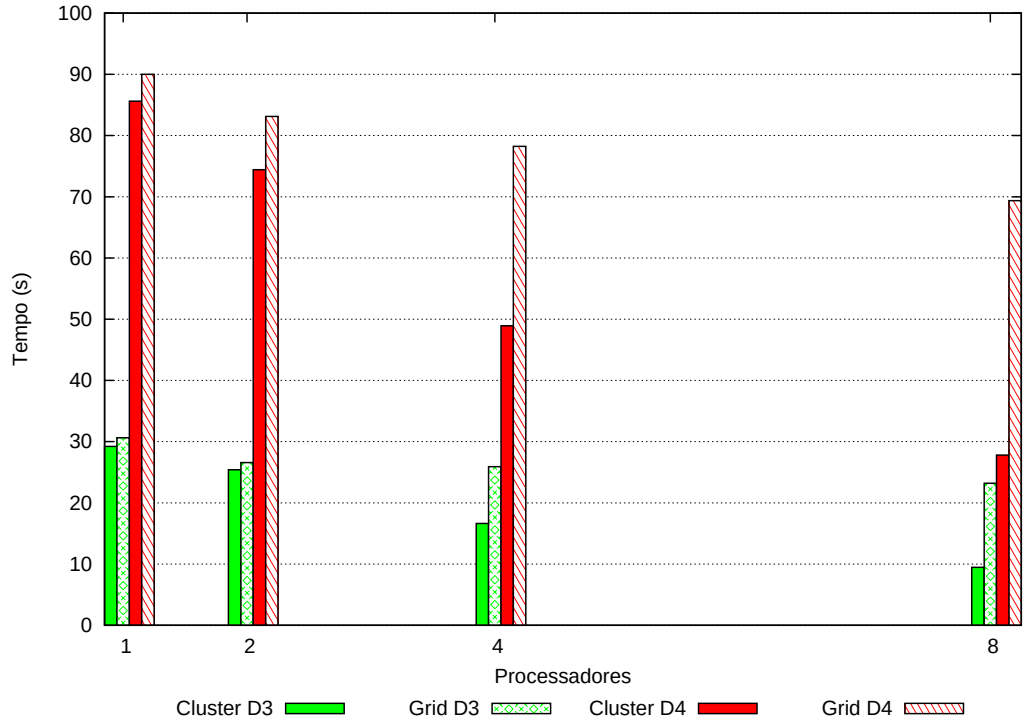


Figura 5.7: Gráfico de comparação entre o *Beowulf* e o InteGrade executando o Algoritmo de Vieira [50].

P	D_1	D_2	D_3	D_4
1	8.816681	70.101291	550.746042	1852.000289
2	2.310720	18.597281	122.838527	403.293859
4	1.221283	9.325313	65.335970	212.905034
8	1.245993	9.610402	71.526010	235.296830

Tabela 5.6: Tempos obtido pelo Algoritmo de Jenq e Sahni [33] utilizando o *Beowulf*.

desempenho do algoritmo no InteGrade. Na instância D_1, quando passamos de 2 para 4 e de 4 para 8 processadores, o tempo de processamento mais que duplicou. Já na instância D_2, o tempo de execução ficou 5 vezes maior quando passamos de 2 para 4 processadores mas reduziu quando passamos de 4 para 8 processadores. Nas maiores instâncias, D_3 e D_4, o tempo de execução só piorou quando passamos de 4 para 8 processadores. Outro ponto importante é que para 4 e 8 processadores, o tempo no InteGrade foi muito superior ao tempo no *Beowulf*.

P	D_1	D_2	D_3	D_4
1	8.877235	70.476558	554.721261	1862.199957
2	5.028641	35.526482	287.254588	745.924543
4	12.097772	187.730511	184.492489	514.539383
8	30.513495	110.352113	435.035362	981.427842

Tabela 5.7: Tempos obtido pelo Algoritmo de Jenq e Sahni [33] utilizando o InteGrade.

Na Figura 5.8, observamos o comportamento da nossa implementação para as os digrafos D_3 e D_4 tanto no *Beowulf* quanto no InteGrade. Os resultados foram retirados das Tabelas 5.6 e 5.7.

