# Um Estudo Sobre Modelos de Computação Paralela

Ronaldo Parente de Menezes

**Tese defendida e aprovada em,** 29 **de** 06 **de 199** 5

**Pela Banca Examinadora composta pelos Profs. Drs.**

**Prof(a). Dr(a).** ARTHUR JOÃO CATTO

**Prof(a). Dr(a).** SIANG WUN SONG

**Prof(a). Dr(a).** JOÃO CARLOS SETÚBAL

# Um Estudo Sobre Modelos de Computação Paralela

Este exemplar corresponde à redação final da tese devidamente corrigida e defendida pelo Sr. Ronaldo Parente de Menezes e aprovada pela Comissão Julgadora.

Campinas, 29 de Junho de 1995.

João Carlos Setubal OK
*Orientador*

Dissertação apresentada ao Instituto de Matemática, Estatística e Ciência da Computação, UNICAMP, como requisito parcial para a obtenção do título de Mestre em Ciência da Computação.

# Um Estudo Sobre Modelos de Computação Paralela[1]

Ronaldo Parente de Menezes[2]

Departamento de Ciência da Computação
IMECC – UNICAMP

Banca Examinadora:

- João Carlos Setubal(Orientador)[3]
- Arthur João Catto[4]
- Siang Wun Song[5]
- João Meidanis[6]

[1]Dissertação apresentada ao Instituto de Matemática, Estatística e Ciência da Computação da UNICAMP, como requisito parcial para a obtenção do título de Mestre em Ciência da Computação.

[2]O autor é Bacharel em Informática pela Universidade de Fortaleza - UNIFOR

[3]Professor do Departamento de Ciência da Computação - IMECC - UNICAMP.

[4]Professor do Departamento de Ciência da Computação - IMECC - UNICAMP.

[5]Professor do Departamento de Ciência da Computação - IME - USP.

[6]Professor do Departamento de Ciência da Computação - IMECC - UNICAMP.

# Dedicatória

Aos meus pais, meus irmãos e a minha esposa.

# Agradecimentos


- Em primeiro lugar a quem eu chamo de Deus, pela sua iluminação nos momentos mais difíceis da minha vida em Campinas, mesmo sabendo que em certas ocasiões eu não fiz por merecer;
- A minha esposa, inspiração maior da minha vida, que mesmo sabendo dos sacrifícios que iriamos ter que fazer, sempre me apoiou;
- Ao meu orientador, Prof. Dr. João Carlos Setubal, que sempre me ajudou, principalmente na minha falta de experiência em trabalhos científicos;
- Aos meus pais, Flávio e Maria do Carmo, que sempre acreditaram em mim e que me proporcionaram, desde a infância, os melhores ambientes de estudo;
- As minhas irmãs, Flávima e Henilde, cujo os momentos de descontração ao telefone foram de fundamental importância para que eu conseguisse superar este tempo longe do convívio familiar;
- Aos meus irmãos Robson e Henilton cujos momentos de lazer, respectivamente na Formula 1 e Free Jazz Festival, me fizeram lembrar os vários momentos de lazer que tínhamos em Fortaleza;
- Ao meu irmão Flávio Augusto que sempre foi meu modelo de ser humano, principalmente pela dedicação a seus objetivos;
- Aos meus sobrinhos Juliana, Ana Rafaela, Fábio Cristino, Marina e Giovanna Cristina que fazem a alegria da família e conseqüentemente a minha;
- A minha Tia e Madrinha Porcina cujo amor para com os outros sempre me mostrou o que significa um ser humano;
- Aos meus amigos de Fortaleza: Cláudio, Eduardo, Francílio, Jorge, Márcio Róger e Sílvio Mauro que sempre apoiaram minha decisão de vir para o mestrado em Campinas.

- Aos meus amigos de mestrado (do Ceará): Carlos Roberto, Flávio, Marcus Vinícius, Pedro Rafael e Victor pelas infindáveis conversas;
- Aos meus amigos do DCC pelo ambiente propício para o desenvolvimento deste trabalho;
- Em especial ao meu amigo Victor, pelas suas valorosas opiniões e pelo ótimo convívio durante a conclusão desta tese;
- Em especial ao meu amigo Jaime Cohen pelos várias conversas sobre Teoria da Computação.
- Novamente a minha esposa por suas ótimas opiniões quanto a revisão ortográfica desta;
- A todos os professores do DCC, principalmente àqueles com que convivi durante as cadeiras do curso: Profa. Célia Picinnin (Teoria dos Grafos), Profa. Cláudia Bauzer (Banco de Dados I), Prof. João Carlos Setubal (Teoria da Computação), Prof. Marcus Poggi (Tópicos Especiais em Computação), Prof. Mário Curtis (Arquitetura de Computadores), Prof. Pedro Rezende (Curso de Verão) e Prof. Tomasz Kowaltowski (Linguagens de Programação);
- A todos funcionários do DCC cuja eficiência é exemplo para todos nós;

# Resumo

Modelos de Computação são uma ferramenta muito importante para um bom desenvolvimento de algoritmos. Em geral, eles visam facilitar o trabalho de projetistas abstraindo diversos fatores existentes nas máquinas reais.

Em computação paralela, a necessidade de um modelo é extrema devido a grande variedade de arquiteturas. O surgimento de um modelo de computação paralela poderia impulsionar ainda mais o crescimento da área que já é bastante acentuado, devido a limitações físicas existentes em computadores seqüenciais.

Nesta dissertação fazemos um estudo de modelos de computação paralela sob o ponto de vista de projeto de algoritmos e com enfoque na computação paralela derivada da arquitetura de *von Neumann*. Para tanto, começamos por estudar um conjunto de máquinas paralelas para que suas diferenças fiquem claras. Escolhemos as máquinas paralelas mais conhecidas, ou mais difundidas, como: CM-2, *Sequent Symmetry*, MasPar MP-1, CM-5, entre outras.

Após este estudo de máquinas, partimos diretamente para os modelos de computação paralela. Escolhemos três como base. Tais modelos apresentam características bem distintas quanto a simplicidade e realismo.

Os modelos estudados são PRAM, BSP [Val90] e LogP [CKP+93]. Muitos defendem que continuemos usando o modelo PRAM , pois este, apesar de ser muito abstrato, facilita bastante o trabalho dos projetistas. A proposta do modelo BSP é um pouco mais ousada pois Valiant tenta, com seu modelo, influenciar as áreas de *hardware* e *software* da mesma forma que a arquitetura *von Neumann* fez com a computação seqüencial. Já a proposta do modelo LogP é bastante imediatista, visto que tenta resolver o problema atual de dificuldade de projeto de algoritmos.

Para que pudéssemos avaliar um modelo sob o ponto de vista de projeto de algoritmos, fizemos um estudo de casos com os problemas de Transformada de Fourier e Eliminação de Gauss. Com este estudo de casos pudemos avaliar quão fácil ou difícil é projetar algoritmos em cada um dos modelos.

# Abstract

Models of Computation are one of most important tools in algorithm design. With these models, the work of an algorithm designer becomes easier, because these models leave out many characteristics of real machines.

In parallel computing there is a great need for a general model, because we have many different parallel machines. The advent of a parallel computing model could make the area grow more than it is already growing.

In this dissertation we study some parallel computing models. First we take a look at a representative set of parallel machines, in order to learn the differences between each architecture. Our set of machines contains some of the most important commercial machines such as: CM-2, Sequent Symmetry, MasPar MP-1 and CM-5. After this, we study the models themselves.

The models chosen were: PRAM, BSP [Val90] and LogP [CKP+93]. Many researchers argue that the PRAM is the best model for algorithm design although it is not realistic. The proposal of the BSP model is bold, since it also seeks to influence parallel architecture design. The proposal of LogP model although similar to the BSP, does not require parallel machines to have synchronization mechanisms. This makes LogP the most realistic but also the most difficult model to use.

We evaluate these models based on the problems of Fourier Transform and Gaussian Elimination. After this study we made an evaluation of the three models.

# Conteúdo

# Lista de Tabelas

# Lista de Figuras

# Capítulo 1

# Introdução

*"For unto us a child is born,*
*unto us a Son is given,*
*and the government shall be upon His shoulder,*
*and His name shall be called:*
*Wonderful, Counsellor, The Mighty God,*
*The Everlasting Father, The Prince of Peace !"*

*ISIAH IX,6*

O mundo está em um processo intenso de evolução. Diversas áreas crescem a um ritmo assustador e cada dia se necessita mais de avanços tecnológicos. A computação possui hoje um papel de muita importância neste contexto, já que a mente humana trabalha muito rapidamente e conseqüentemente temos que possuir tecnologia suficiente para acompanhar esta velocidade.

