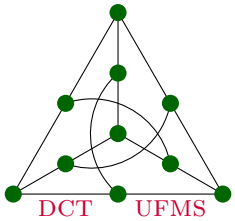
Algoritmos BSP/CGM para Programação Dinâmica

Leonardo Vinícius Rolan Loureiro

Dissertação de Mestrado apresentada à Faculdade de Computação do Centro de Ciências Exatas e Tecnologia da Universidade Federal de Mato Grosso do Sul

Orientação:Prof. Dr. Edson Norberto Cáceres



Algoritmos BSP/CGM para Programação Dinâmica

Este exemplar corresponde à versão final aprovada pela banca do mestrado.

Campo Grande/MS, Março de 2010.

Banca Examinadora:

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

Agradecimentos

Ao meu orientador, professor Dr. Edson Noberto Cáceres, pela paciência, insistência, cobrança e compreensão nos momentos difíceis, sem ele este trabalho se prolongaria por muito mais tempo.

Aos Professores Henrique Mongelli e Siang Wun Song por todas as dicas e contribuições para a preparação dos artigos que antecederam esta dissertação, artigos estes que são a base deste trabalho.

A minha amiga Christiane Nishibe, que me ajudou do primeiro ao último dia, tanto na execução dos algoritmos quanto na preparação da dissertação.

Aos amigos e familiares que acreditaram e nunca me deixaram esquecer da importância deste trabalho.

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

Resumo

A medida que a computação paralela vem deixando de ser um tópico a parte e isolado no mundo da computação para ser um tópico essencial e presente em todas as máquinas recentes, o estudo dos modelos e algoritmos paralelos passa a ser uma obrigação para os futuros cientistas da computação. Neste trabalho abordaremos os principais modelos de computação paralela, desde os modelos teóricos (PRAM) até os modelos reais (BSP, CGM, LogP) mostrando suas principais características, seus pontos de acerto e suas falhas ao modelar as arquiteturas paralelas reais. Dois problemas de grande importância em Programação Dinâmica foram estudados: o problema do Alinhamento Local e o problema do Produto da Cadeia de Matrizes. Para cada um dos problemas apresentados, estudamos e desenvolvemos algoritmos paralelos BSP/CGM usando o paradigma de frente de onda, os algoritmos foram implementados num *cluster* usando a biblioteca LAM-MPI e numa *grid* usando o *middleware* InteGrade. Os tempos obtidos foram os esperados de acordo com a análise de complexidade do modelo BSP/CGM e os resultados mostram que o *overhead* da computação em *grid* é satisfatório considerando as facilidades da mesma.

Palavras-Chaves: Computação Paralela, Programação Dinâmica, Alinhamento Local, Produto da Cadeia de Matrizes, Algoritmos Paralelos BSP/CGM, computação em grid .

Abstract

As parallel computing are no longer an apart and isolated topic in the world of computing to be an essential and present topic in all recent machines, study of parallel models and algorithms becomes an obligation for future scientists computing. This paper will discuss the main parallel computing models, from the theoretical models (PRAM) to the real models (BSP, CGM, LogP) showing its main characteristics, their hit points and their failures to model the real parallel architectures. Two major problems in Dynamic Programming were studied: the Local Alignment problem and the Matrix Chain Product problem. For each of the problems, we study and develop a BSP/CGM algorithm using the wavefront paradigm, the algorithms were implemented in a cluster using the LAM-MPI library and in a grid using the InteGrade middleware. The running times obtained were those expected according to the analysis of complexity of the BSP/CGM model and the results show that the overhead of grid computing is satisfactory considering the facilities of the same.

Keywords: Parallel Computing, Local Alignment, Matrix Chain Product, BSP/CGM Algorithms, grid computing.

Conteúdo

Lista de Figuras

Lista de Tabelas

Capítulo 1

Introdução

O conceito de paralelizar atividades para se ganhar tempo existe há muito tempo, há registros de tábuas de cálculo (*abacus*) com múltiplas posições para a realização de operações simultâneas desde o ano 100 AC [8]. Mas do ponto de vista computacional, a computação paralela surgiu em meados da década de 1970, nessa época os primeiros computadores começaram a ser programados como máquinas paralelas reais. A computação paralela consiste em designar mais de um processador para resolver uma tarefa, essa tarefa é dividida em sub-tarefas, onde a carga de processamento é dividida entre os processadores, resultando em um poder computacional maior do que de apenas um processador. Uma característica importante da computação paralela é a comunicação entre os processadores, sendo que os dois modelos mais utilizados dependem de como está organizada a memória dos processadores. Em caso de memória compartilha, muito popular atualmente por causa dos computadores *multi-core*, a comunicação é feita pela barramento de memória, no caso de memória distribuída, a comunicação é feita por troca de mensagens. Uma tendência atual é o uso de computadores *multi-core* em *clusters*[13], usando tanto o barramento de memória quanto a troca de mensagens para a comunicação. Embora a arquitetura *multi-core* possua um *overhead* de comunicação menor do que a troca de mensagens, esta ainda é limitada pelo número de processadores. A popularização dos computadores *multi-core* aumenta a visibilidade e importância da computação paralela, tanto que alguns cientistas da computação[39][57] pregam o ensino da computação paralela não mais como uma matéria separada, mas como parte de qualquer curso de algoritmos e programação. É bom lembrar que o desenvolvimento de processadores *multi-core* foi causado em boa parte por limitações físicas e de super-aquecimento dos processadores de um núcleo.

O *speedup* ideal e buscado na computação paralela é linear quanto ao número p de processadores, isto é, o programa paralelo deve rodar p vezes mais rápido que seu equivalente sequencial, porém o trabalho de *Wood* [59] mostra que mesmo com *speedup* inferiores a paralelização pode ter um custo-benefício considerável. Além do ganho em tempo, a computação paralela provê um aumento na confiabilidade (mesmo que um processador para de funcionar os outros podem terminar a tarefa) e escalabilidade.

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.

Seguindo esta evolução de *hardware* e *software*, vários modelos de computação paralela foram propostos, os algoritmos paralelos que serão implementados nesse trabalho utilizarão o modelo BSP/CGM. O modelo BSP (*Bulk Synchronous Parallel*) foi proposto por *Valiant* [56] 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" vem do fato que o tamanho do problema n é consideravelmente maior que o número de processadores p, ou seja, $n \gg p$.

Para permitir a comunicação entre os processadores paralelos em um *cluster* será utilizado a especificação de biblioteca de troca de mensagens MPI (*Message-Passing Interface*). O MPI foi desenvolvido entre 1993 e 1997 e tornou-se um padrão de fato, pois mesmo sem ser adotada por nenhum grande órgão de padronização, é largamente usado na computação paralela, possuindo diversas implementações para as principais linguagens e plataformas. Enquanto o MPI é o padrão para comunicação usando troca de mensagens, o OpenMP é o padrão de comunicação para modelos de memória compartilhada [42], mais informações sobre o OpenMP podem ser encontrada em [19].

Para a computação em *grid* será utilizado o *middleware* InterGrade. O Projeto InteGrade [31][1] visa a construção de um *middleware* que permite a implementação de uma computação em *grid* com recursos não-dedicados, usando a capacidade ociosa normalmente disponível em um laboratório de computação. O Projeto InteGrade foi desenvolvido em conjunto por pesquisadores de várias instituições: Departamento de Ciência da Computação da Universidade de São Paulo, Departamento de Informática da Pontifícia Universidade Católica do Rio de Janeiro, Universidade Federal do Maranhão e Departamento de Computação e Estatística da Universidade Federal do Mato Grosso do Sul.

InteGrade tem uma arquitetura orientada a objeto, onde cada módulo do sistema comunica-se com outro módulo através da invocação de um método remoto. InteGrade usa CORBA [45] como infraestrutura de objetos distribuídos, com o benefício de uma arquitetura elegante e sólida. Facilitando assim a implementação, já que a comunicação com módulos do sistema é abstrata da invocação de métodos remotos.

InteGrade foi projetado com o objetivo de permitir o desenvolvimento de aplicações para resolver uma vasta gama de problemas em paralelo. Outros sistemas de computação em *grid* restringem seu uso para problemas que podem ser decompostos em tarefas independentes, tais como *Bag-of-Tasks* [52] ou aplicações parametrizadas. Em adição a lidar com aplicações *Bag-of-Tasks*, InteGrade também lida com aplicações paralelas que dependem de comunicação entre processadores.

Uma questão que surge quando usamos computação em *grid* é o *overhead* que o *middleware* adiciona para lidar com tarefas como submissão, *checkpoint*, segurança, migração de tarefa, etc., em contraste com rodar algoritmos paralelos em um *cluster* sem um *middleware*. Neste trabalho compararemos a execução de algoritmos em um *cluster* usando apenas MPI e em uma *grid* usando InteGrade.

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.

A Biologia Molecular é uma área de utilização intensa de computação e se defronta com alguns dos principais desafios científicos. Neste contexto, a busca de ferramentas que identifiquem, armazenem, comparem e analisem efetivamente números grandes e crescentes de biosequências tem se tornado cada vez mais importante. Biosequências são rotineiramente comparadas ou alinhadas, numa grande variedade de formas, para inferir hereditariedade, para detectar equivalência funcional, ou simplesmente enquanto buscamos entradas parecidas em um banco de dados. Por isso é necessário ter algoritmos eficientes para o problema do Alinhamento Local.

O problema do Encadeamento de Matrizes consiste em determinar a melhor ordem, que minimiza o número de operações de multiplicação, de multiplicar uma cadeia de matrizes. A grande utilidade desse problema é que ele pode ser generalizado para um problema mais abstrato: dado uma seqüência linear de objetos S e uma operação binária associativa O sobre esses objetos, calcular o custo mínimo de aplicar O sobre S. Exemplos dessa abstração é concatenar uma seqüência de *strings* e a triangulação de polígonos [36].

Capítulo 2

Modelo Computacional

O modelo de arquitetura mais utilizado na computação paralela é o modelo *Single Program Multiple Data* (SPMD)[42], neste modelo o mesmo programa (código executável) é executado em todos os processadores, mas cada processador trabalha com um conjunto diferente de dados. A comunicação dos processadores pode ocorrer via uma memória global (computadores *multi-core*) ou via troca de mensagens utilizando uma rede de interconexão. A seguir vamos apresentar os principais modelos de computação paralela, que diferente da computação sequencial onde são analisados apenas o tempo de execução e a memória utilizada, os modelos paralelos devem analisar o número de processadores, o custo de comunicação e a topologia da rede.