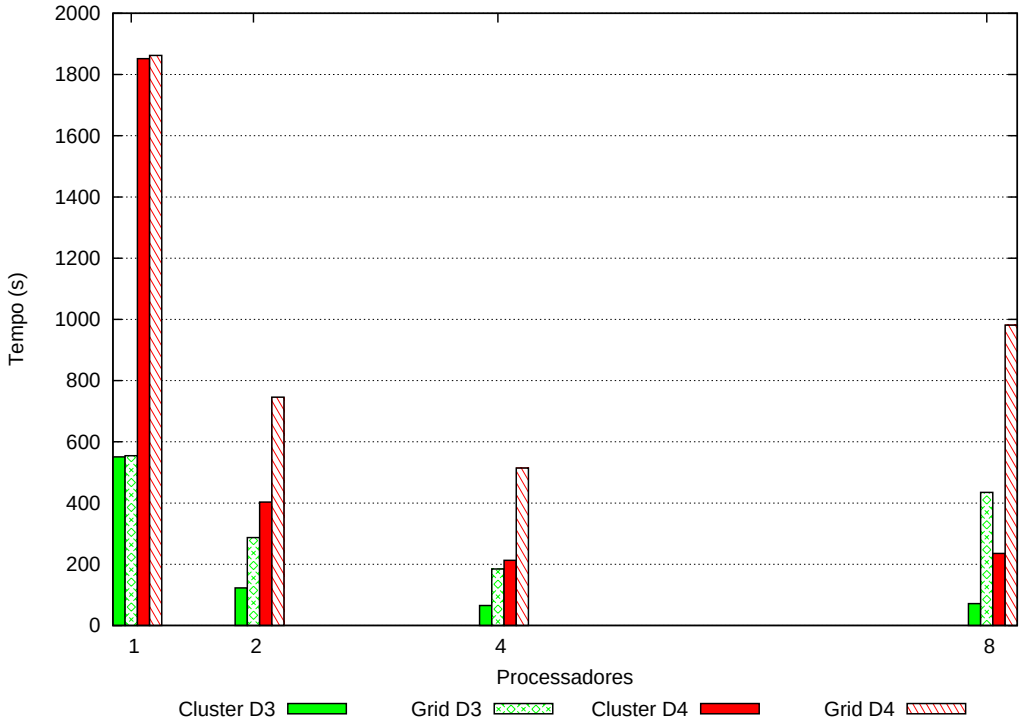


Figura 5.8: Gráfico de comparação entre o *Beowulf* e o InteGrade para o Algoritmo de Jenq e Sahni [33].

Ao avaliarmos os três algoritmos, fica evidente, através da Figura 5.9, que o algoritmo apresentado por Vieira [50] possui um desempenho muito melhor que os demais. Já o algoritmo proposto por Jenq e Sahni [33] possui um desempenho melhor que o algoritmo de Castro Jr. [34], apesar de possuir n rodadas de comunicação, no entanto esse bom resultado só é válido quando são utilizados 2 ou 4 processadores, com 8 o algoritmo de Castro Jr. [34] é superior. Isto ocorre porque quanto maior o número de processadores, o algoritmo de Jenq e Sahni [33] enviará informação para mais processadores e o elevado número de rodadas de comunicação acaba influenciando no resultado final do algoritmo. As limitações do número de estações do *Beowulf* utilizado não possibilitaram uma conclusão mais precisa a respeito da escalabilidade do algoritmo de Jenq e Sahni [33].

Comparando apenas os algoritmos que possuem rodadas de comunicação, a tendência é que, conforme aumentamos o número de processadores, melhor será o resultado apresentado pelo algoritmo de Castro Jr. [34], visto que, no pior caso, ele gastará $O(p)$ rodadas de comunicação, contra $O(n)$ do algoritmo de Jenq e Sahni [33].