Atualmente é inconcebível se fazer pesquisa sem o apoio de um computador, que a cada dia dispõe de mais e mais recursos. Não sabemos porém, até que ponto a tecnologia destes computadores suprirá as necessidades existentes. Desta forma temos que pensar em soluções para este problema antes que ele se torne crítico. O nosso conceito de pesquisa é este: antecipar problemas e resolvê-los antes que ocorram.

O conceito de paralelismo é muito antigo. Desde os primórdios o homem já fazia trabalhos em paralelo. Atualmente, podemos afirmar que uma grande parte dos trabalhos existentes são executados em paralelo. Qualquer trabalho em grupo, por exemplo, pode ser visto como uma tarefa paralela, já que se tem um problema grande a ser resolvido e várias pessoas trabalhando nele.

Por que então não utilizamos este conceito tão difundido em computação ? Pensando desta maneira, diversas propostas de arquiteturas com tecnologia de paralelismo surgiram na última década, dando início a uma nova subárea da computação a qual chamaremos simplesmente de **Computação Paralela**.

Formada por um conjunto muito grande e distinto de arquiteturas, onde cada uma destas possui poucas máquinas como representantes, a computação paralela atual é uma área bastante diversificada e ainda pouco explorada quando comparada com a computação seqüencial. Além disso, o rápido crescimento em desempenho das arquiteturas seqüenciais tem tornado a computação paralela uma área mais esquecida ainda, já que, em geral, o desempenho das máquinas seqüenciais ainda é satisfatório.

Mesmo assim, existem diversas propostas que nos mostram os benefícios da utilização da computação paralela. Em última análise sabemos que, cedo ou tarde, a velocidade da computação seqüencial atingirá seus limites físicos; quando este momento chegar, um aumento no desempenho computacional só será possível através do paralelismo .

Nesta dissertação tentamos esclarecer os principais fatores que tornam a computação paralela uma área um pouco nebulosa para grande parte dos usuários. Tenderemos a mostrar que acreditamos que a melhor solução para a computação paralela é o surgimento de um modelo de computação padrão, como também é defendido por Valiant [Val90].

Esta dissertação está dividida em nove capítulos, dos quais este é o primeiro.

No capítulo 2 definimos modelos de computação de uma forma geral e situamos a computação seqüencial atual no contexto deste trabalho. Em seguida mostramos o porquê do sucesso da computação seqüencial, além de pontos que podem ser utilizados para melhorar a generalidade da computação paralela.

No capítulo 3 entramos na computação paralela propriamente dita onde mostramos algumas classificações existentes e descrevemos algumas máquinas paralelas mais difundidas. Usando as classificações apresentadas mostramos que tais máquinas possuem arquiteturas bastante distintas.

No capítulo 4 mostramos a possibilidade de a computação paralela vir a substituir a computação seqüencial naturalmente. Logo depois definimos modelos de computação paralela, mostrando sua necessidade. Em seguida descrevemos o modelo PRAM, por este ser o modelo de computação paralela mais difundido, de uma forma bem detalhada mostrando onde este falha e também mostrando suas boas características.

No capítulo 5 descrevemos sucintamente algumas extensões do modelo PRAM que tentam torná-lo mais realista. Descrevemos extensões de tal sorte que todos os principais problemas do PRAM sejam abordados. Algumas destas extensões são bastante recentes, como o caso do QRQW PRAM [GMR94c], nos mostrando que muitos acreditam na viabilidade deste tipo de solução.

Nos capítulos 6 e 7 descrevemos duas novas propostas de modelos de computação paralela que representam bem o espectro existente de novos modelos que partem de uma linha diferente do PRAM. No capítulo 6 descrevemos o modelo BSP [Val90] que, a partir do sucesso do modelo RAM e da arquitetura *von Neumann*, se propõe a ser um modelo ponte ligando as áreas de *hardware* e *software*. Para tanto este modelo supõe a existência de uma arquitetura especial que, de certa forma, faz o papel da arquitetura *von Neumann* na

computação paralela. No capítulo 7 descrevemos um modelo de mais baixo nível, chamado LogP [CKP+93]. Este modelo diferentemente dos outros tenta resolver o problema atual modelando um tipo de arquitetura especial dentre os apresentados no capítulo 2.

No capítulo 8 fazemos um estudo dos modelos apresentados, sob o ponto de vista de projeto e análise de algoritmos, abordando dois problemas fundamentais em computação: Eliminação de Gauss e Transformada Rápida de Fourier.

No capítulo 9 fazemos uma conclusão geral do trabalho onde comparamos os modelos PRAM, BSP, e LogP entre si sob o ponto de vista de projeto de algoritmos. Em seguida mostramos as contribuições do trabalho e quais suas possíveis extensões.

# Capítulo 2

# Modelos de Computação

## 2.1 Caracterização

Um sistema de computação é uma estrutura muito complexa, tornando difícil, para grande parte das pessoas, o desenvolvimento de programas que considerem todos os detalhes nele existentes. Devido a esta dificuldade, surgiram abstrações, as quais chamamos **Modelos de Computação**. Tais modelos consideram apenas as características mais importantes dos sistemas, objetivando facilitar o desenvolvimento de programas (vide figura 2.1).

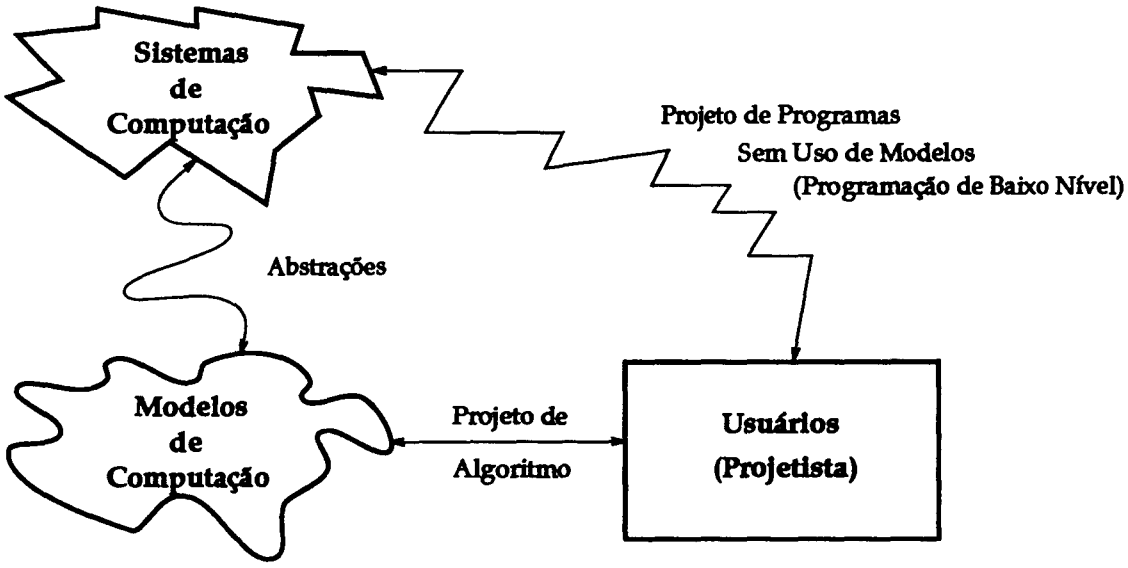


Figura 2.1: Modelo de computação como meio de interação entre usuário e sistemas de computação.

Mais formalmente temos:

*Um modelo de computação caracteriza as operações primitivas que podem ser executadas nos sistemas de computação e seus respectivos custos* [PS85].

Pela figura 2.1, podemos notar que um modelo pode facilitar bastante o trabalho de um projetista de algoritmos. Sem um modelo o projetista teria que desenvolver seus programas diretamente para os sistemas de computação seguindo o caminho complexo representado na figura citada. Já com a existência de um modelo de computação o trabalho se torna bem mais fácil já que o trabalho teoricamente mais complexo, o de abstração da arquitetura, já foi realizado durante a construção do modelo.

A elaboração de um modelo de computação não é uma tarefa simples. Muitas são as características desejáveis, mas pelo menos duas podem ser consideradas fundamentais:

- **Simplicidade:** Por ser uma abstração de sistemas de computação, os modelos devem ser mais simples que tais sistemas. Esta simplicidade viabiliza o desenvolvimento de algoritmos por projetistas que não sejam especialistas nos sistemas de computação.

- **Fidelidade:** Um modelo de computação pode ser dito fiel se atenta para os pontos fundamentais dos sistemas de computação e se o comportamento de algoritmos desenvolvidos no modelo for semelhante ao comportamento da implementação dos mesmos algoritmos em um sistema real.

Um balanceamento entre simplicidade e fidelidade pode, em certos casos, caracterizar um bom modelo. Outras características importantes são:

- **Unicidade:** O número de modelos existentes voltados para um conjunto de sistemas de computação deve ser pequeno ou preferencialmente único, pois isto permite o surgimento de uma base de pensamento comum, a qual teria uma grande chance de ser difundida.