2.1 Modelo PRAM

O primeiro modelo de computação paralela existente é o *Parallel Random Access Machine* (PRAM). A PRAM define uma máquina paralela síncrona com memória global acessível a todos os processadores. A Figura 2.1 exemplifica o modelo PRAM. No PRAM o número de processadores disponíveis é ilimitado e em função do tamanho da entrada do problema, por exemplo, no algoritmo paralelo de aproximação de Czumaj [18] para o problema do Encadeamento de Matrizes o número de processadores p utilizados em função da entrada de tamanho n é $p = \frac{n^6}{\log^6 n}$. Além de considerar um número ilimitado de processadores disponíveis, os algoritmos projetados para este modelo não levam em conta a comunicação. 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 seqüêncial. Um outro ponto também crucial era a falta de portabilidade das implementações (muito dependente da topologia). Por esses motivos o PRAM é considerado um modelo abstrato, de fácil aplicação na teoria mas pouca aplicabilidade na prática. Durante os anos 90 as pequisas em torno do PRAM diminuíram drasticamente, ao ponto de edições mais novas de alguns livros descartarem os capítulos sobre PRAM [58], porém a nova onda de computadores *multi-core*, *Massive Parallel Processing* (MPP) e *eXplicit Multi-Threading* (XMT) [58] trouxeram o PRAM

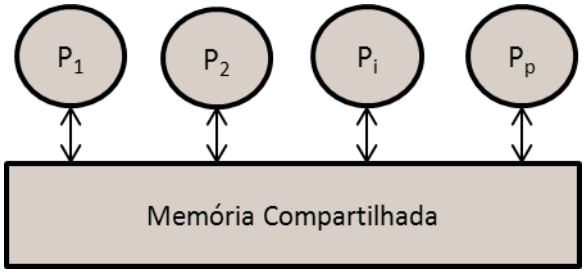


Figura 2.1: Modelo PRAM

de volta a ativa com modelos derivados que limitam o número de processadores [48]. Dependendo das restrições (exclusiva ou concorrente) de acesso a memória global para leitura e escrita, classificamos os algoritmos do PRAM em quatro tipos:

1. *Exclusive Read Exclusive Write* (EREW): O mais restritos dos modelos, onde apenas um processador pode acessar uma determinada posição na memória, seja para leitura ou escrita, ao mesmo tempo;
2. *Concurrent Read Exclusive Write* (CREW): Dois ou mais processadores podem acessar ao mesmo tempo uma mesma posição de memória para leitura, mas apenas um processador pode escrever numa determinada posição por vez;
3. *Exclusive Read Concurrent Write* (ERCW): Apenas um processador por vez pode ler uma dada posição na memória, mas uma mesma posição da memória pode ser escrita por dois ou mais processadores;
4. *Concurrent Read Concurrent Write* (CRCW): O modelo que dá mais liberdade ao programador, onde dois ou mais processadores podem acessar a mesma posição na memória, seja para leitura ou escrita, ao mesmo tempo.

Esses quatro modelos definem que o acesso a memória é em tempo constante, independente da quantidade de processadores que tentam acessar uma mesma posição. Porém Gibbons, Matias e Ramachandran[27] argumentam que estas regras não se aplicam as máquinas paralelas atuais, por isso eles propuseram uma alternativa mais realista, chamado de *Queue Read Queue Write* (QRQW), onde acesso concorrente é permitido, mas gasta um tempo proporcional ao número de acessos concorrentes [41].

2.2 Modelo BSP

No início dos anos 90, *Valiant* [56] introduz 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 Model* - BSP, além de ser um dos modelos realísticos mais importantes, foi um dos primeiros a considerar os custos

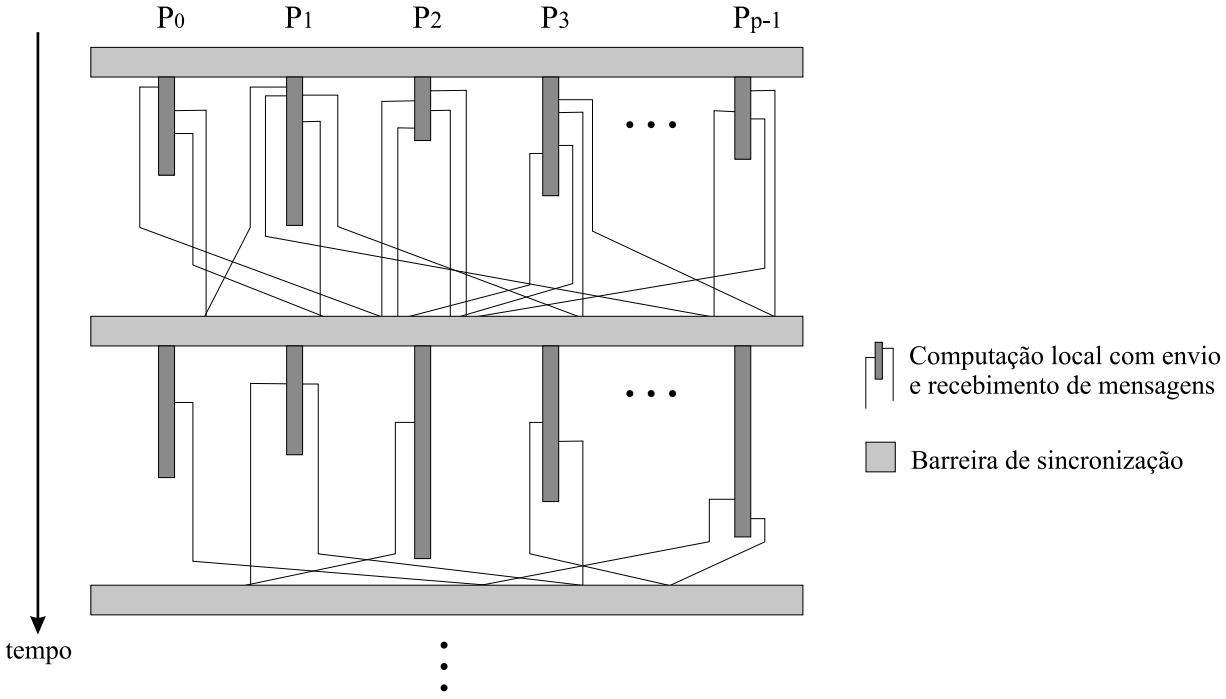


Figura 2.2: Modelo BSP

de comunicação e a abstrair as características de uma máquina paralela em um pequeno número de parâmetros.

O modelo BSP consiste de um conjunto de p processadores com memória local, onde a comunicação é feita através de algum meio de interconexão, gerenciados por um roteador e com facilidades de sincronização global. Um algoritmo BSP consiste numa seqüência de super-passos separados por barreiras de sincronização. Em um super-passo, 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 super-passo, e passos de comunicação, através de instruções de envio e recebimento de mensagens. Neste modelo uma h-relação em um super-passo corresponde ao envio e/ou recebimento de, no máximo, h mensagens em cada processador. A resposta a uma mensagem enviada em um super-passo somente será utilizada no próximo super-passo. A Figura 2.2 exemplifica o modelo BSP.

O tempo de comunicação em um algoritmo no modelo BSP é dado por uma função de custo com dois parâmetros: (1) o gap g que reflete a largura de banda, e (2) a latência L que é a duração mínima de um super-passo [32]. O parâmetro g depende da largura de banda, mas também é efetado pelos protocolos de comunicação e algoritmos de roteamento [35], por isso a melhor forma de determinar g é através de *benchmarks*, um protocolo para realizar isto é descrito em [53]. Assim como g, L também é melhor determinado através da execução de um *benchmarks*. Uma propriedade interessante do modelo BSP é que os valores de g e L não interferem no resultado do programa [7].

Considere um programa BSP com S super-passos, então o tempo de execução do super-passo i é dado pela fórmula $S_i = w_i + gh_i + L$, onde w_i é a maior computação

local realizada e h_i é o maior número de mensagens enviadas ou recebidas por algum processador durante o super-passo. O tempo de execução T do programa é dado por $T = \sum_{i=0}^{S-1} w_i + g \sum_{i=0}^{S-1} h_1 + LS$. *Hill* [34] argumenta que um modelo paralelo de sucesso deve possuir três propriedades:

1. Escalabilidade: a performance do *software* e *hardware* deve ser escalável de um para centenas de processadores;
2. Portabilidade: o *software* deve rodar sem mudanças e com alta performance em qualquer arquitetura de propósito geral;
3. Previsibilidade: a performance do *software* em diferentes arquiteturas deve ser previsível.

Pesquisas com algoritmos, arquiteturas e linguagens BSP mostram que o modelo atende as três propriedades [34].

2.3 Modelo CGM

O modelo *Coarse Grained Multicomputer* (CGM) que foi proposto por *Dehne et al.* [21] e é uma simplificação do modelo BSP. No CGM os algoritmos realizam sequências de super-passos, onde cada super-passo é dividido em uma fase de comunicação global e uma fase de computação local. A Figura 2.3 exemplifica o modelo CGM. O CGM consite de p processadores idênticos [44] mas não especifica como os processadores estão conectados, assim qualquer meio de interconexão é permitido. Normalmente, durante uma rodada de computação local é utilizado o melhor algoritmo sequencial para o processamento dos dados disponibilizados localmente. A grande diferença entre o BSP e o CGM é que este último considera apenas dois parâmetros para a análise dos algoritmos: (1) o tamanho do problema n e (2) o número de processadores disponíveis p. Para minimizar o custo e o emphoverhead de comunicação, o CGM exige que as mensagens enviadas sejam empacotadas em uma longa mensagem de no máximo$O(\frac{n}{p})$ dados. O *coarse grained* (granularidade grossa), vêm do fato de que o tamanho do problema, n, é consideravelmente maior que o número p de processadores, ou seja, $n >> p$. No CGM todos os processadores possuem a mesma quantidade M de memória, onde $M = O(\frac{n}{p})$.

Um legado do modelo PRAM é que o número de super-passos deve ser polilogaritmo em função de p, porém *Goudreau et al* [32] mostrou que o número de super-passos em função de p (e não n) apresentam resultados melhores na prática [24]. 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 super-passos e a quantidade de computação local.