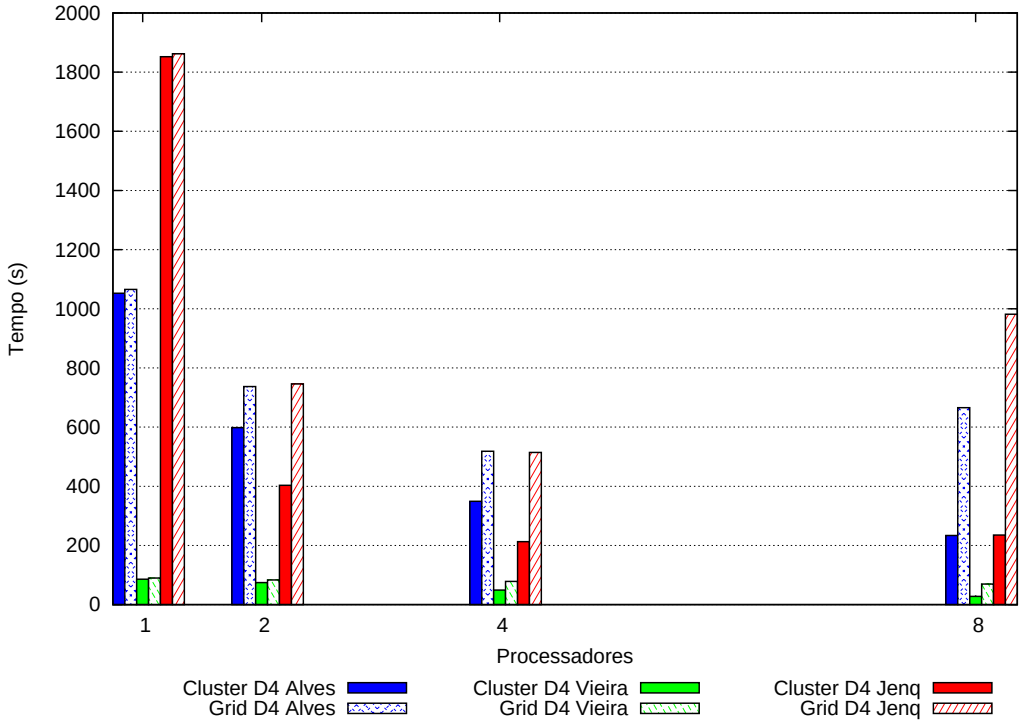


Figura 5.9: Gráfico de comparação entre o *Beowulf* e o InteGrade para os Algoritmos de Fecho Transitivo.

Capítulo 6

Conclusão

Em nosso trabalho, observamos que o modelo BSP/CGM é adequado para o projeto de algoritmos paralelos para máquinas paralelas "reais", uma vez que a implementação de algoritmos, para problemas que utilizam muita comunicação, apresentou resultados compatíveis com os resultados teóricos apresentados na bibliografia. Além disso, implementamos algoritmos BSP/CGM em *cluster* (*Beowulf*) com bons resultados de desempenho, também utilizamos o aglomerado (*grid*) InteGrade e também conseguimos obter em vários casos *speedups* compatíveis com os observados no *Beowulf*. Portanto, a utilização de grades computacionais na realização de tarefas que demandem alto poder computacional mostra-se como uma alternativa viável de ser implementada.

Como pontos positivos, destacamos a possibilidade de se utilizar uma enorme quantidade de máquinas para solucionar problemas complexos a um custo relativamente baixo e os bons resultados para aplicações com poucas rodadas de comunicação, quando os tamanhos das mensagens não é muito grande e quando os processadores se comunicam com poucos processadores.

Os problemas onde o InteGrade não obteve bons resultados serão objeto de análise para os desenvolvedores do *middleware* InteGrade, pois apesar do custo adicional de tempo provocado pelas rotinas de segurança, esperava-se um melhor resultados dessas aplicações no InteGrade.

Os resultados obtidos no Capítulo 3 mostraram um bom desempenho do InteGrade. Além de mostrarmos o bom desempenho do InteGrade, apresentamos um algoritmo BSP/CGM sistólico para resolver o Problema da Mochila 0-1. Neste algoritmo, os dados trocados entre os processadores possuem tamanho fixo, além disso cada processador comunica-se no máximo com outros dois processadores, contribuindo para o bom desempenho do InteGrade em relação ao *Beowulf*. Outro ponto importante é que o algoritmo da mochila pode ser estendido para os problemas do LCS (*Longest Common Subsequence*) [9] e Parentização na Multiplicação de Matrizes [10].