- **Independência:** Um modelo não deve estar inteiramente ligado a um tipo de arquitetura em particular nem à semântica de uma determinada linguagem; o modelo deve ser o mais geral possível, algo que seja independente de *hardware* e *software* abstraindo características importantes de ambos os lados. Desta forma os algoritmos desenvolvidos terão um tempo de vida maior, pois não se tornariam inadequados com o surgimento de novas tecnologias.

- **Facilidade de Análise:** Um modelo de computação deve permitir o uso de alguma forma de avaliação de complexidade de algoritmos.

## 2.2 O Caso Seqüencial

Quando falamos em modelos de computação seqüencial, há dois modelos principais: a Máquina de Turing e o modelo RAM (*Random Access Machine*). Descrevemos neste trabalho apenas o segundo, pois o modelo de Máquina de Turing, apesar ser extremamente geral, não nos permite fazer uma análise de complexidade de uma maneira fácil devido ao fato de este ser um modelo extremamente simples. Ao trabalharmos com o modelo RAM, que é mais complexo, podemos analisar mais facilmente um algoritmo. Além disso, sabemos que Máquina de Turing e RAM possuem poder computacional equivalente [AHU74].

O modelo RAM consiste basicamente de dois componentes: um processador e uma memória ilimitada que pode ser acessada por este processador. Neste modelo temos as seguintes operações primitivas disponíveis a custo unitário:

1. operações aritméticas $(+, -, \times, \div)$;
2. comparações entre dois números $(=, >, <, \neq, \geq, \leq)$;
3. operações de acesso à memória.

O modelo RAM obteve êxito na sua tarefa conseguindo se tornar um **Modelo Ponte**[1] em computação seqüencial. Sua proximidade com a arquitetura *von Neumann*, fez com que os usuários de computação seqüencial o utilizassem na tarefa de desenvolvimento de algoritmos. Sua estabilidade permitiu o desenvolvimento de uma base de pensamento na qual tanto empresas de *hardware* quanto de *software* puderam confiar. Também permitiu todo um desenvolvimento de uma teoria de complexidade, possibilitando se saber mais facilmente quais problemas são tratáveis ou não.

As previsões de desempenho no modelo RAM não são muito precisas devido a diferenças existentes entre modelo e máquina. Estas previsões de desempenho, porém, possuem boa precisão relativa, isto é, se um algoritmo $A$ for assintoticamente melhor que um outro algoritmo $B$ no modelo RAM, o algoritmo $A$ implementado em uma máquina seqüencial será, muito provavelmente, melhor que $B$ implementado na mesma máquina para entradas suficientemente grandes.

Apesar de todo este sucesso, o modelo RAM possui alguns problemas que podem comprometer seu uso na prática:

- **Memória Ilimitada:** O modelo RAM supõe que a memória é suficientemente grande de modo a todos os dados necessários a uma operação se encontrarem disponíveis em memória principal. Esta suposição nem sempre é verdadeira quando consideramos máquinas seqüenciais reais. Suponha como ilustração uma ordenação

[1] *Bridging Model*, conceito introduzido por Valiant [Val90]

de 100 milhões de elementos onde cada elemento ocupa 4 bytes. Neste caso nem toda máquina conseguirá manter uma quantidade tão grande de dados em memória principal, pois seria necessário em torno de 400 Mbytes de memória para tal armazenamento. Desta forma a operação em questão sofreria uma degradação no seu tempo de execução, já que dados armazenados em memórias secundárias possuem tempo de acesso maior. Note que máquinas seqüenciais reais mantêm seus dados armazenados em uma hierarquia de memória de diferentes velocidades.

- **Custo Unitário das Instruções:** Para o modelo RAM, todas as operações primitivas custam uma unidade de tempo. Esta outra suposição também não é realista pois sabemos que em máquinas seqüenciais reais as operações têm custos diferentes. Como ilustração, observemos que mesmo operações simples como adição e multiplicação possuem custos diferenciados. Desta forma, a diferença de custos existente entre uma leitura de um dado e uma soma de dois números tende a ser muito maior. Tal diferença não suposta pelo modelo RAM pode ocasionar que um algoritmo $A$ seja mais eficiente do que um algoritmo $B$ de acordo com a análise feita sobre o modelo RAM e que na prática ocorra o inverso.

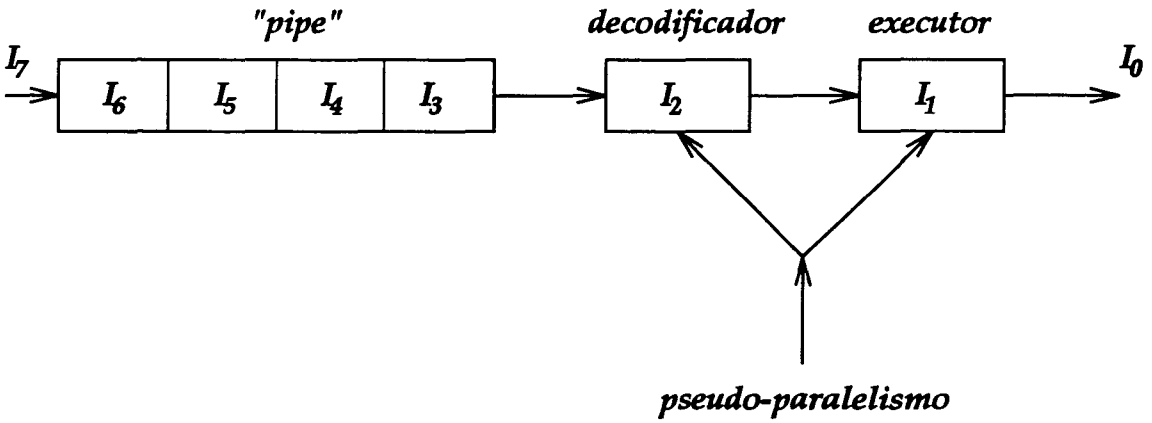


Figura 2.2: Estrutura simplificada de uma execução através do uso de *pipelining*. Execução não atômica.

- **Igualdade de Tempo de Acesso:** Além de a memória ser ilimitada, o modelo RAM supõe que o tempo de acesso a qualquer dado é o mesmo, não importando sua posição na memória. Isto seria até aceitável se não tivéssemos a memória das máquinas reais organizada em uma hierarquia. Tal hierarquia não só apresenta diferenças de tempo de acesso de um nível para outro, como também dentro de um mesmo nível, dependendo da localização do dado. Um bom exemplo deste caso são os acessos a discos rígidos.

- **Palavra de Memória Ilimitada:** O modelo RAM supõe que a palavra de memória é grande o suficiente para manipular dados de qualquer tamanho. Já em uma máquina seqüencial real isso não acorre; a palavra de memória é limitada e caso o dado não consiga ser manipulado em uma única palavra, ele terá de ser manipulado por duas ou mais. A isto associa-se um custo que não é considerado no modelo RAM básico. O modelo RAM de **custo logarítmico** [AHU74], por outro lado, considera tal custo.

- **Pipelining:** Esquema de execução de instruções não convencional característico da maioria das máquinas seqüenciais o qual permite que uma execução se processe mais rapidamente. Tal aceleração pode ser feita já que o processo de execução não é atômico; geralmente é dividido em subpartes como decodificação e execução. Esta divisão possibilita que enquanto uma instrução esteja sendo executada outra já possa estar sendo decodificada (vide figura 2.2). Novamente o modelo RAM não considera este tipo de operação.

# Capítulo 3

# Arquiteturas Paralelas

## 3.1 Classificação das Arquiteturas Paralelas

As máquinas paralelas começaram a surgir com o objetivo de resolver algumas aplicações científicas de maior porte. As máquinas eram projetadas com a finalidade de resolver um determinado problema da forma mais rápida possível sem preocupações com padronização entre arquiteturas; o conceito da arquitetura proposta refletia exatamente o problema a resolver.