A simplicidade e eficiência dos modelos BSP e CGM foram base para muitos novos modelos, como o PRO [25] e o LogP [15].

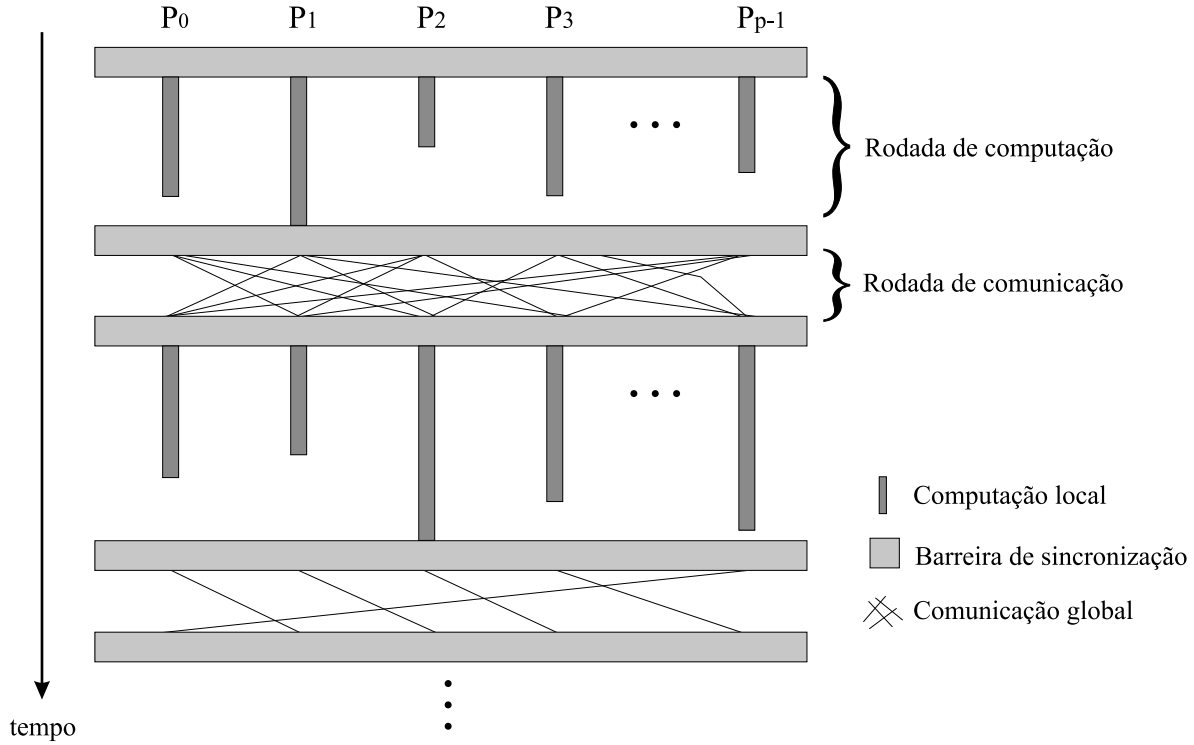


Figura 2.3: Modelo CGM

2.4 Modelo LogP

O LogP foi desenvolvido por *Culler et al* [15] como um modelo assíncrono realístico para o projeto de algoritmos paralelos em que questões críticas para o desempenho podem ser resolvidas sem se perder nos detalhes da arquitetura envolvida [16]. Uma das premissas do LogP é que o algoritmo com melhor desempenho no modelo, será o melhor algoritmo na prática. O modelo BSP foi usado como ponto de partida para a criação do LogP [15], as principais diferenças entre os modelo é a ausência da barreira de sincronização no LogP (modelo assíncrono) e a adição de um novo parâmetro. De acordo com *Bilardi et al* [7] o BSP oferece maior abstração para o projeto do algoritmos e programação, enquanto o LogP provêm melhor controle dos recursos das máquina.

Como não há consenso nas topologias de interconexão, as redes das novas máquinas são diferentes das redes de seus predecessores, que por sua vez já eram diferentes entre si, LogP evite especificar a topologia da rede, assim ele usa parâmetro genéricos que independem da topologia de rede. Os quatro parâmetros usados para calcular o desempenho do algoritmo no modelo LogP são:

1. Latência (L): tempo máximo de transmitir uma mensagem da fonte até o destino, durante este tempo os processadores envolvidos podem realizar outras tarefas;

2. *overhead* (o): tempo que um processador gasta para enviar ou receber uma mensagem, durante este tempo os processadores envolvidos não podem realizar nenhuma outra tarefa;

3. *gap* (g): tempo mínimo entre duas transmissões ou dois recebimentos consecutivos;

4. processadores (P): o número de processadores.

Com relação a comunicação, o LogP supõe que a rede tem capacidade finita, esta capacidade é atingida quando um processador está enviando mensagens a uma taxa maior do que o destino pode receber [16]. As comunicações são ponto-a-ponto onde as mensagens tem tamanho fixo e são pequenas [2].

O tempo total para transferir n mensagens de um processador para outro é $2o + L + (n - 1)g$, onde cada processador gasta $n \times o$ ciclos e o tempo restante fica disponível para realizar outras tarefas. Felizmente, os parâmetros não são igualmente importantes em todas as situações, assim é possível ignorar um ou mais parâmetros para simplificar o modelo [15].

Capítulo 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 [30] 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 [38].

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

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

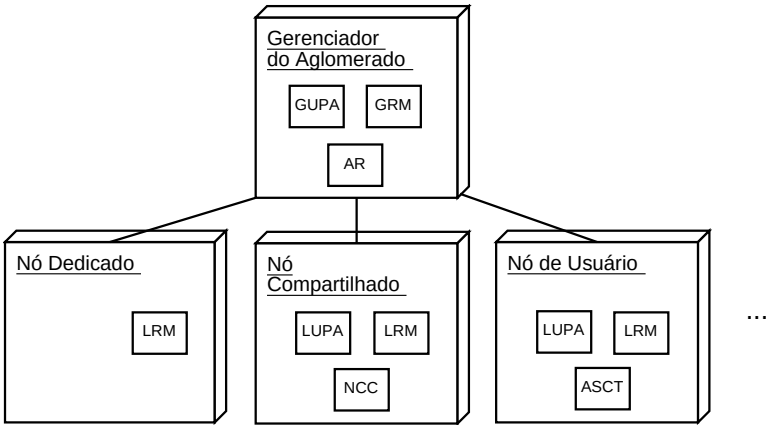


Figura 3.1: Arquitetura intra-aglomerado do InteGrade [29].

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 é 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* [11].

- **Programação Dinâmica e Algoritmos Gulosos**: Considerando que os algoritmos de similaridade de sequências e maior subsequência comum (LCS) de Alves *et al* [4] 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 [9, 10].

- **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 [43].

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.

Capítulo 4

Alinhamento Local

Sejam $A = a_1a_2 \ldots a_m$ e $B = b_1b_2 \ldots b_n$ duas sequências pertencentes ao mesmo alfabeto I. Para alinhar as sequências, inserimos espaços nas duas sequências de tal forma que ambas possuam o mesmo comprimento. Um alinhamento entre A e B é o emparelhamento de símbolos $a \in A$ e $b \in B$ sujeito à restrição de que se linhas são traçadas entre os símbolos emparelhados, as linhas não podem se interceptar. O alinhamento mostra a similaridade entre as duas sequências e o que buscamos é um valor de alinhamento ótimo, ou seja, máximo.

O algoritmo sequencial utilizado para computar a similaridade de duas sequências utiliza a técnica de programação dinâmica [51]. A complexidade desse algoritmo é $O(n \times m)$, onde n e m são os comprimentos das sequências sendo comparadas. Dada a matriz de similaridade, a construção do alinhamento ótimo pode ser computado em tempo $O(n+m)$.

A comparação de sequências é uma das ferramentas fundamentais e mais importantes em Biologia Molecular Computacional, servindo como base para a solução de outros problemas mais complexos [51], tal como a busca de similaridades entre biosequências [46, 50, 54]. Além das aplicações em Biologia Molecular, problemas que utilizam a comparação de sequências incluem a comparação de arquivos [37], correção de ortografia [33] e recuperação de informações [60].

A motivação principal para compararmos sequências biológicas, em particular proteínas, vem do fato de que proteínas, com formas tridimensionais similares geralmente têm a mesma funcionalidade. Mais ainda, a forma tridimensional é ditada pela própria sequência de símbolos que a formam. Assim é possível inferir a função de uma nova proteína através da descoberta de uma outra proteína com função conhecida que seja similar a ela.

A mesma idéia utilizada para identificar similaridade pode ser aplicada quando queremos descobrir a distância de edição entre duas sequências. Neste caso, queremos descobrir qual o número mínimo de operações de inserção, remoção e substituição, que devem ser feitas para, a partir de uma das sequências obtermos a outra.

As noções de similaridade e de distância de edição são muitas vezes intercambiáveis e ambas servem ao propósito de podermos inferir tanto funcionalidade, como também aspectos relacionados à história evolutiva das sequências envolvidas.

Para resolver o problema do Alinhamento Local foi usado o algoritmo de *Smith-Waterman* [55], que é uma variação do algoritmo de Alinhamento Global de *Needleman-Wunsch* [47].

Para duas sequências S_1 e S_2 de tamanho m e n respectivamente, o algoritmo roda em tempo $O(m \times n)$ com espaço em memória de $O(m \times n)$. Para alinhar as sequências o algoritmo considera três possibilidades [49]:

1. Alinhar um elemento de S_1 com um elemento de S_2;
2. Alinhar um elemento de S_1 com um *gap*;
3. Alinhar um elemento de S_2 com um *gap*.