No Capítulo 4, implementamos o algoritmo proposto por Cáceres *et al* [8] para o Problema da Árvore Geradora. O algoritmo possui $O(\log\ p)$ rodadas de comunicação, contudo o InteGrade não obteve um bom resultado. Apesar do pequeno número de

rodadas de comunicação, em cada rodada é necessário uma grande troca de dados entre todos os processadores para realizar a compactação do grafo para a próxima iteração do algoritmo. Por outro lado, os resultados obtidos no *Beowulf* mostram que o algoritmo proposto por Cáceres *et al* [8] tem um bom desempenho. Esses resultados mostram que a não utilização do *list ranking* é uma boa estratégia na obtenção de um algoritmo paralelo para o Problema da Árvore Geradora.

No Capítulo 5, apresentamos diferentes algoritmos para computar o fecho transitivo de um grafo. Cada algoritmo possui um tipo diferente de comunicação entre os processadores, além de possuir quantidades diferentes de rodadas.

O algoritmo apresentado por Alves *et al* [3] e Castro Jr. [34] possui complexidade $O(\frac{n^3}{p})$ em computação local e $O(p)$ rodadas de comunicação. O número de arestas enviada/recebida por cada processador varia consideravelmente em cada rodada, além disso, todos os processadores comunicam-se entre si. Com essa grande quantidade de dados trocados entre todos os processadores, o InteGrade não teve um bom desempenho, os tempos obtidos foram bem distantes dos obtidos no *Beowulf*.

Observando o trabalho de Alves *et al* [3] e Castro Jr. [34], Vieira [50] observou o alto custo relacionado à comunicação na computação do fecho transitivo, e apresentou um algoritmo baseado em uma busca em profundidade com complexidade $O(\frac{n}{p}(n+m))$ de computação e sem rodadas de comunicação. Neste caso, o InteGrade obteve bons e mals resultados em relação ao *Beowulf*.

Utilizando as ideias apresentadas por Jenq e Sahni [33], implementamos um algoritmo com complexidade $O(\frac{n^3}{p})$ de computação local e com $O(n)$ rodadas de comunicação. Neste algoritmo, utilizamos a distribuição por faixas em nossos testes, por isso apenas 1 processador envia informação para todos os demais, entretanto o desempenho do InteGrade não foi satisfatório. Isso porque há um grande número de rodadas de comunicação apesar da quantidade de dados ser relativamente pequena.

Com essas implementações, utilizando um *Beowulf*, pudemos comparar os dois principais algoritmos existentes para o Problema do Fecho Transitivo. Os resultados obtidos indicam que o algoritmo proposto por Alves *et al* [3] torna-se competitivo quando o número de processadores aumenta.

Na Tabela 6.1 apresentamos uma síntese das principais características dos algoritmos apresentados neste trabalho.

O desenvolvimento deste trabalho resultou, direta ou indiretamente, na publicação de alguns artigos [9, 10, 11, 12, 27].

Entre os trabalhos futuros podemos incluir:

1. Implementar do algoritmo proposto por Dehne *et al* [22] para computar os componentes conexos de um grafo G, além de outros algoritmos existentes para árvore geradora e componentes conexos para efetuar uma comparação.

2. Adaptar o algoritmo de árvore geradora de Cáceres *et al* [8] para computar a árvore geradora mínima.

3. Implementar algoritmos que utilizam PVM do fecho transitivo e fazer um comparação global utilizando um número grande de processadores.

Algoritmos	Características
Algoritmo 2 Algoritmo de Frente de Onda	✓ $O(p)$ rodadas de comunicação; ✓ Cada processador comunica-se apenas com 2 processadores em uma rodada de comunicação; ✓ Em cada rodada são trocados $O(\alpha W/p)$ dados entre os processadores; ✓ *Beowulf* e InteGrade possuem tempos parecidos.
Algoritmo 3 Algoritmo de Cáceres *et al* [8]	✓ $O(\log p)$ rodadas de comunicação; ✓ Em cada rodada de comunicação, todos os processadores podem se comunicar; ✓ Quantidade de dados enviado/recebido em cada rodada não é fixo; ✓ Resultados divergentes entre o *Beowulf* e o InteGrade.
Algoritmo 6 Algoritmo de Alves *et al* [3]	✓ $O(p)$ rodadas de comunicação no pior caso; ✓ Durante uma rodada de comunicação todos os processadores comunicam-se entre si; ✓ Muitos dados são enviados em cada rodada de comunicação; ✓ Desempenho inferior do InteGrade em relação ao *Beowulf*.
Algoritmo 7 Algoritmo de Vieira [50]	✓ Não possui rodadas de comunicação; ✓ Nenhuma informação é trocada entre os processadores; ✓ Obteve bons e mals resultados no InteGrade;
Algoritmo 8 Algoritmo de Jenq e Sahni [33]	✓ Gasta $O(n)$ rodadas de comunicação; ✓ Apenas um processador comunica-se com todos os demais em cada rodada de comunicação; ✓ Em cada rodada $O(n)$ dados são enviados; ✓ Desempenho ruim no InteGrade.