Apesar das diferenças existentes entre as arquiteturas paralelas, podemos discernir algumas características comuns que são de grande importância para o nosso trabalho, pois podemos olhar este diversificado mundo em partes. A seguir expomos uma caracterização quanto a organização da memória global. Antes porém precisamos das seguintes definições:

**Definição 3.1** *Latência de Memória é o tempo que um processador espera para que um dado requisitado que não esteja em sua memória local fique disponível.*

**Definição 3.2** *Escalabilidade é a capacidade que uma máquina possui de melhorar o seu desempenho à medida que o número de processadores é aumentado.*

A caracterização por **Memória Distribuída** supõe que cada processador possua uma parte da memória global que lhe é acessível, e onde a comunicação entre eles seja realizada através de troca de mensagens. A ligação entre os processadores é representada através de uma rede de interconexão. Esta rede pode, em geral, ser visualizada como um grafo onde cada vértice representa um par processador-memória e onde as arestas representam as ligações diretas entre estes pares. A latência de memória está intimamente ligada ao diâmetro da topologia da rede, que, em geral, é expresso em relação ao número de processadores, $n$ (número de vértices do grafo). Existem diversas topologias (vide figura 3.1), dentre as mais conhecidas temos:

- **Linear:** Topologia muito simples e pouco usada, pois seu diâmetro é dado pelo tamanho da estrutura, que é linear, $O(n)$.
- **Completa:** Topologia também pouco usada, pois seu custo de construção é muito alto, principalmente quando temos muitos processadores. Em contraste, seu diâmetro é constante, $O(1)$.
- **Malha:** O uso desta topologia já é mais difundido, pois possui uma boa tolerância a falhas (devido ao grande número de caminhos alternativos entre dois processadores) e diâmetro $O(\sqrt{n})$.

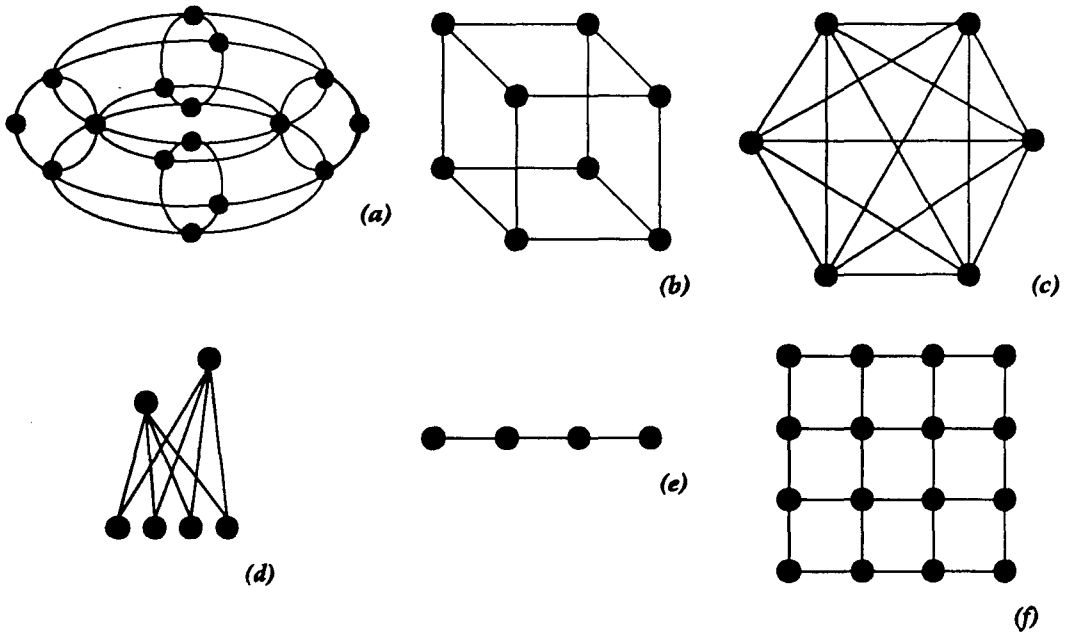


Figura 3.1: Algumas topologias de rede de interconexão: *(a)* Toro, *(b)* Hipercubo 3-D, *(c)* Completa, *(d)* Árvore gorda, *(e)* Linear e *(f)* Malha.

- **Hipercubo:** Topologia bastante usada devido ao seu baixo diâmetro em relação a outras topologias, $O(log\ n)$.
- **Toro:** É uma malha em que os processadores das bordas estão ligados entre si. O objetivo desta topologia é aumentar o número de caminhos alternativos entre dois processadores melhorando sua confiabilidade. O diâmetro permanece $O(\sqrt{n})$. O conceito de toro pode ser estendido para dimensões maiores (por exemplo um reticulado cúbico com ligações entre as faces).

- **Árvore gorda**[1]: Na árvore gorda, cada vértice, ao invés de um, possui dois ou mais pais. Sendo assim, se supusermos que cada vértice possui dois pais, podemos afirmar que cada folha da árvore possui 2 pais, 4 avôs, 8 bisavôs e assim sucessivamente. Esta topologia possui a facilidade de podermos aumentar o número de processadores facilmente. Seu uso se tornou mais difundido com a proposta da máquina CM-5 (veja seção 3.4.6). Possui diâmetro $O(log_d n)$, onde $d$ é o número máximo de filhos que um dado vértice pode ter.

Já a caracterização por **Memória Compartilhada** supõe que a memória é visualizada pelos processadores como um único espaço de endereçamento, podendo até estar fisicamente distribuída entre eles. A comunicação entre os processadores dá-se através da memória compartilhada, assim um processador $A$ desejando comunicar-se com um $B$ deve escrever em uma posição da memória para que $B$ leia desta mesma posição.

Uma outra caracterização de máquinas paralelas é a proposta por Flynn[Fly72], que apesar de ser considerada ambígua em certos casos, torna-se interessante devido a sua simplicidade. As classes de arquiteturas foram propostas de acordo com a unicidade ou multiplicidade de dados e instruções:

- *Single-Instruction Stream—Single-Data Stream* (SISD): Classe que compreende as arquiteturas seqüenciais, assim consideradas, por não expressarem nenhum tipo de paralelismo, seja de dados seja de instruções. Alguma dúvida pode ocorrer com relação a arquiteturas seqüenciais que se utilizam de *pipelining* (vide figura 2.2), porém a grande maioria dos autores as consideram como pertencentes a esta classe.

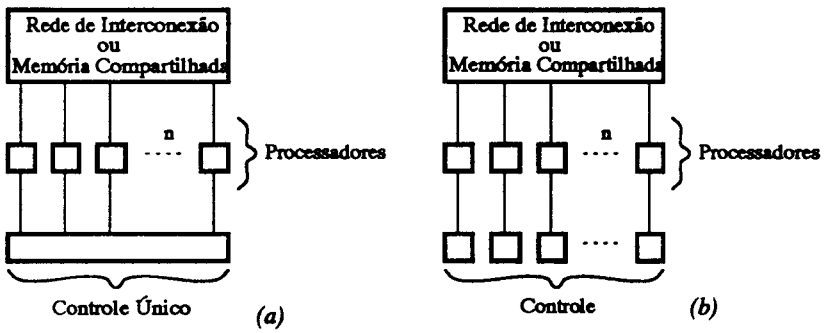


Figura 3.2: Estrutura das duas classes de arquiteturas mais importantes para a computação paralela: *(a)* SIMD *(b)* MIMD.

[1] *Fat tree*

- *Single-Instruction Stream—Multiple-Data Stream* (SIMD): Abrange arquiteturas que possuem somente paralelismo de dados (uma mesma instrução é executada por diversos processadores contendo dados diferentes). Os processadores trabalham sincronamente, porque as instruções são geradas em uma única unidade de controle (vide figura 3.2). Além disso cada processador possui uma pequena memória local que é utilizada no manuseio dos dados de uma operação.

- *Multiple-Instruction Stream—Multiple-Data Stream* (MIMD): As arquiteturas pertencentes a esta classe possuem tanto paralelismo de dados quanto de instruções. Os processadores podem trabalhar assincronamente, pois as instruções são geradas por unidades de controle diferentes. Cada processador executa sua instrução no seu próprio dado (vide figura 3.2).

A seguir comparamos as classes das duas classificações apresentadas, objetivando identificar melhor suas diferenças e realçar suas similaridades. Antes, porém, apresentamos duas definições necessárias a um melhor entendimento de tal comparação:

**Definição 3.3** *Contenção por Memória é uma situação onde vários processadores acessam um mesmo módulo de memória ao mesmo tempo. A concorrência existente entre estes processadores faz com que a latência de memória seja maior.*

**Definição 3.4** *O grau da contenção é igual ao número de processadores que participam da contenção por memória.*

Compararemos, a seguir, a classificação quanto a organização da memória global e em seguida a classificação de Flynn.

## 3.2 Memória Compartilhada vs. Distribuída

Ao usarmos o termo memória distribuída temos uma tendência natural de pensarmos em arranjo físico de memória. Desta forma, usaremos nesta comparação o termo **troca de mensagem** objetivando evitar qualquer tipo de ambigüidade que possa surgir em casos onde a memória é classificada como compartilhada mas está fisicamente distribuída entre os processadores.

Um modelo de memória compartilhada se utiliza da memória para qualquer comunicação entre os processadores, enquanto que no modelo de troca de mensagem a comunicação entre os processadores é realizada através de transmissões e recepções de mensagens. Uma descrição mais detalhada pode ser encontrada na seção 3.1.