Para cada caso o algoritmo usa uma matriz $m \times n$ que guarda o melhor alinhamento local até a rodada em execução. Recomendamos a leitura do livro de Setubal e Meidanis [51] para mais detalhes do algoritmo seqüêncial. Uma matriz de substituição é usada para pontuar o alinhamento entre dois elementos. Em nossos testes foi usada a matriz de substituição *BLOSUM62*, e para pontuar o alinhamento de um elemento com um *gap* foi usada uma função linear. Uma versão em alto nível do algoritmo de Alinhamento Local pode ser visto em 1

4.1 Algoritmo Paralelo

Dado o tamanho e a quantidade das sequências envolvidas, faz sentido pensarmos na utilização de computação paralela para solucionar esse problema. A obtenção de algoritmos paralelos eficientes para problemas envolvendo programação dinâmica no modelo PRAM foi estudado por *Galil* e *Park* [22, 23]. Algoritmos paralelos, usando o modelo PRAM, para o problema de edição de *strings* foi estudado por *Apostólico et al.* [6]. O problema de edição de *strings* no modelo BSP/CGM foi estudado por *Alves, Cáceres, Dehne e Song* [3, 5]. Um estudo mais geral sobre algoritmos paralelos para Programação Dinâmica pode ser visto em *Gengler* [26]. Neste trabalho desenvolvemos um algoritmo BSP/CGM para o problema do Alinhamento Local baseado no paradigma de frente de onda de [4]. A característica e vantagem do paradigma de frente de onda ou sistólico é a modesta demanda de comunicação pelo fato que cada processador comunica-se com poucos outros processadores, tornando o problema perfeito para a computação em grid. Nós apresentamos um algoritmo BSP/CGM para resolver o problema do Alinhamento Local de duas cadeias S_1 e S_2 de comprimento m e n respectivamente, usando p processadores. O algoritmo realiza $O(p)$ rodadas de comunicação e tem complexidade de tempo igual a $O(m \times n)$.

No algoritmo paralelo a cadeia S_1 é dividida em p partes de tamanho $\frac{m}{n}$, cada processador P_i é responsável por calcular o alinhamento local da i-ésima parte de S_1 com a cadeia S_2. Na primeira rodada ($r = 1$), apenas o processador P_1 realiza trabalho e ao fim da rodada envia a última linha do bloco calculado para P_2. Na rodada seguinte, $r = 2$, P_2 usa os dados recebidos de P_1 para calcular seu bloco (em paralelo P_1 calcula um novo

Algoritmo 1 Algoritmo Seqüêncial de Alinhamento Local

Entrada: (1) Seqüência S_1 e S_2, (2) h: penalidade para começar um gap, (3) g: penalidade para extender um gap

Saída: Melhor Alinhamento Local entre S_1 e S_2

1: $A(S_1.length+1, S_2.length+1)$; // matriz para alinhar um elemento de S_1 com um elemento de S_2
2: $B(S_1.length+1, S_2.length+1)$; // matriz para alinhar um elemento de S_1 com um gap
3: $C(S_1.length+1, S_2.length+1)$; // matriz para alinhar um elemento de S_2 com um gap
4: **for** $i \leftarrow 0$ **to** $S_1.length$ **do**
5: $A[i,0] \leftarrow 0$;
6: $B[i,0] \leftarrow 0$;
7: $C[i,0] \leftarrow 0$;
8: **end for**
9: **for** $j \leftarrow 0$ **to** $S_2.length$ **do**
10: $A[0,j] \leftarrow 0$;
11: $B[0,j] \leftarrow 0$;
12: $C[0,j] \leftarrow 0$;
13: **end for**
14: **for** $i \leftarrow 0$ **to** $S_1.length$ **do**
15: **for** $j \leftarrow 0$ **to** $S_2.length$ **do**
16: $A[i,j] \leftarrow max(A[i-1,j-1], B[i-1,j-1], C[i-1,j-1]) + BLOSUM62[S1[i-1], S2[j-1]]$;
17: $B[i,j] \leftarrow max(-(h+g) + A[i-1,j], -g + B[i-1,j], -(h+g) + C[i,j-1])$;
18: $C[i,j] \leftarrow max(-(h+g) + A[i,j-1], -(h+g) + B[i,j-1], -g + C[i,j-1])$;
19: **end for**
20: **end for**

bloco) e envia a última linha do bloco calculado para P_3 e assim sucessivamente até a rodada $r = p$. A partir da rodada $r = p + 1$ o processador P_1 já terminou de calcular todos os seus blocos e deixa de trabalhar, assim, a cada nova rodada onde $r = p + i$ o processador P_i deixa de trabalhar. Na última rodada apenas o processador P_p estará trabalhando. De forma geral, a cada rodada paralela onde $r = i <= p$ o processador P_i inicia seu trabalho, e a cada rodada paralela onde $r >= p + i$ o processador P_i deixa de trabalhar.

O algoritmo paralelo para o problema do Alinhamento Local é mostrado no Algoritmo 2.

Algoritmo 2 Algoritmo Paralelo para o Alinhamento Local

Entrada: (1) Sequências S_1 de tamanho m e S_2 de tamanho n, (2) Número de processadores p (3) *Rank* i do processador (3) Cada processador i contém $S_1[i * (\frac{n}{p})..(i + 1) * (\frac{n}{p})]$ and $S_2[0..m - 1]$

Saída: Melhor alinhamento local entre S_1 e S_2

1: $blocoH \leftarrow \frac{n+1}{p}$ // tamanho do bloco na horizontal (colunas)
2: $blocoV \leftarrow \frac{m+1}{p}$ // tamanho do bloco na vertical (linhas)
3: matriz A(blocoV+1, n+1), matriz B(blocoV+1, n+1), matriz C(blocoV+1, n+1)
4: $sucessor \leftarrow i + 1$
5: $antecessor \leftarrow i - 1$
6: $col \leftarrow 1$
7: **for** $rodada \leftarrow 0$ **to** $p - 1$ **do**
8: $col \leftarrow col + bloco$
9: **if** $i \neq 0$ **then**
10: recebe $(A[0, blocoH * rodada..(blocoH * (rodada + 1)) - 1], antecessor)$
11: recebe $(B[0, blocoH * rodada..(blocoH * (rodada + 1)) - 1], antecessor)$
12: recebe $(C[0, blocoH * rodada..(blocoH * (rodada + 1)) - 1], antecessor)$
13: **end if**
14: calcular $A[1..blocoV, blocoH * rodada..(blocoH * (rodada + 1)) - 1]$
15: calcular $B[1..blocoV, blocoH * rodada..(blocoH * (rodada + 1)) - 1]$
16: calcular $C[1..blocoV, blocoH * rodada..(blocoH * (rodada + 1)) - 1]$
17: **if** $i \neq p - 1$ **then**
18: enviar $(A[blocoV, blocoH * rodada..(blocoH * (rodada + 1)) - 1], sucessor)$
19: enviar $(B[blocoV, blocoH * rodada..(blocoH * (rodada + 1)) - 1], sucessor)$
20: enviar $(C[blocoV, blocoH * rodada..(blocoH * (rodada + 1)) - 1], sucessor)$
21: **end if**
22: **end for**

4.2 Resultados Experimentais

Nós rodamos o algoritmo BSP/CGM para o problema do Alinhamento Local em um *cluster* composto por 12 nós formado por 6 processadores *Intel Pentium* IV de 1,7 GHz e 6 processadores *AMD Athlon* de 1,6 GHz. Os nós estão conectados por um *switch* 1Gb *Fast-Ethernet*. Os dados usados foram gerados aleatoriamente. O objetivo do experimento

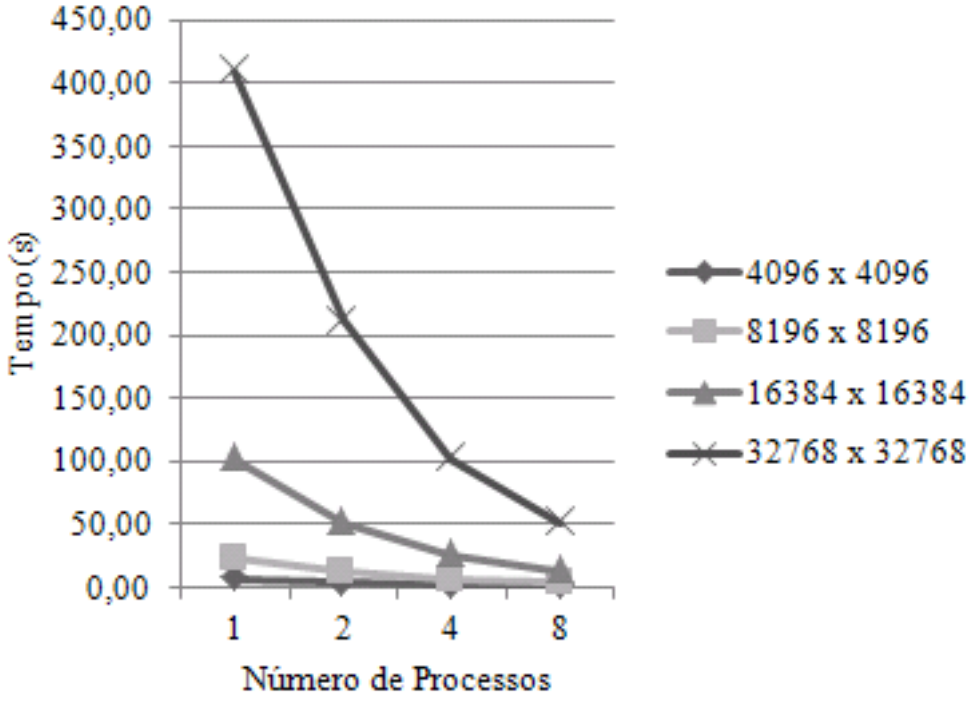


Figura 4.1: Gráfico do Tempo de Execução (em segundos) para o Alinhamento Local em um *cluster* usando LAM-MPI

é comparar a execução usando um *cluster* com LAM-MPI contra uma *grid* usando o InteGrade.

A Tabela 4.2 e a Figura 4.2 mostram os tempos de execução (em segundos) para um *cluster* usando LAM-MPI.

$m \times n$	1	2	4	8
4096×4096	6.481	2.813	1.418	0.715
8196×8196	22.658	11.322	5.639	2.842
16384×16384	101.614	50.616	25.436	12.883
32768×32768	409.418	211.074	101.422	50.680

Tabela 4.1: Tempo de Execução (em segundos) para o Alinhamento Local em um *cluster* usando LAM-MPI

A Tabela 4.2 e a Figura 4.2 mostram os tempos de execução (em segundos) para uma *grid* usando InteGrade.