Tabela 6.1: Características dos algoritmos implementados.

Referências Bibliográficas

[1] F. Almeida, F. Garcia, D. Morales, J. Roda, e C. Rodrigues. Integral Knapsack Problems: Parallel Algorithms and theirs Implementations on Distributed Systems. In *Proc. of the ACM-ICS 95*, pp. 218–226. 1995.

[2] C. E. R. Alves, E. N. Cáceres, F. Dehne, e S. W. Song. A Parallel Wavefront Algorithm for Efficient Biological Sequence Comparison. In V. Kumar, M. L. Gavrilova, C. J. K. Tan, e P. L'Ecuyer, editores, *ICCSA (2)*, volume 2668 de *Lecture Notes in Computer Science*, pp. 249–258. Springer, 2003. ISBN 3-540-40161-X.

[3] C. E. R. Alves, E. N. Cáceres, A.A. Castro Jr., S. W. Song, e J.L. Szwarcfiter. Efficient Parallel Implementation of Transitive Closure of Digraphs. In J. Dongarra, D. Laforenza, e S. Orlando, editores, *PVM/MPI*, volume 2840 de *Lecture Notes in Computer Science*, pp. 126–133. Springer, 2003. ISBN 3-540-20149-1.

[4] R. Andonov, F. Raimbault, e P. Quinton. Dynamic Programming Parallel Implementations for Knapsack Problem. Relatório Técnico RI 740, IRISA, 1993.

[5] F. R. Arruda. *Algoritmos Paralelos para o Problema da Mochila*. Dissertação de Mestrado, Universidade de São Paulo, Agosto 2004.

[6] S. Baase. *Computer Algorithms: Introduction to Design and Analysis*. Addison-Weley, 1978.

[7] D. A. Bader e G. Cong. A Fast, Parallel Spanning Tree Algorithm for Symmetric Multiprocessors (SMPs). *J. Parallel Distrib. Comput.*, 65(9):994–1006, 2005. ISSN 0743-7315.

[8] E. N. Cáceres, F. Dehne, H. Mongelli, S. W. Song, e J. L. Szwarcfiter. A Coarse-Grained Parallel Algorithm for Spanning Tree and Connected Components. In Marco Danelutto, Marco Vanneschi, e Domenico Laforenza, editores, *Euro-Par*, volume 3149 de *Lecture Notes in Computer Science*, pp. 828–831. Springer, 2004. ISBN 3-540-22924-8.

[9] E. N. Cáceres, H. Mongelli, L. Loureiro, C. Nishibe, e S. W. Song. Performance Results of Running Parallel Applications on the InteGrade. In *2nd. International Latin American Grid Workshop (LAGrid 2008). In 20th International Symposisum on Computer Architecture and High Performance Computing (SBAC-PAD)*, pp. 25–30. Campo Grande, Brazil, October 29 - November 1, 2008.

[10] E. N. Cáceres, H. Mongelli, L. Loureiro, C. Nishibe, e S. W. Song. A Parallel Chain Matrix Product Algorithm on the InteGrade Grid. In *HPC-Asia*, pp. 304–311. Kaohsiung, Taiwan, March 2-5, 2009.

[11] E. N. Cáceres, H. Mongelli, C. Nishibe, e H. C. Sandim. Implementações em Grades Computacionais de Algoritmos BSP/CGM para os Problemas da Mochila 0-1 e Mínimo Intervalar. In *VII Workshop em Sistemas Computacionais de Alto Desempenho (WSCAD 2006). In 18th International Symposisum on Computer Architecture and High Performance Computing (SBAC-PAD)*, pp. 81–88. Ouro Preto, Brazil, October 17-20, 2006.