LeBlanc e Markatos [LM93] nos mostram que os dois modelos possuem boas características e que não devemos dizer que um modelo ou outro é o melhor sem analisarmos os vários fatores que influenciam seu uso, como por exemplo a própria aplicação.

Um obstáculo existente na construção de arquiteturas com memória compartilhada (não fisicamente distribuída) é a existência de um limitante superior no número de processadores que podem estar ligados ao seu barramento, prejudicando assim a escalabilidade da arquitetura. Logo, a construção de máquinas paralelas com grande número de processadores torna-se difícil, o que nos leva a acreditar que o uso da memória compartilhada é aconselhável somente se a mesma encontrar-se fisicamente distribuída entre os processadores.

Quando ocorre sincronização, cada processador toma conhecimento da execução de outros. Nos modelos de troca de mensagem a sincronização, caso seja necessária, é efetuada através de troca de mensagens entre os processadores. Já nos de memória compartilhada a sincronização é feita através da escrita na própria memória.

Uma característica favorável ao modelo de troca de mensagem, com relação a acesso à memória, é o fato de este tirar proveito da localidade de referência, trazendo os dados para a memória local antecipadamente. As máquinas de memória compartilhada também podem se utilizar de busca antecipada, mas a um custo de uso de memórias *cache*.

A maior vantagem do modelo de memória compartilhada é a diminuição do tempo geral que os processadores ficam parados. Segundo LeBlanc e Markatos [LM93], os modelos de memória compartilhada se utilizam, em geral, de uma fila de tarefas[2], onde à medida que um processador termina sua tarefa ele retira outra da fila de execução escondendo a latência de memória existente. No modelo de troca de mensagem isso pode até ser implementado mas, em geral, as divisões de tarefas entre os processadores são efetuadas estaticamente antes da execução, levando alguns processadores a ficarem parados esperando pelo término de outros. Note que a estrutura do modelo de memória compartilhada permite a implementação de fila de tarefas mais facilmente.

## 3.3 SIMD vs. MIMD

Existem discussões a respeito de qual seja a melhor arquitetura paralela: SIMD ou MIMD. O que podemos afirmar é que as duas possuem características importantes e talvez não devam ser vistas como concorrentes.

As arquiteturas MIMD são consideradas de propósito mais geral pois, teoricamente, podem executar programas destinados a arquiteturas SIMD. Na arquitetura MIMD os processadores trabalham assincronamente, mas podem, se necessário, trabalhar sincronamente, bastando para isso que exista um mecanismo para executar este controle. Já as arquiteturas SIMD são fixamente síncronas pois, como suas instruções são geradas pelo mesmo controle, não existe a possibilidade de um processador se atrasar com relação a outros.

[2] *threads*

Nas arquiteturas MIMD o assincronismo inerente possibilita que os processadores mais rápidos não sejam atrasados pelos mais lentos, só que se alguma sincronização for necessária seu custo será muito alto [AG94].

Uma boa característica existente em máquinas SIMD é sua maior facilidade de programação. Um caso ilustrativo é o da contenção por memória em arquiteturas SIMD; ocorre que, ou todos processadores necessitam de um recurso compartilhado ou nenhum, facilitando assim a contabilização deste custo. Por outro lado, em arquiteturas MIMD, a contenção por memória pode ser variável, tornando difícil a estimativa do custo associado que é dado sempre pelo grau da contenção.

As arquiteturas SIMD necessitam de um menor espaço para o armazenamento do conjunto de instruções (controle) pois estas são as mesmas para todos os processadores. Apesar disso, existe um custo relacionado com o tempo de envio de uma instrução da memória para cada processador, seja esta compartilhada ou distribuída. Em oposição, as arquiteturas MIMD exigem que o conjunto de instruções esteja replicado em todos os processadores caso este seja o mesmo. Tal problema pode não ser muito sério caso o custo de memória continue decaindo com relação ao custo dos processadores [AG94].

Apesar de existirem estas discussões sobre as diferenças entre estes dois tipos de arquitetura, existem estudos que nos mostram a possibilidade de simulação de uma em outra a um custo bastante atrativo. Campbell [Cam94], por exemplo, nos mostra que se garantirmos alguns fatores podemos simular MIMD em SIMD ou vice-versa com relativa facilidade.

## 3.4 Máquinas Paralelas

Apresentaremos abaixo descrições sucintas de algumas máquinas paralelas existentes no mercado. O leitor notará que existe uma diversidade muito grande entre cada uma destas máquinas, de tal sorte que nos faremos valer de uma tabela, mostrada na seção final, para um melhor esclarecimento de tais diferenças.

Antes de apresentarmos as descrições, porém, mostraremos uma definição que nos será de grande importância para melhor entendimento destas:

**Definição 3.5** *Largura de Banda*[3] *é a taxa de transferência de dados máxima entre dois componentes, geralmente medida em número de mensagens por unidade de tempo, a qual pode impor limitações no desempenho total de uma máquina [Hay88].*

[3] *Bandwidth*

### 3.4.1 CM-2 (Connection Machine 2)

Configurável de 4.092 até 65.536 processadores, a CM-2 é uma máquina SIMD com memória distribuída. Seus processadores estão organizados em um hipercubo, onde cada vértice deste hipercubo equivale ao que é chamado de *sprint-node*. Estes *sprint-nodes* são formados por 32 processadores não poderosos (1 bit), com estrutura semelhante a mostrada na figura 3.3 [Mac92, AG94].

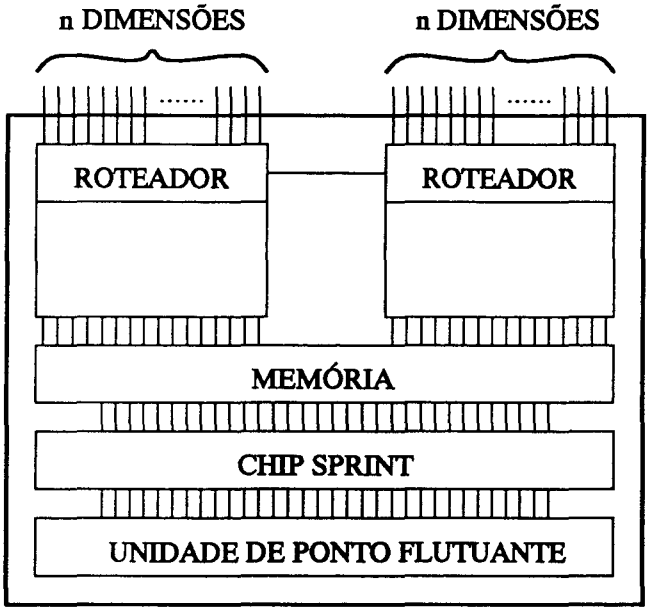


Figura 3.3: Estrutura de um *sprint-node* da CM-2.

Essa estrutura em hipercubo, com cada vértice sendo um *sprint-node*, é talvez a característica particular desta arquitetura. Utilizando esta estrutura uma CM-2 com 65.536 processadores possui 2.048 *sprint-nodes* organizados em um hipercubo de 11 dimensões. Os processadores são customizados enquanto que as memórias utilizadas são DRAMs padrão usadas comercialmente [Mac92, RO93].

A CM-2 se destaca também por ser a máquina predecessora da CM-5.

### 3.4.2 DAP-6xx (Distributed Array Processor)

Lançada em meados de 1992 a série DAP-6xx se constitui de máquinas SIMD de memória distribuída. Seus processadores estão organizados em uma topologia em malha de 64 x 64 processadores não poderosos, customizados, e onde cada um está ligado diretamente a sua memória local.

Como memória, são utilizadas RAMs estáticas que, apesar de possuírem um alto custo, visam diminuir o tempo de acesso [Mac92]. O tamanho da memória local a cada processador não pode ultrapassar 1 Mbit, mas geralmente varia de 32 a 256 Kbit por processador.

Antes do lançamento do primeiro DAP-6xx, a série topo de linha era a DAP-5xx que possui 32 x 32 processadores em malha. Note que o número máximo de processadores é diferente em cada série e é definido como sendo sempre uma malha de $2^y$ x $2^y$ processadores onde $y$ é dado no nome da série (DAP-yxx). Neste mesmo sentido, a velocidade em MHz da série é identificada pelo valor $xx$. Exemplificando, uma máquina DAP-610 possui 64 x 64 processadores e *clock* de 10MHz.

Esta máquina se destaca por possuir a arquitetura que mais se aproxima da arquitetura SOLOMON[4].

### 3.4.3 MP-1 (MasPar)

Configurável de 1.024 a 16.384 processadores em duas séries (MP1100 a MP1200) a MP-1 é uma máquina SIMD de memória distribuída. Os processadores são customizados e estão organizados em uma topologia em malha, mas com grupos de 16 processadores em malha 4 x 4. A este tipo de rede é dado o nome de X-NET (vide figura 3.4). A largura de banda desta rede cresce quase linearmente com o número de processadores [AG94, Mac92].