A tabela 4.3 mostra uma comparação entre os tempos de execução em um *cluster* usando LAM-MPI e em uma *grid* usando o InteGrade. Coluna I e coluna II mostram os tempos para o *cluster* e para a *grid*, respectivamente. A Figura 4.3 mostra o gráfico correspondente.

Podemos observar que os tempos de execução sofrem um decaimento linear quanto ao número de processadores, atingindo o chamado *speedup* ideal. Temos dois motivos que

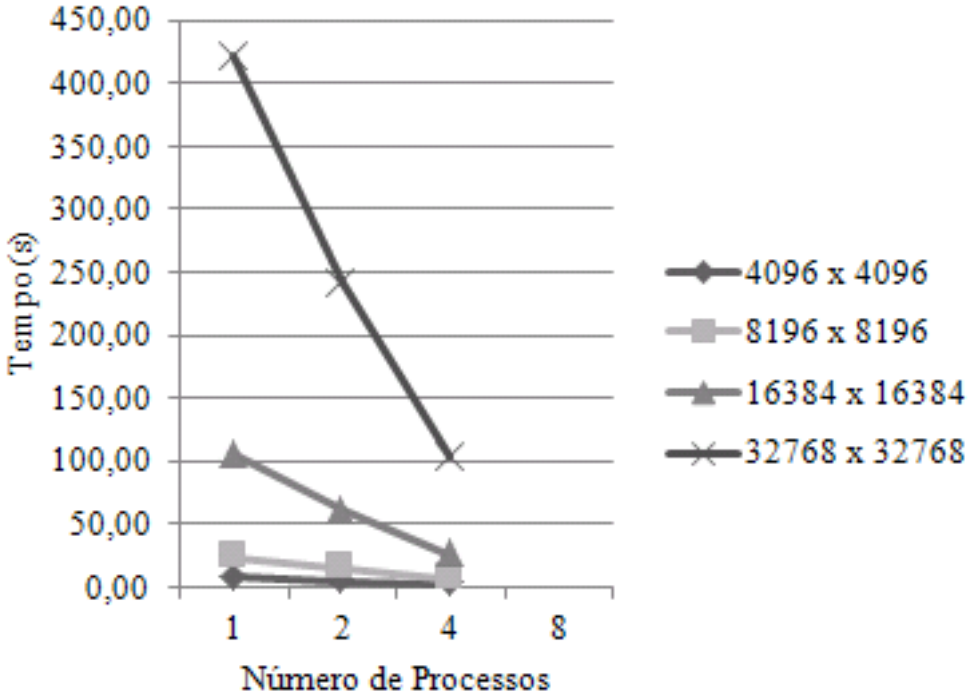


Figura 4.2: Gráfico do Tempo de Execução (em segundos) para o Alinhamento Local em uma *grid* usando InteGrade

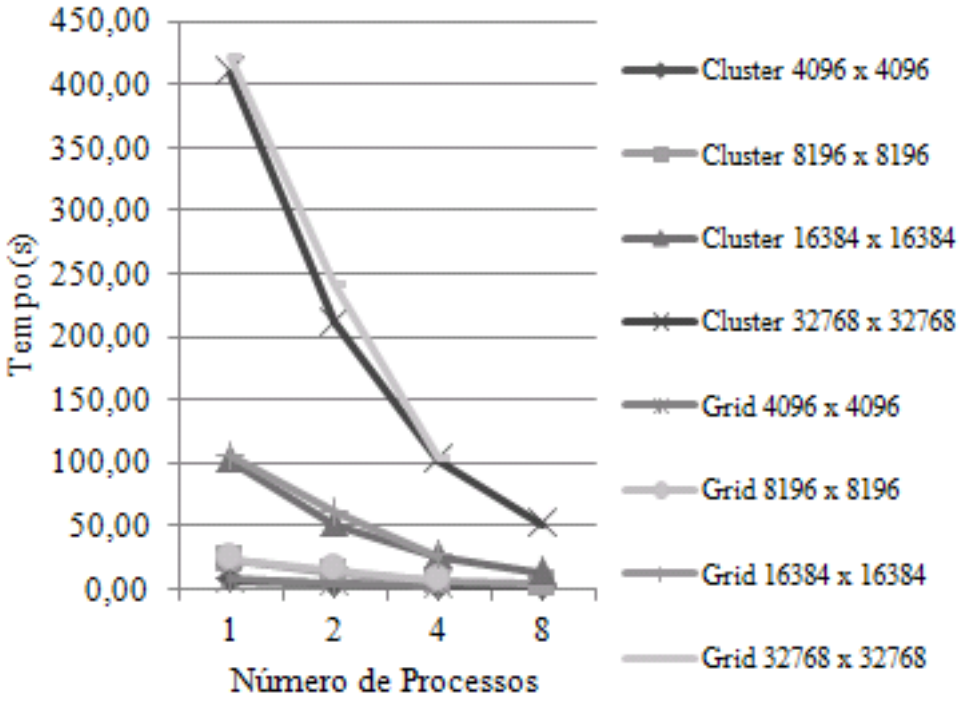


Figura 4.3: Gráfico comparativo dos tempos de execução (em segundos) entre um *cluster* e uma *grid* para o Alinhamento Local

$m \times n$	1	2	4	8
4096×4096	7.204	3.854	1.812	-
8196×8196	23.622	14.889	6.853	-
16384×16384	105.489	60.952	26.070	-
32768×32768	420.705	240.733	103.411	-

Tabela 4.2: Tempo de Execução (em segundos) para o Alinhamento Local em uma *grid* usando InteGrade

	4096×4096		8196×8196		16384×16384		32768×32768	
p	I	II	I	II	I	II	I	II
1	6.481	7.204	22.658	23.622	101.614	105.489	409.418	420.705
2	2.813	3.854	11.322	14.889	50.616	60.952	211.074	240.733
4	1.418	1.812	5.639	6.853	1.418	1.812	5.639	6.853
8	0.715	-	2.842	-	12.883	-	50.680	-

Tabela 4.3: Comparativo dos tempos de execução (em segundos) entre um *cluster* (I) e uma *grid* (II) para o Alinhamento Local

levam a este bom desempenho:

1. O baixo número de troca de mensagens e o tamanho pequeno das mensagens.
2. O uso eficiente da memória RAM, evitando o uso da memória virtual. Como as matrizes são de uma ordem grande, quando dividimos estes matrizes entre os processadores temos uma redução no acesso a disco, melhorando o desempenho.

Capítulo 5

Produto da Cadeia de Matrizes

Considere o cálculo do produto de n matrizes, onde M_i é uma matriz de dimensões $d_i \times d_{i+1}$.

$$M = M_1 \times M_2 \times \ldots \times M_n$$

A multiplicação de matrizes satisfaz a propriedade associativa, logo o resultado final é o mesmo para qualquer ordem que as matrizes sejam multiplicadas. Entretanto, a ordem da multiplicação afeta o número total de operação para calcular M. O problema do Produto da Cadeia de Matrizes é achar a ordem ótima de multiplicar as matrizes, tal que o número de operações é minimizado [40].

O primeiro algoritmo com tempo polinomial para o problema do Produto da Cadeia de Matrizes foi proposto por *Godbole* [28]. O algoritmo usa a técnica de Programação Dinâmica e roda em $O(n^3)$ com espaço em memória de $O(n^2)$.

As idéias principais do uso da Programação Dinâmica para solucionar o problema do Produto da Cadeia de Matrizes serão dadas adiante, maiores detalhes podem encontrados em [14]. Programação Dinâmica é uma técnica que computa a solução de um problema resolvendo primeiramente a solução de subproblemas. A computação segue dos subproblemas menores para os maiores, e as soluções parciais dos subproblemas são guardadas para uso futuro, para não terem que ser computadas novamente.Como exemplo, considere o produto de $n = 4$ matrizes.

$$\underbrace{M}_{10\times100} = \underbrace{M_1}_{10\times20} \times \underbrace{M_2}_{20\times50} \times \underbrace{M_3}_{50\times1} \times \underbrace{M_4}_{1\times100}$$

Sejam as dimensões de M_1, M_2, M_3 e M_4 os valores 10×20, 20×50, 50×1 e 1×100, respectivamente. Em outras palavras, $d_1 = 10$, $d_2 = 20$, $d_3 = 50$, $d_4 = 1$ and $d_5 = 100$. O algoritmo trivial de multiplicação de matrizes para multiplicar uma matriz de dimensões $a \times b$ por outra de dimensões $b \times c$ gasta abc operações, resultando em uma matriz de dimensões $a \times c$. Se calcularmos o produto da seqüência de matrizes na seguinte ordem

$$M_1 \times (M_2 \times (M_3 \times M_4))$$

vamos gastar 125000 operações. Entretanto, se calcularmos o mesmo produto na ordem

$$(M_1 \times (M_2 \times M_3)) \times M_4$$

vamos gastar apenas 2200 operações. A melhor forma de calcular o produto de uma seqüência de matrizes pode ser obtida através do método de Programação Dinâmica como se segue. Suponha que desejamos calcular o produto de n matrizes, onde M_i é uma matriz de dimensões $d_i \times d_i + 1$.

$$M = M_1 \times M_2 \times \ldots \times M_n$$

Seja $m_{i,j}$, $1 \leq i \leq j \leq n$, o custo mínimo (em termos do número de operações) para calcular

$$M_i \times M_{i+1} \times \ldots \times M_j$$

Podemos calcular $m_{i,j}$ pela fórmula:

$$m_{i,j} = \begin{cases} 0 & \text{se } i = j \\ \min_{i \leq k < j}(m_{i,k} + m_{k+1,j} + d_i d_{k+1} d_{j+1}) & \text{se } i < j \end{cases}$$

Para facilitar a compreensão da fórmula, considere o produto abaixo. A expressão a ser minimizada abaixo tenta primeiramente achar o melhor ponto (k) que minimiza o número de operações para computar dois produtos parciais ou sub-cadeias.

$$\overbrace{\underbrace{M_i}_{d_i \text{linhas}} \times M_{i+1} \times \ldots \times \underbrace{M_k}_{d_{k+1} \text{colunas}}}^{\text{calcular primeiro este}} \times \overbrace{\underbrace{M_{k+1}}_{d_{k+1} \text{linhas}} \times \ldots \times \underbrace{M_j}_{d_{j+1} \text{colunas}}}^{\text{calcular primeiro este}}$$

A idéia principal é achar o custo de multiplicar todas as sub-cadeias e combiná-las, primeiro calculamos todas as sub-cadeias de tamanho 2 e guardamos seus custos, depois calculamos todas a sub-cadeias de tamanho 3 usando os custos já calculados e assim sucessivamente [17].