[12] E. N. Cáceres e C. Nishibe. 0-1 Knapsack Problem: BSP/CGM Algorithm and Implementation. In S. Q. Zheng, editor, *IASTED PDCS*, pp. 331–335. IASTED/ACTA Press, 2005. ISBN 0-88986-525-6.

[13] A. Chan e F. Dehne. A Note on Coarse Grained Parallel Integer Sorting. *Parallel Processing Letters*, 9(4):533–538, 1999.

[14] K. J. Chan, A. Gibbons, M. Pias, e W. Rytter. On the PVM Computations of Transitive Closure and Algebraic Path Problems. In *Proceedings of the 5th European PVM/MPI Users' Group Meeting on Recent Advances in Parallel Virtual Machine and Message Passing Interface*, pp. 338–345. Springer-Verlag, London, UK, 1998. ISBN 3-540-65041-5.

[15] Gen-Huey Chen, Maw-Sheng Chern, e Jin Hwang Jang. Pipeline Architectures for Dynamic Programming Algorithms. *Parallel Computing*, 13(1):111–117, 1990.

[16] C. Chung, M. S. Hung, e W. O. Rom. A Hard Knapsack Problem. *Naval Research Logistics*, 35:85–98, 1988.

[17] D. Coppersmith e S. Winograd. Matrix Multiplication Via Arithmetic Progressions. In *STOC '87: Proceedings of the nineteenth annual ACM conference on Theory of computing*, pp. 1–6. ACM Press, New York, NY, USA, 1987. ISBN 0-89791-221-7.

[18] T. H. Cormen, C. E. Leiserson, R. L. Rivest, e C. Stein. *Introduction to Algorithms - Second Edition*. McGraw-Hill, 2001.

[19] E. N. Cáceres, S. W. Song, e J. L. Szwarcfiter. A Parallel Algorithm for Transitive Closure. In S. G. Akl e T. F. Gonzalez, editores, *IASTED PDCS*, pp. 114–116. IASTED/ACTA Press, 2002. ISBN 0-88986-366-0.

[20] F. Dehne. Coarse Grained Parallel Algorithms. *Special Issue of Algorithmica*, 24(3/4):173–176, 1999.

[21] F. Dehne, A. Fabri, e A. Rau-Chaplin. Scalable Parallel Computational Geometry for Coarse Grained Multicomputers. *International Journal on Computational Geometry*, 6:298–307, 1996.

[22] F. Dehne, A. Ferreira, E. N. Caceres, S. W. Song, e A. Roncato. Efficient Parallel Graph Algorithms for Coarse-Grained Multicomputers and BSP. *Algorithmica*, 33(2):183–200, 2002.

[23] M. R. Garey e D. S. Johnson. *Computers and Intractability - A Guide to the Theory of NP-completeness*. W. H. Freeman and Company, 1979.

[24] R. Garfinkel e G. Nemhauser. *Integer Programming*. John Wiley and Sons, 1972.

[25] P. C. Gilmore e R. E. Gomory. The Theory and Computation of Knapsack Functions. *Operations Research*, 14:1045–1074, 1966.

[26] A. Goldchleger. *InteGrade: Um Sistema de Middleware para Computação em Grade Oportunista*. Dissertação de Mestrado, Universidade de São Paulo - USP - São Paulo/SP - Brasil, Dezembro 2004.

[27] A. Goldchleger, F. Kon, S. W. Song, e E. N. Cáceres et al. The InteGrade Project: Status Report. In *Proceedings of the III Workshop on Computational Grids and Applications WCGA*, pp. 1–6. LNCC, Petrópolis, RJ, Brazil, 2005.

[28] S. Götz. *Communication-Efficient Parallel Algoritms for Minimum Spanning Tree Computation*. Tese de Doutoramento, University of Paderborn, 1998.

[29] M. Habib, M. Morvan, e J. X. Rampon. On the Calculation of Transitive Reduction Closure of Orders. *Discrete Math.*, 111(1-3):289–303, 1993. ISSN 0012-365X.

[30] S. Halperin e U. Zwick. An Optimal Randomized Logarithmic Time Connectivity Algorithm for the EREW PRAM (Extend Abstract). In *SPAA '94: Proceedings of the Sixth Annual ACM Symposium on Parallel Algorithms and Architectures*, pp. 1–10. ACM, 1994. ISBN 0-89791-671-9.