Com relação a memória, a MP-1 utiliza atualmente DRAMs de 16 Kbytes por processador, o que dá uma memória total de 256 Mbytes na configuração com 16.384 processadores.

Uma característica importante da MP-1, e talvez a que a tornou diferente das CM-2 e DAP, é que esta já no seu primeiro projeto visualizava a utilização de unidades de ponto flutuante, o que nas outras máquinas citadas não era previsto nos projetos iniciais, sendo incorporada posteriormente.

### 3.4.4 KSR1 (Kendall Square Research)

A KSR1 é uma máquina MIMD configurável de 8 até 1088 processadores no projeto atual, mas pode ser estendida para dezenas de milhares de processadores [AG94]. Utiliza-se de um sistema de *caches* privadas mantidas consistentes via *hardware* chamado ALLCACHE

Através do sistema ALLCACHE é possível se ver o espaço de endereçamento, que está fisicamente distribuído entre os processadores, como um único espaço de endereçamento compartilhado por todos os processadores. Este sistema é bastante diferente dos sistemas convencionais. Geralmente as *caches* em sistemas convencionais são um meio de armazenamento local e temporário de dados e onde existe uma memória global que contém uma cópia permanente dos dados. Já no sistema ALLCACHE temos o primeiro nível de *caches* mas, ao invés da memória global, existe um segundo nível de *caches* com uma cópia dos dados que são, de tempos em tempos, copiadas em disco [AG94].

[4] ***Simultaneous Operation Linked MOdular Network*** [SBM62], máquina que mais influenciou as máquinas SIMD atuais.

Apesar de ser uma máquina assíncrona, a sincronização na KSR-1, quando necessária, pode ser feita através do uso de variáveis condicionais, regiões críticas, barreiras ou pelo travamento dos dados através uso de funções adequadas a este trabalho.

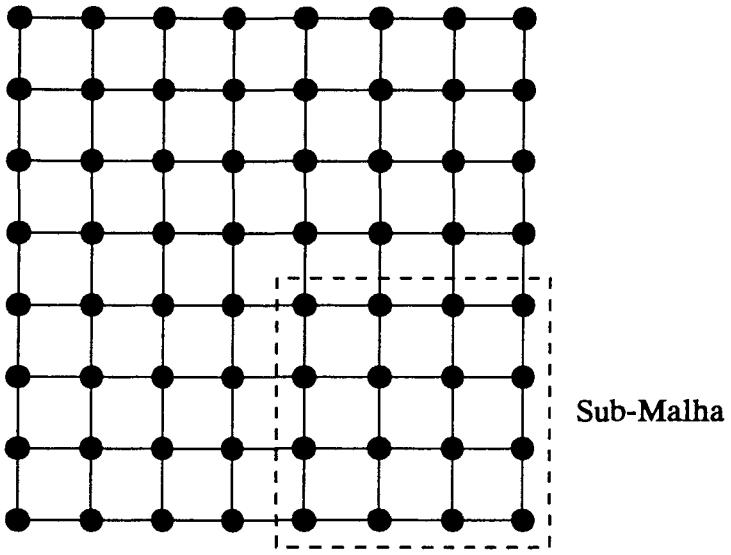


Figura 3.4: Estrutura em malha da MasPar MP-1, chamada particularmente de X-NET.

### 3.4.5 *Sequent Symmetry*

Baseada nos processadores de prateleira Intel 80X86, a *Sequent Symmetry* é uma máquina MIMD de memória compartilhada, configurável com até 30 pares processador-coprocessador. Apesar de serem poucos processadores, o desempenho da máquina é muito bom e gira em torno de 150 Mips para a arquitetura com 30 pares de processador-coprocessador. Este desempenho é conseguido devido ao fato de os processadores utilizados serem poderosos, ao invés de processadores pequenos existentes em outras máquinas.

Seus processadores estão organizados em uma topologia baseada em barramento. Todos os processadores estão conectados a um barramento central que é o mesmo para controladores de I/O e memória.

O uso de um sistema de barramento central pode gerar alta contenção devido a concentração de tarefas neste barramento. Especificamente na *Sequent Symmetry* este problema seria sério com mais de 4 processadores se não existisse uma *cache* de 64 Kbytes por processador que diminui o tráfego de dados. Além disso, o uso de um barramento mais largo suportando até 64 bits de dados ou 32 bits de dados e 32 bits de controle, multiplexados, minimiza o problema de contenção.

A característica mais importante da *Sequent Symmetry*, é o fato de ela utilizar processadores de prateleira, permitindo uma maior flexibilidade da arquitetura. Atualmente, a última máquina oferecida, a *Sequent* 2000, já utiliza processadores Intel da família 80486.

### 3.4.6 CM-5 (Connection Machine 5)

Último modelo lançado pela *Thinking Machines Corporation*, a CM-5 é uma máquina que possui uma arquitetura bastante diferente de suas antecessoras (CM-2 e CM-200). A CM-5 é classificada como MIMD mas possui muitas características também de SIMD, principalmente pelo fato de ter instruções rápidas de sincronização. Muitos problemas existentes na CM-2 foram resolvidos na CM-5 principalmente os problemas de não facilidade de expansão inerentes à organização dos processadores em hipercubo.

A CM-5 possui memória distribuída entre os processadores que estão organizados em árvores gordas, que é uma estrutura mais flexível que o hipercubo. O hipercubo dificulta o acréscimo de mais processadores, já que este não pode ser feito arbitrariamente. O hipercubo só pode ser expandido caso dobremos o número de processadores. As árvores gordas, por sua vez, podem ser expandidas com um número arbitrário de processadores além de possuírem propriedades de tolerância a falhas (vide figura 3.1), já que os vértices possuem mais de um vértice pai [RO93, AG94]

A implementação atual da CM-5 possui processadores RISC de prateleira baseados em SPARC com uma memória de 32 Mbytes por processador. A escolha de processadores baseados em SPARC facilita um pouco o processo de desenvolvimento de *software* pelo fato de já existir um bom conhecimento de como se desenvolver *software* para SPARC. Notamos também que cada processador da CM-5 é muito mais poderoso do que um processador da CM-2.

A CM-5 pode chegar a um número máximo de 16.384 processadores. Atualmente porém, a maior máquina disponível possui apenas 1.024 processadores, resultando em uma máquina com 32 Gbytes de memória total.

### 3.4.7 Cray T3D

Considerada a primeira máquina maciçamente paralela da Cray Research, o Cray T3D é uma máquina MIMD de memória compartilhada, mas onde esta está fisicamente distribuída entre os processadores. Dizemos que a memória é compartilhada porque qualquer processador pode ler ou escrever em memórias não-locais sem conhecimento prévio dos processadores locais a estas.

O acesso à memória possui custos diferentes dependendo de os acessos serem locais ou não. O tempo de acesso à memória gira em torno de 25 ciclos de *clock* enquanto que a memórias não-locais gira em torno de 225 ciclos de *clock*. Já com relação a largura

de banda entre memória e processador temos taxas em torno de 320 Mbytes/seg para memórias locais e 100 Mbytes/seg para memórias não-locais.

O tamanho da memória do Cray T3D é dado pelo número de processadores multiplicado pelo tamanho da memória local a cada processador, que varia de 16 a 64 Mbytes. Logo uma máquina com 1024 processadores possui uma memória de até 64 Gbytes.

Os processadores do Cray T3D estão organizados em um reticulado cúbico com ligações entre as faces (um toro). Nesta máquina existem dois processadores por vértice do toro, e por isso uma configuração de 1024 processadores usa um toro de 512 vértices (8 x 8 x 8). A justificativa para o uso do toro como topologia de interconexão é o fato de ele possuir caminhos redundantes entre processadores, aumentando assim a confiabilidade do sistema. Além disso o toro possui baixa latência (devido ao seu diâmetro) quando comparado com outras topologias [KFW94].

A proposta do Cray T3D é bastante interessante pois atenta para pontos críticos em computação paralela. A arquitetura possui mecanismos para esconder o custo das operações de longa latência, como fila de busca antecipada[5]. Com relação à sincronização, vários mecanismos existentes facilitam este trabalho independentemente da granularidade ou do tipo de paralelismo (de dados ou de controle), como: Barreira de Sincronização e Mensagens entre os processadores.

Finalmente, vale salientar que o Cray T3D possui uma boa escalabilidade, sendo esta quase linear [KFW94].

### 3.4.8 NCUBE/x

Sistema MIMD com configurações variando de 16 a 8192 processadores, a família NCUBE/x é constituída de máquinas com processadores de prateleira ligados em hipercubo. O termo $x$ que identifica o nome da máquina é igual ao logaritmo base dois do número de processadores.