O objetivo é minimizar $m_{1,n}$, assim para calcular o custo de $M = M_1 \times M_2 \times \ldots \times M_n$ primeiro calculamos $m_{i,i}$ (diferença entre dois índices = 0). Como $m_{i,i}$ é uma sub-cadeia de tamanho 1, não há necessidade de multiplicar, logo $m_{i,i} = 0$. Depois calculamos $m_{i,i+1}$ (diferença entre dois índices = 1), $m_{i,i+2}$ (diferença entre dois índices = 2), e assim sucessivamente. Todos os valores calculados são guardados para serem usados para calcular $m_{i,j}$ com uma diferença de índices maior.

No exemplo proposto, desejamos obter m_{14}. Então calculamos os seguintes valores, linha por linha, aumentando a diferença entre os índices.

$m_{11} = 0$	$m_{22} = 0$	$m_{33} = 0$	$m_{44} = 0$
$m_{12} = 10000$	$m_{23} = 1000$	$m_{34} = 5000$	
$m_{13} = 1200$	$m_{24} = 3000$		
$m_{14} = 2200$			

O algoritmo sequencial para o problema do Produto da Cadeia de Matrizes é mostrado no Algoritmo 3.

Algoritmo 3 Algoritmo Seqüêncial para o problema do Produto da Cadeia de Matrizes

Entrada: (1) Vetor $d[0..n]$ contendo as dimensões $d_1, d_2, \ldots, d_n, d_{n+1}$ das n matrizes. $d.length$ é o tamanho do vetor d.

Saída: Vetor $m[i, j]$ que contém o custo mínimo para obter o produto $M = M_1 \times M_2 \times \ldots \times M_n$.

```
1: m(d.tamanho − 1, d.tamanho − 1); // matriz de custos
2: for i ← 0 to d.tamanho − 2 do
3:    m[i, i] ← 0;
4: end for
5: for rodada ← 1 to d.tamanho − 2 do
6:    for i ← 1 to d.tamanho − 2 − rodada do
7:       j ← i + rodada;
8:       m[i, j] ← ∞;
9:       for k ← 1 to j − 1 do
10:         aux ← m[i, k] + m[k + 1, j] + d[i] × d[k + 1] × d[j + 1];
11:         if aux < m[i, j] then
12:            m[i, j] ← aux;
13:         end if
14:      end for
15:   end for
16: end for
17: retorna m[0, d.length − 2];
```

5.1 Algoritmo Paralelo

Neste trabalho desenvolvemos um algoritmo BSP/CGM para o problema do Produto da Cadeia de Matrizes baseado no paradigma de frente de onda de [4]. A característica e vantagem do paradigma de frente de onda ou sistólico é a modesta demanda de comunicação pelo fato que cada processador comunica-se com poucos outros processadores, tornando o problema perfeito para a computação em grid.

Nós apresentamos um algoritmo BSP/CGM para resolver o problema do Produto da Cadeia de Matrizes com $n+1$ dimensões (n matrizes) usando p processadores, o algoritmo realiza $O(p)$ rodadas de comunicação e tem complexidade de tempo igual a $O(\frac{n^3}{p})$. No algoritmo paralelo, o vetor de custo $m[i, j]$ é dividido em p partes de dimensão $\frac{n}{p} \times n$. Cada processador P_i calcula a parte i, $1 \leq i \leq p$.

Na primeira rodada paralela, o processador P_i computa um bloco de dimensão $n/p \times n/p$ e envia para o processador P_{i-1}, na próxima rodada paralela um novo bloco é calculado usando o bloco recebido e assim sucessivamente. Na primeira rodada paralela todos os processadores estarão trabalhando, e ao fim de cada rodada um processador deixa de

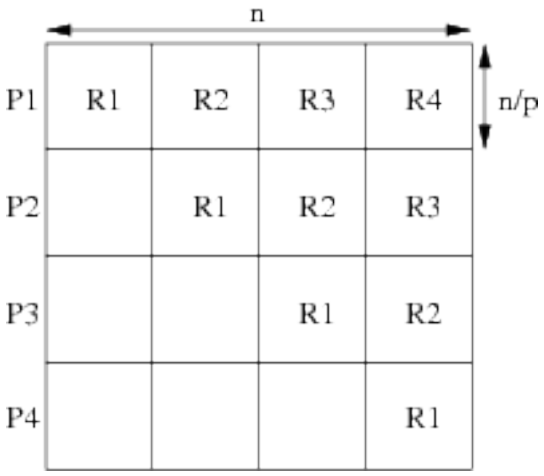


Figura 5.1: Divisão de tarefas

trabalhar, assim no fim da rodada r o processador P_{p-r+1} deixa de trabalhar. A figura 5.1 mostra um exemplo para $p = 4$, onde P_i é o processador e R_i é a rodada paralela.

Na primeira rodada, todos os p processadores estão trabalhando, mas ao fim de cada rodada um processador deixa de trabalhar, isto mostra que a carga não é balanceada (enquanto um processador trabalha muito, outro trabalha pouco), o que é uma característica de muitos algoritmos sistólicos, para compensar esse "problema"ganhamos no número reduzido de comunicação necessária, onde cada processador comunica-se com no máximo outros dois processadores, um para receber dados e outro para enviar dados, assim P_i recebe dados de P_{i+1} e envia dados para P_{i-1}.

O algoritmo paralelo para o problema do Produto da Cadeia de Matrizes é mostrado no Algoritmo 4.

5.2 Resultados Experimentais

Nós rodamos o algoritmo BSP/CGM para o problema do Produto da Cadeia de Matrizes em um *cluster* composto por 12 nós formado por 6 processadores *Intel Pentium* IV de 1,7 GHz e 6 processadores *AMD Athlon* de 1,6 GHz. Os nós estão conectados por um *switch* 1Gb *Fast-Ethernet*. Os dados usados foram gerados aleatoriamente.

O algoritmo para o Produto da Cadeia de Matrizes foi implementado em C++ *Standard*, no *cluster* foi usado a biblioteca LAM-MPI e na *grid* foi usado o *middleware* InteGrade.

A tabela 5.1 e a figura 5.2 mostram o tempo de execução (em segundos) para o algoritmo BSP/CGP do Produto da Cadeia de Matrizes em um *cluster* usando LAM-MPI.

A tabela 5.2 e a figura 5.3 mostram o tempo de execução (em segundos) para o algoritmo BSP/CGP do Produto da Cadeia de Matrizes em uma *grid* usando o *middleware*

Algoritmo 4 Algoritmo Paralelo para o Problema do Produto da Cadeia de Matrizes

Entrada: (1) Vetor d com as dimensões das matrizes (2) O número de processadores p (3) O *rank* i do processador.

Saída: Vetor $m[i, j]$ contendo o custo mínimo para obter o produto da cadeia de matrizes$M$.

1: $bloco \leftarrow (d.tamanho - 1)/p$;
2: $m(bloco, d.tamanho - 1)$; // matriz de custos
3: **for** $rodada \leftarrow 0$ **to** $p - i$ **do**
4: calcular $m[0..block - 1, round * block..((round + 1) * block) - 1]$;
5: **if** $i \neq 1$ **then**
6: enviar bloco calculado para P_{i-1};
7: **end if**
8: **if** $i \neq p$ **then**
9: receber bloco de P_{i+1};
10: **end if**
11: **end for**
12: **if** $i = p$ **then**
13: retorna $m[0, d.length - 2]$;
14: **end if**

n	$p = 1$	$p = 2$	$p = 4$	$p = 8$
256	0.158	0.135	0.095	0.078
512	2.464	2.205	1.704	1.174
1024	32.609	28.283	21.207	14.407
2048	259.346	227.029	175.939	117.089

Tabela 5.1: Tempo de Execução (em segundos) para o Produto da Cadeia de Matrizes em um *cluster* usando LAM-MPI

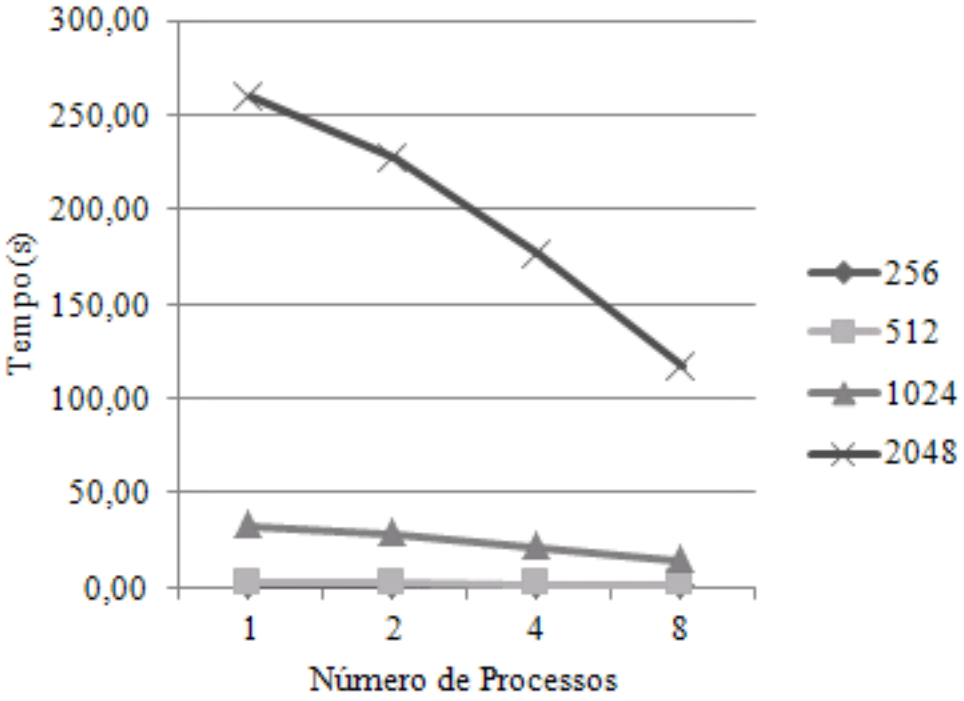


Figura 5.2: Gráfico da Execução (em segundos) para o Produto da Cadeia de Matrizes em um *cluster* usando LAM-MPI

n	$p = 1$	$p = 2$	$p = 4$	$p = 8$
256	0.180	0.165	0.159	-
512	2.756	2.449	2.045	-
1024	38.651	31.210	30.880	-
2048	325.776	276.156	272.819	-

Tabela 5.2: Tempo de Execução (em segundos) para o Produto da Cadeia de Matrizes em uma *grid* usando InteGrade

InteGrade.