[31] D. S. Hirschberg, A. K. Chandra, e D.V. Sarwate. Computing Connected Components on Parallel Computers. *Commun. ACM*, 22(8):461–464, 1979.

[32] T. C. Hu. *Combinatorial Algorithms*. Addison Wesley, 1982.

[33] J. Jenq e S. Sahni. All Pairs Shortest Paths on a Hypercube Multiprocessor. In *ICPP - International Conference on Parallel Processing*, pp. 713–716. 1987.

[34] A. A. Castro Jr. *Implementação e Avaliação de Algoritmos BSP/CGM para o Fecho Transitivo e Problemas Relacionados*. Dissertação de Mestrado, Universidade Federal de Mato Grosso do Sul - UFMS - Campo Grande/MS - Brasil, Março 2003.

[35] J. JáJá. *An Introduction to Parallel Algorithms*. Addison-Wesley, 1992.

[36] F. Kon e A. Goldman. *Grades Computacionais: Conceitos Fundamentais e Casos Concretos*, volume 1, capítulo 2, pp. 55–104. Belém - PA, 2008.

[37] A. Koubková e V. Koubek. Algorithms for Transitive Closure. *Inf. Process. Lett.*, 81(6):289–296, 2002. ISSN 0020-0190.

[38] S. Martello e P. Toth. *Knapsack Problems: Algorithms and Computer Implementations*. John Wiley and Sons, 1990.

[39] H. Mongelli e R. C. Sakamoto. Implementações de Algoritmos Paralelos FPT para o Problema da k-Cobertura por Vértices Utilizando Clusters e Grades Computacionais. In *VIII Workshop em Sistemas Computacionais de Alto Desempenho (WSCAD 2007). In 18th International Symposisum on Computer Architecture and High Performance Computing (SBAC-PAD)*, pp. 155–162. Gramado, Brazil, October 24–27, 2007.

[40] E. Nuutila. Efficient Transitive Closure Computation in Large Digraphs. *Acta Polytechnica Scandinavia: Math. Comput. Eng.*, 74:1–124, 1995. ISSN 1237-2404.

[41] A. Pagourtzis, I. Potapov, e W. Rytter. PVM Computation of the Transitive Closure: The Dependency Graph Approach. In *Proceedings of the 8th European PVM/MPI Users' Group Meeting on Recent Advances in Parallel Virtual Machine and Message Passing Interface*, pp. 249–256. Springer-Verlag, London, UK, 2001. ISBN 3-540-42609-4.

[42] A. Pagourtzis, I. Potapov, e W. Rytter. Observations on Parallel Computation of Transitive and Max-Closure Problems. In *Proceedings of the 9th European PVM/MPI Users' Group Meeting on Recent Advances in Parallel Virtual Machine and Message Passing Interface*, pp. 217–225. Springer-Verlag, London, UK, 2002. ISBN 3-540-44296-0.

[43] J. H. Reif. Depth-First Search is Inherently Sequential. *Inf. Process. Lett.*, 20(5):229–234, 1985.

[44] R. Roy. Transitivité et Connexité. *C. R. Acad. Sci. Paris*, 249:216–218, 1959.

[45] Y. Shiloach e U. Vishkin. An O(log n) Parallel Connectivity Algorithm. *Journal Algorithms*, 3(1):57–67, 1982.

[46] K. Simon. An Improved Algorithm for Transitive Closure on Acyclic Digraphs. *Theor. Comput. Sci.*, 58(1-3):325–346, 1988. ISSN 0304-3975.

[47] J. L. Szwarcfiter. *Grafos e Algoritmos Computacionais*. Campus, 2a. edição, 1988.

[48] S. Teng. Adaptive Parallel Algorithm for Integral Knapsack Problems. *J. of Parallel and Distributed Computing*, 14:1045–1074, 1990.

[49] L. G. Valiant. A Bridging Model for Parallel Computation. *Commun. ACM*, 33(8):103–111, 1990.

[50] C. C. A. Vieira. *Algoritmo BSP/CGM para o Problema do Fecho Transitivo*. Dissertação de Mestrado, Universidade Federal de Mato Grosso do Sul - UFMS - Campo Grande/MS - Brasil, Setembro 2003.

[51] S. Warshall. A Theorem on Boolean Matrices. *J. ACM*, 9(1):11–12, 1962. ISSN 0004-5411.