Apesar de ser uma topologia muito flexível em termos de aumento do número de processadores, o hipercubo é bastante modular e pode ser dividido em hipercubos menores que podem ser alocados a diferentes aplicações. Cada vértice do hipercubo constitui-se de um processador customizado e um conjunto de 6 a 10 *chips* de memória de até 64 Mbytes.

[5] *prefetch queue*

### 3.4.9 Quadro Comparativo

Como observamos as máquinas paralelas apresentadas possuem características arquiteturais as mais distintas. Abaixo sumarizamos estas características pela tabela 3.1.

| Característica / Máquina | Topologia | Organização de Memória | Classificação Flynn | Tipos de Processadores | Tamanho dos Processadores |
|---|---|---|---|---|---|
| **CM-2** | Hipercubo | Distribuída | SIMD | Customizado | Pequenos |
| **DAP-6xx** | Malha | Distribuída | SIMD | Customizado | Pequenos |
| **MP-1** | Malha (X-NET) | Distribuída | SIMD | Customizado | Pequenos |
| **KSR1** | Hierárquico | Compart. | MIMD | Customizado | Poderosos |
| ***Sequent Symetry*** | Barramento | Compart. | MIMD | Prateleira | Poderosos |
| **CM-5** | Árvore Gorda | Distribuída | MIMD | Prateleira | Poderosos |
| **Cray T3D** | Toro | Compart. | MIMD | Prateleira | Poderosos |
| **NCube/X** | Hipercubo | Distribuída | MIMD | Customizado | Poderosos |

Tabela 3.1: Resumo das características de algumas máquinas paralelas

# Capítulo 4

# Modelo PRAM

## 4.1 Computação Paralela como Substituta da Computação Seqüencial

Atualmente, um dos principais desafios da ciência da computação e da indústria de computação é determinar a extensão em que a computação paralela pode atuar e se é possível esta substituir gradualmente a computação seqüencial, tornando-se o que McColl chama de **Computação Paralela de Propósito Geral**[1][McC93, McC94b].

Na seção 3.4 fizemos uma breve descrição de algumas máquinas paralelas existentes e vimos que elas possuem características bem distintas. Apesar de um possível ganho em desempenho, a computação paralela não conseguiu ainda se tornar comercialmente atrativa. As indústrias de *hardware* não possuem muitas máquinas paralelas disponíveis no mercado já que as máquinas seqüenciais estão a cada dia com melhor desempenho e menor custo, e todos nós sabemos a importância do custo de uma máquina para seu sucesso no mercado. Além disso, grande parte das máquinas paralelas existentes são especializadas na resolução de problemas mais complexos, principalmente na área científica. Já as indústrias de *software* não se sentem muito atraídas pela computação paralela devido ao pequeno número de máquinas existentes no mercado e a grande diferença entre elas, o que geraria a necessidade de uma versão de *software* para cada arquitetura.

McColl defende que o surgimento de um modelo de computação paralela que exerça o mesmo papel que a arquitetura *von Neumann* para a computação seqüencial, poderá aumentar o interesse das indústrias de *hardware* e *software* pela computação paralela, pois tal modelo formaria uma base de pensamento em que ambas as indústrias pudessem confiar. Além disso, a computação paralela se tornaria gradualmente um método simples de computação, expandindo seus limites de atuação.

[1] *General Purpose Parallel Computing*

## 4.2 Modelos de Computação Paralela

O surgimento da arquitetura *von Neumann* permitiu a difusão da computação seqüencial. Desta forma, acreditamos que o surgimento de uma arquitetura padrão para a computação paralela poderá difundi-la.

Após o surgimento da arquitetura *von Neumann*, a necessidade de um modelo de computação que viesse facilitar a tarefa de desenvolvimento de algoritmos se tornou clara, daí o surgimento do modelo RAM. Em computação paralela o processo é um pouco diferente dado a diversidade de arquiteturas existentes. Sendo assim acreditamos que um modelo de computação paralela deva ser o mais genérico possível, conseguindo abstrair, senão todas, as principais arquiteturas paralelas existentes. Com base nestas observações, podemos afirmar que enquanto não surgir um modelo genérico, a computação paralela dificilmente atingirá a generalidade existente na computação seqüencial.

Com base nos problemas existentes em computação paralela, podemos acrescentar algumas necessidades específicas àquelas já descritas na seção 2.1:

- **Exploração do Paralelismo:** Necessitamos de um modelo de computação paralela que nos permita expressar, de uma maneira simples, o paralelismo existente em alguns problemas, objetivando assim, reduzir o tempo de computação dos algoritmos.

- **Transportabilidade:** Os modelos de computação paralela devem, se possível, abstrair diversas arquiteturas distintas entre si, permitindo que programas desenvolvidos em uma máquina sejam transportáveis para outras com o menor número possível de modificações. Também é desejável que esta transportabilidade seja independente do número de processadores e do tipo de memória: distribuída ou compartilhada.

A análise de quão bom é um modelo de computação paralela é feita verificando-se o quanto ele atende às necessidades acima, em adição aos dois fatores principais: simplicidade e realidade.

Além da verificação de fatores teóricos, como os apresentados acima, a comprovação experimental é um fator extremamente importante na avaliação de um modelo de computação paralela [AKP93]. Em computação paralela a necessidade de comprovação experimental é mais intensa devido a grande diversidade de fatores como: classes de arquiteturas, arranjos de memória, processos de comunicação, entre outros.

## 4.3 O Modelo PRAM

### 4.3.1 Descrição

O PRAM é o modelo de computação paralela mais popular. Até o surgimento do PRAM os modelos existentes em computação paralela como redes de ordenação[2] e circuitos, dificultavam o desenvolvimento de algoritmos que utilizam estruturas de dados. Os modelos de redes de ordenação e os circuitos não possuem memória definida, fazendo com que leituras e escritas não sejam permitidas dentro de uma execução. Um problema a ser resolvido nestes modelos deve ter uma entrada inicial e uma única saída gerada pelo modelo.

Observando o problema definido acima o modelo PRAM foi proposto como uma extensão paralela do modelo de computação seqüencial RAM.

O modelo PRAM de computação paralela consiste, em linhas gerais, das seguintes partes:

- um **conjunto de processadores idênticos** que trabalham sincronamente. Um processador só pode avançar ao passo seguinte de sua execução se todos os outros processadores tiverem terminado seus respectivos passos correntes.
- uma **memória compartilhada**, ilimitada, que pode ser acessada por qualquer número de processadores indistintamente para leitura e escrita em uma unidade de tempo.

Os algoritmos desenvolvidos no modelo PRAM consideram no seu projeto a possibilidade ou não de acesso concorrente a uma determinada posição da memória compartilhada. Sendo assim podemos afirmar que o modelo está dividido em submodelos de acordo com a possibilidade ou não de um algoritmo possuir acesso de escrita ou leitura concorrente a posições de memória. Estes submodelos são:

- *Exclusive Read Exclusive Write* (EREW PRAM): Este é o submodelo mais fraco. Os projetistas devem considerar que os algoritmos não podem conter instruções de leitura ou escrita concorrente.
- *Concurrent Read Exclusive Write* (CREW PRAM): No desenvolvimento de algoritmos neste submodelo, os projetistas podem usar a facilidade de vários processadores poderem ler de uma mesma posição de memória, mas se atendo ao fato de somente um processador poder escrever em uma mesma posição de memória por vez.
- *Concurrent Read Concurrent Write* (CRCW PRAM): Este submodelo é o mais forte. Os projetistas de algoritmos podem se utilizar da facilidade de todos os processadores poderem ler ou escrever em uma mesma posição de memória ao mesmo tempo.

[2] *sorting networks*

- *Exclusive Read Concurrent Write* (ERCW PRAM): A existência deste submodelo é descrita em alguns trabalhos, como [Akl89]. Nele os projetistas de algoritmos podem usar a facilidade de vários processadores poderem escrever ao mesmo tempo em uma mesma posição de memória, mas não podendo haver leitura simultânea a uma mesma posição de memória. Vale observar que este submodelo não é muito comum pois a permissão de escrita concorrente é um problema mais sério que o de leitura concorrente. Em geral, quando o problema de escrita concorrente é resolvido o de leitura pode ser resolvido sem nenhuma grande modificação no submodelo.

Apesar das diferenças nas restrições dos submodelos, existem resultados que mostram que o modelo mais fraco consegue simular o mais forte com perda de $O(log\ n)$ no tempo, onde $n$ é o tamanho da entrada [CLR90].

No caso dos modelos ERCW e CRCW podem aparecer conflitos de escrita. Com o objetivo de controlar estes possíveis conflitos, são utilizadas algumas políticas de acesso. As mais usadas são:

- **Comum:** A escrita é efetuada somente se todos os processadores que estão tentando escrever na mesma posição contiverem o mesmo dado.
- **Prioridade:** Existe uma prioridade de escrita entre os processadores. Havendo conflito, o processador de maior prioridade terá sucesso.
- **Arbitrário:** Dentre os processadores que geraram o conflito, somente um deles, escolhido arbitrariamente, terá sucesso.