A tabela 5.3 mostra uma comparação entre os tempos de execução em um *cluster* usando LAM-MPI e em uma *grid* usando o InteGrade. Coluna I e coluna II mostram os tempos para o *cluster* e para a *grid*, respectivamente. A Figura 5.4 mostra o gráfico correspondente.

Podemos observar que o tempo de execução no *cluster* usando LAM-MPI é ligeiramente melhor do que o tempo na *grid* com InteGrade. Apenas em um caso os tempos são iguais. Os resultados foram considerados promissores e a diferença entre os dois não é substancial. Isto demonstra que o *overhead* do InteGrade é aceitável considerando os benefícios obtidos com a facilidade da computação em *grid*.

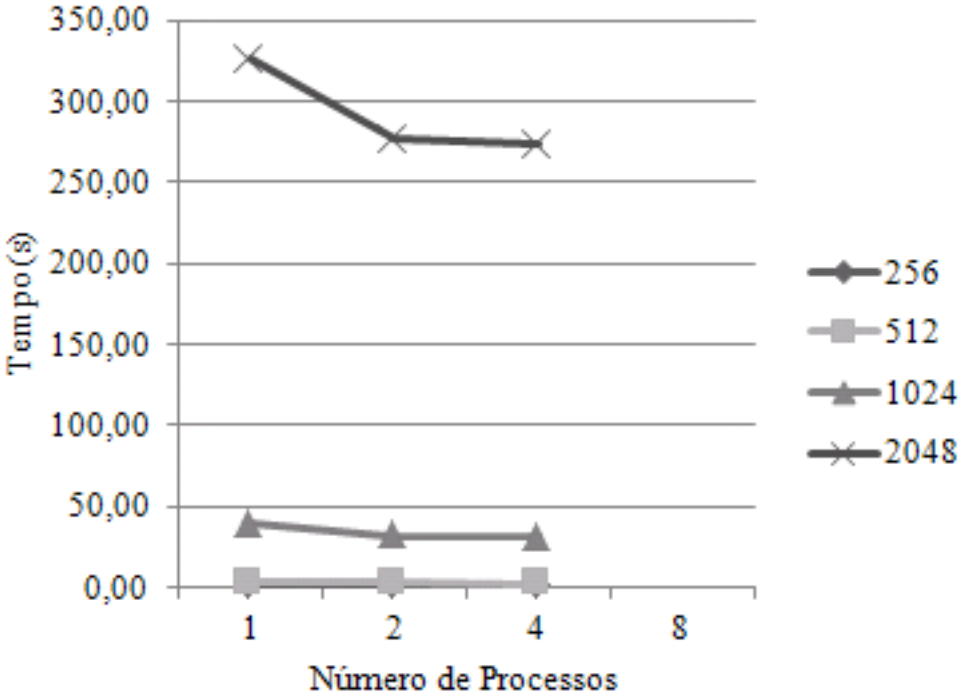


Figura 5.3: Gráfico do Tempo de Execução (em segundos) para o Produto da Cadeia de Matrizes em uma *grid* usando InteGrade e MPI

	$n = 256$		$n = 512$		$n = 1024$		$n = 2048$	
p	I	II	I	II	I	II	I	II
1	0.158	0.180	2.464	2.756	32.609	38.651	259.346	325.776
2	0.135	0.165	2.205	2.449	28.283	31.210	227.029	276.156
4	0.095	0.159	1.704	2.045	21.207	30.880	175.939	272.819
8	0.078	-	1.174	-	14.407	-	117.089	-

Tabela 5.3: Comparativo dos tempos de execução (em segundos) entre um *cluster* (I) e uma *grid* (II) para o Produto da Cadeia de Matrizes

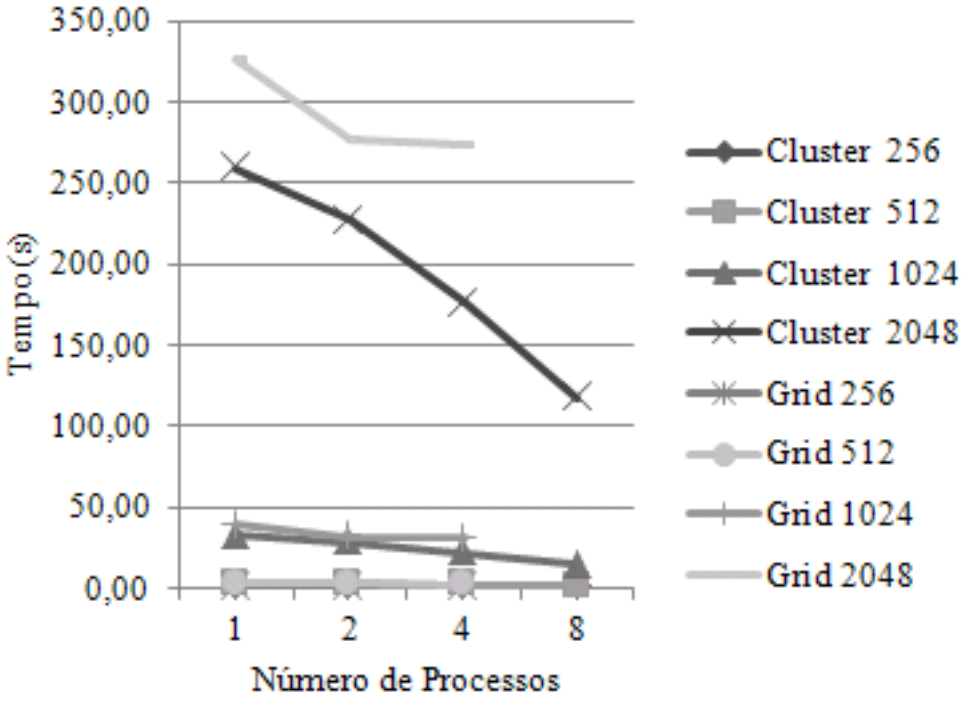


Figura 5.4: Gráfico comparativo dos tempos de execução (em segundos) entre um *cluster* e uma *grid* para o Produto da Cadeia de Matrizes

Capítulo 6

Conclusão

Em nosso trabalho, observamos que o diferença entre a teoria e a prática na computação paralela, que até alguns anos atrás era bem grande, está continuamente diminuindo. Vimos os principais modelos computacional para computação paralela, com destaque para o BSP/CGM, que como mostrado por nossos experimentos mostrou resultados práticos compatíveis com os resultados teóricos, confirmando que o modelo é adequado para máquinas paralelas reais.

Também realizamos uma comparação entre computação em *cluster* e computação em *grid*. Os resultados foram animadores, e mesmo o tempo da execução em *grid* sendo superior a da execução em *cluster* a diferença não foi significativa. O resultado mostra que para problemas resolvidos utilizando o paradigma de frente de onda a computação em *grid* é viável, como nos casos do Alinhamento Local e Produto da Cadeia de Matrizes.

Os resultados obtidos no problema do Alinhamento Local resultou no artigo [12], e mostra uma forma eficiente de resolver um dos problemas mais importantes da biologia computacional. Já os resultados obtidos no problema do Produto da Cadeia de Matrizes resultou no artigo [11] e, mesmo não usando o melhor algoritmo sequencial para o problema [36] pois o mesmo não é facilmente paralelizável, a abordagem usando Programação Dinâmica mostrou bons resultados nas máquinas paralelas.

Propomos para trabalhos futuros o uso do algoritmo de Hu e Sing [36] para o Produto da Cadeia de Matrizes, embora o algoritmo não seja facilmente paralelizável, nossos estudos iniciais mostram que o mesmo pode ser modificado para criar cadeias de matrizes independentes, facilitando assim a distribuição entre os processadores e paralelizando o algoritmo.

Referências Bibliográficas

[1] InteGrade. Disponível em `http://gsd.ime.usp.br/integrade`.

[2] Albert Alexandrov, Mihai F. Ionescu, Klaus E. Schauser, e Chris Scheiman. Loggp: Incorporating long messages into the logp model — one step closer towards a realistic model for parallel computation. Relatório técnico, 1995.

[3] C. E. R. Alves, E. N. Cáceres, e F. Dehne. Parallel Dynamic Programming for Solving the String Editing Problem on a CGM/BSP. In *SPAA '02: Proceedings of the fourteenth annual ACM symposium on Parallel algorithms and architectures*, páginas 275–281. ACM Press, New York, NY, USA, 2002. ISBN 1-58113-529-7.

[4] C. E. R. Alves, E. N. Cáceres, F. Dehne, e S. W. Song. A Parallel Wavefront Algorithm for Efficient Biological Sequence Comparison. In *Proceedings ICCSA 2003*, volume 2668 de *Lecture Notes in Computer Science*, páginas 249–258. Springer, 2003.

[5] C. E. R. Alves, E. N. Cáceres, F. Dehne, e S. W. Song. A CGM/BSP Parallel Similarity Algorithm, 2002.

[6] A. Apostolico, M. J. Atallah, L. L. Larmore, e S. MacFaddin. Efficient Parallel Algorithms for String Editing and Related Problems. *SIAM J. Comput.*, 19(5):968–988, 1990.

[7] Gianfranco Bilardi, Kieran T. Herley, Andrea Pietracaprina, Geppino Pucci, e Paul Spirakis. Bsp vs logp. In *SPAA '96: Proceedings of the eighth annual ACM symposium on Parallel algorithms and architectures*, páginas 25–32. ACM, 1996. ISBN 0-89791-809-6.

[8] R. Buyya. The design of paras microkernel. Relatório técnico, 1995.

[9] 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)*, páginas 81–88. Ouro Preto, Brazil, 2006.

[10] E. N. Cáceres e C. Nishibe. 0-1 Knapsack Problem: BSP/CGM Algorithm and Implementation. In *IASTED PDCS*, páginas 331–335. 2005.

[11] E. N. Cáceres, Henrique Mongelli, Leonardo Loureiro, Christiane Nishibe, e Siang Wun Song. A Parallel Chain Matrix Product Algorithm on the InteGrade Grid. *Grid and e-Science in Asia Pacific Region (HPC Asia 2009)*, páginas 304–311, 2009.

[12] E. N. Cáceres, Henrique Mongelli, Leonardo Loureiro, Christiane Nishibe, e Siang Wun Song. Performance results of running parallel applications on the integrade. *Concurrency and Computation: Practice and Experience*, 22(3):375–393, 2010.

[13] Lei Chai, Qi Gao, e Dhabaleswar K. Panda. Understanding the impact of multi-core architecture in cluster computing: A case study with intel dual-core system. In *CCGRID '07: Proceedings of the Seventh IEEE International Symposium on Cluster Computing and the Grid*, páginas 471–478. IEEE Computer Society, 2007. ISBN 0-7695-2833-3.

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

[15] David Culler, Richard Karp, David Patterson, Abhijit Sahay, Klaus Erik Schauser, Eunice Santos, Ramesh Subramonian, e Thorsten von Eicken. Logp: towards a realistic model of parallel computation. In *PPOPP '93: Proceedings of the fourth ACM SIGPLAN symposium on Principles and practice of parallel programming*, páginas 1–12. ACM, 1993. ISBN 0-89791-589-5.

[16] David Culler, Lok Tin Liu, Richard P. Martin, e Chad Yoshikawa. Logp performance assessment of fast network interfaces, 1996.

[17] Artur Czumaj. Parallel algorithm for the matrix chain product and the optimal triangulation problems (extended abstract). In *STACS '93: Proceedings of the 10th Annual Symposium on Theoretical Aspects of Computer Science*, páginas 294–305. Springer-Verlag, 1993. ISBN 3-540-56503-5.

[18] Artur Czumaj. Very fast approximation of the matrix chain product problem. *J. Algorithms*, 21(1):71–79, 1996. ISSN 0196-6774.

[19] L. Dagum e R. Menon. OpenMP: An Industry-Standard API for Shared-memory Programming. *IEEE Computational Science Engineering*, 5(1):46–55, 1998.

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

[21] Frank Dehne, Andreas Fabri, e Andrew Rau-Chaplin. Scalable parallel geometric algorithms for coarse grained multicomputers. In *SCG '93: Proceedings of the ninth annual symposium on Computational geometry*, páginas 298–307. ACM, 1993. ISBN 0-89791-582-8.

[22] Z. Galil e K. Park. Parallel Dynamic Programming. Relatório Técnico CUCS-040-91, Columbia Univ., Computer Science Dept., New York, NY, 10027, USA, 1991.

[23] Z. Galil e K. Park. Dynamic Programming with Convexity, Concavity and Sparsity. *Theoretical Computer Science*, 92:49–76, 1992.

[24] Thierry Garcia, Jean-Frederic Myoupo, David Seme, Universite Picardie, e Jules Verne. A coarse-grained multicomputer algorithm for the longest common subsequence problem. In *11-th Euromicro Conference on Parallel Distributed and Network based Processing (PDP'03*, páginas 349–356. 2003.

[25] Assefaw Hadish Gebremedhin, Mohamed Essaïdi, Isabelle Guérin Lassous, Jens Gustedt, e Jan Arne Telle. Pro: a model for the design and analysis of efficient and scalable parallel algorithms. *Nordic J. of Computing*, 13(4):215–239, 2006. ISSN 1236-6064.

[26] M. Gengler. An Introduction to Parallel Dynamic Programming. In *Solving Combinatorial Optimization Problems in Parallel - Methods and Techniques*, volume 1054, páginas 87–114. Springer-Verlag, London, UK, 1996. ISBN 3-540-61043-X.

[27] Phillip B. Gibbons, Yossi Matias, e Vijaya Ramachandran. Efficient low-contention parallel algorithms. In *SPAA '94: Proceedings of the sixth annual ACM symposium on Parallel algorithms and architectures*, páginas 236–247. ACM, 1994. ISBN 0-89791-671-9.

[28] Sadashiva S. Godbole. On efficient computation of matrix chain products. *IEEE Trans. Comput.*, 22(9):864–866, 1973. ISSN 0018-9340.

[29] 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*, páginas 1–6. LNCC, 2005.

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

[31] Andrei Goldchleger, Fabio Kon, Alfredo Goldman, Marcelo Finger, e Germano Capistrano Bezerra. Integrade object-oriented grid middleware leveraging the idle computing power of desktop machines: Research articles. *Concurr. Comput. : Pract. Exper.*, 16(5):449–459, 2004. ISSN 1532-0626.

[32] Mark Goudreau, Kevin Lang, Satish Rao, Torsten Suel, e Thanasis Tsantilas. Towards efficiency and portability: programming with the bsp model. In *SPAA '96: Proceedings of the eighth annual ACM symposium on Parallel algorithms and architectures*, páginas 1–12. ACM, 1996. ISBN 0-89791-809-6.

[33] P. A. V. Hall e G. R. Dowling. Approximate String Matching. *ACM Comput. Surv.*, 12(4):381–402, 1980. ISSN 0360-0300.

[34] Jonathan M. D. Hill, Bill McColl, Dan C. Stefanescu, Mark W. Goudreau, Kevin Lang, Satish B. Rao, Torsten Suel, Thanasis Tsantilas, e Rob H. Bisseling. Bsplib: The bsp programming library. *Parallel Comput.*, 24(14):1947–1980, 1998. ISSN 0167-8191.

[35] Jonathan M.D. Hill e David B. Skillicorn. Lessons learned from implementing bsp. *Future Gener. Comput. Syst.*, 13(4-5):327–335, 1998. ISSN 0167-739X.

[36] T. C. Hu e M. T. Shing. Computation of matrix chain products: Part i, part ii. Relatório técnico, Stanford, CA, USA, 1981.

[37] J. W. Hunt e T. Szymansky. An Algorithm for Differential File Comparison. *Comm. ACM*, (20):350–353, 1977.

[38] Fabio Kon e Andrei Goldchleger. *Grades Computacionais: Conceitos Fundamentais e Casos Concretos*, volume 1, capítulo 2, páginas 55–104. 2008.

[39] Barry L. Kurtz, Chinhyun Kim, e Jamal Alsabbagh. Parallel computing in the undergraduate curriculum. In *SIGCSE '98: Proceedings of the twenty-ninth SIGCSE technical symposium on Computer science education*, páginas 212–216. ACM, 1998. ISBN 0-89791-994-7.

[40] Keqin Li. Analysis of parallel algorithms for matrix chain product and matrix powers on distributed memory systems. *IEEE Trans. Parallel Distrib. Syst.*, 18(7):865–878, 2007. ISSN 1045-9219.

[41] P. D. MacKenzie. A lower bound for the qrqw pram. In *SPDP '95: Proceedings of the 7th IEEE Symposium on Parallel and Distributeed Processing*, página 231. IEEE Computer Society, 1995. ISBN 0-8186-7195-5.

[42] B. Mohr. Introduction to Parallel Computing. *Computational Nanoscience: Do It Yourself!*, 5, 2006.

[43] 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)*, páginas 155–162. October 24–27, 2007.

[44] Pat Morin. Coarse grained parallel computing on heterogeneous systems. In *SAC '98: Proceedings of the 1998 ACM symposium on Applied Computing*, páginas 628–634. ACM, 1998. ISBN 0-89791-969-6.

[45] MA. Needham. *CORBA v3.0 specification*. OMG Document, 2002.

[46] S. B. Needleman e C. D. Wunsch. A General Method Applicable to the Search for Similarities in the Amino Acid Sequence of Two Proteins. *J. Mol. Bio.*, (48):443–453, 1970.

[47] S. B. Needleman e C. D. Wunsch. A general method applicable to the search for similarities in the amino acid sequence of two proteins. *Journal of Molecular Biology*, páginas 443–453, 1970.

[48] Rolf Niedermeier e Peter Rossmanith. Pram's towards realistic parallelism: Bram's. In *FCT '95: Proceedings of the 10th International Symposium on Fundamentals of Computation Theory*, páginas 363–373. Springer-Verlag, 1995. ISBN 3-540-60249-6.

[49] T. Rognes e E. Eeberg. Six-fold speed-up of Smith-Waterman sequence database searches using parallel processing on common microprocessors. *Bioinformatics*, páginas 699–706, 2000.

[50] P. H. Sellers. The Theory and Computation of Evolutionary Distances: Pattern Recognition. *J. Algorithms*, (1):359–373, 1980.

[51] J. Setubal e J. Meidanis. *Introduction to Computational Molecular Biology*. PWS Publishing Company, 1997.

[52] Daniel Paranhos Da Silva, Walfredo Cirne, Francisco Vilar Brasileiro, e Campina Grande. Trading cycles for information: Using replication to schedule bag-of-tasks applications on computational grids. In *Applications on Computational Grids, in Proc of Euro-Par 2003*, páginas 169–180. 2003.

[53] D. B. Skillicorn, Jonathan M. D. Hill, e W. F. McColl. Questions and answers about bsp. *Scientific Programming*, 6(3):249–274, 1997.

[54] T. F. Smith e M. S. Waterman. Identification of Common Molecular Subsequences. *J. Mol. Bio.*, (147):195–197, 1981.

[55] T. F. Smith e M. S. Waterman. Identification of common molecular subsequences. *Journal of Molecular Biology*, páginas 195–197, 1981.

[56] Leslie G. Valiant. A bridging model for parallel computation. *Commun. ACM*, 33(8):103–111, 1990. ISSN 0001-0782.

[57] Uzi Vishkin. Thinking in parallel: Some basic dataparallel algorithms and techniques. *College Park, MD*, 2007, 1993.

[58] Xingzhi Wen e Uzi Vishkin. Fpga-based prototype of a pram-on-chip processor. In *CF '08: Proceedings of the 5th conference on Computing frontiers*, páginas 55–66. ACM, 2008. ISBN 978-1-60558-077-7.

[59] David A. Wood e Mark D. Hill. Cost-effective parallel computing. *Computer*, 28(2):69–72, 1995. ISSN 0018-9162.

[60] S. Wu e U. Manber. Fast Text Searching Allowing Errors. *Comm. ACM*, (35):83–91, 1992.