### 4.3.2 Avaliação de Desempenho em PRAM

Existem várias propostas de como se avaliar um algoritmo PRAM. Nesta seção descreveremos duas que consideramos mais importantes.

#### Custo por Tempo vs. Processadores

Este método de avaliação consiste em achar o custo do algoritmo, $C(n)$, que é o tempo de execução do algoritmo multiplicado pelo número de processadores necessários a esta execução. Logo temos que $C(n) = T(n) \times P(n)$, onde $T(n)$ é o tempo de execução do algoritmo e $P(n)$ é o número de processadores necessários a execução [CLR90].

Este talvez seja o método de avaliação mais difundido dentre os existentes para o modelo PRAM. Para fazermos uso dele, temos que, ao desenvolver um algoritmo. nos preocupar com quantos processadores existem e como dividir a tarefa entre eles. Tendo o algoritmo pronto, fazer uma análise de complexidade se torna uma tarefa simples para um usuário com experiência em análise de algoritmos seqüenciais.

Um dos motivos que influenciou na difusão deste método, é que dado um algoritmo paralelo com custo igual a $C(n)$, podemos transformá-lo em um seqüencial equivalente, fazendo o único processador existente simular os $P(n)$ processadores existentes no caso paralelo. Este fato possibilita que cotas superiores para algoritmos paralelos possam ser levadas para o caso seqüencial.

Verificamos a otimalidade de um algoritmo através de uma comparação com algoritmos seqüenciais. Desta forma, dizemos que um algoritmo é ótimo se seu custo for assintoticamente igual ao tempo de execução do melhor algoritmo seqüencial. Formalmente temos que um algoritmo é ótimo se $C(n) = \theta(T^*(n))$, onde $T^*(n)$ é o tempo de execução do melhor algoritmo seqüencial [CLR90, Jáj92].

#### Custo por Número de Operações

Este tipo de avaliação consiste em contar o número de operações de um algoritmo, $W(n)$, independente de estas estarem sendo executadas em paralelo ou não [Jáj92].

Para fazermos uso deste método é desejável que projetemos os algoritmos paralelos com base no **paradigma de tempo-trabalho**[3], que nos propõe fazer o desenvolvimento através de um método *top-down* em dois níveis.

No primeiro nível devemos descrever os algoritmos como uma seqüência de passos onde podem existir operações concorrentes de qualquer grau[4]. Já no segundo nível (mais baixo), devemos seguir o **princípio do escalonamento tempo-trabalho**. Este princípio advoga que, dado um algoritmo projetado de acordo com o primeiro nível com tempo $T(n)$ e trabalho $W(n)$, é possível executá-lo em um PRAM com $p$ processadores em $\lfloor \frac{W(n)}{p} \rfloor + T(n)$ passos ou menos.

A principal vantagem do uso deste método de avaliação é que não precisamos nos preocupar com o número de processadores nem com o escalonamento das tarefas entre eles. Todos os algoritmos são representados sem preocupação de quantos processadores existem ou como os dados devem ser divididos entre os mesmos.

Da mesma forma que no caso anterior, verificamos a otimalidade de um algoritmo através de uma comparação com algoritmos seqüenciais. Desta forma dizemos que um algoritmo é ótimo se o número de operações executadas for assintoticamente igual ao tempo de execução do melhor algoritmo seqüencial. Logo um algoritmo é ótimo se $W(n) = \theta(T^*(n))$, onde $T^*(n)$ é o tempo de execução do melhor algoritmo seqüencial [Jáj92].

### 4.3.3 Algoritmos

Nesta seção descreveremos alguns algoritmos PRAM com uma pequena análise de complexidade de acordo com os dois critérios descritos na seção anterior. Nosso objetivo principal

[3] *work-time*

[4] número de processadores que realizam tarefas concorrentes

é mostrar as diferenças existentes entre as duas formas de avaliação.

### Posicionamento em Uma Lista

O problema de **posicionamento em uma lista**[5] consiste em, dada uma lista ligada, acharmos a distância de cada elemento da lista ao final da mesma. A importância deste problema se deve principalmente ao fato de o seu algoritmo fazer uso de uma técnica importante em computação paralela chamada **salto de ponteiros**[6](vide figura 4.1)

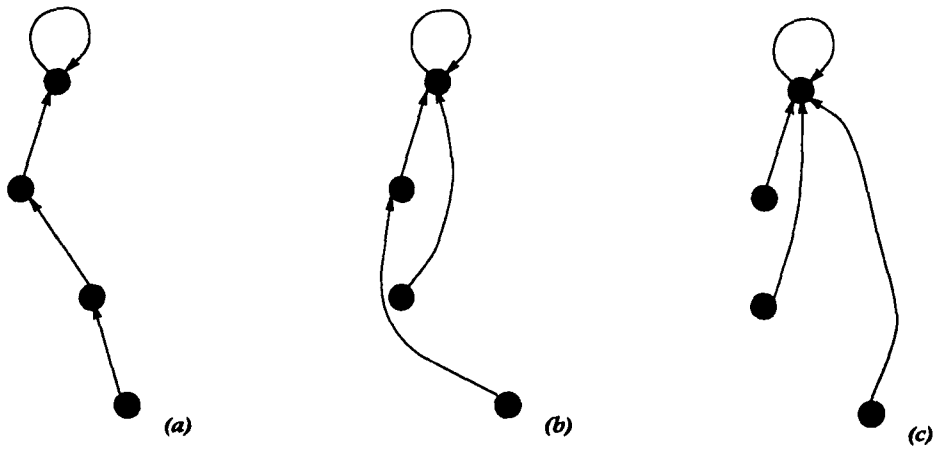


Figura 4.1: Processo da técnica de salto de ponteiros.

Apresentamos na figura 4.2 um algoritmo EREW PRAM para a resolução deste problema.

Ao analisamos o algoritmo da figura 4.2 utilizando o método de Tempo vs. Processadores podemos afirmar, pelos laços iniciados nas linhas (1) e (4), que são necessários $n$ processadores para que o tempo do algoritmo seja o menor possível. Este tempo é definido pelos passos que serão executados em cada processador. O passo (2) - (3) leva tempo constante para ser executado por cada processador. Os passos referentes a técnica de salto de ponteiros representados por (5) - (8) levam tempo $O(log\ n)$ para serem executados em cada processador, pois a cada passo do laço iniciado em (6) a lista ligada através do apontador **aux** é dividida ao meio. Como esta lista possui tamanho $n$ o tempo de execução é o logaritmo deste número.

A partir desta análise podemos concluir que o algoritmo executa em tempo $T(n) = O(log\ n)$ e que são necessários $n$ processadores para essa execução. Logo dizemos que o custo do algoritmo é $C(n) = O(n\ log\ n)$.

Diferentemente do primeiro tipo, a análise por número de operações não se preocupa com o número de processadores disponíveis para a execução do algoritmo. Aqui são con-

[5] *list ranking*
[6] *pointer jumping*

**Entrada**: Uma lista ligada e uma lista com o sucessor de cada elemento $i$ dado por $succ(i)$. $succ(i)$ para o último elemento é 0.

**Saida**: Uma lista $dist(i)$ com a distâncias de cada elemento, $i$, ao final da lista.

```
(1)          for i := 1 to n do in parallel
(2)             if succ(i) <> 0 then dist(i) := 1
(3)                             else dist(i) := 0
(4)          for i := 1 to n do in parallel
(5)             aux(i) := succ(i)
(6)             while (aux(i) <> 0) do
(7)                dist(i) := dist(i) + dist(aux(i))
(8)                aux(i) := aux(aux(i))
```

Figura 4.2: Algoritmo EREW PRAM para o problema de posicionamento em uma lista.

tadas as operações independente do seu local de execução. Vemos que o passo (2) - (3) possui somente uma operação, mas esta é executada $n$ vezes devido ao laço (1). Podemos também observar que os passos (5) - (8) executam $log\ n$ operações, que representam a técnica de salto de ponteiros; esta técnica é executada $n$ vezes devido ao laço (4).

Com esta análise podemos concluir que o algoritmo executa $O(n\ log\ n)$ operações no total, isto é, $W(n) = O(n\ log\ n)$.

Note que os valores de $C(n)$ e $W(n)$ são iguais porque não existe limitação no número de processadores, já que assumimos que temos o número de processadores necessários à execução do algoritmo. Note também que o algoritmo apresentado só possui tempo de execução $O(log\ n)$ porque os $n$ processadores necessários a sua execução estão disponíveis.

Esta igualdade entre $C(n)$ e $W(n)$ poderia não ocorrer caso tivéssemos menos processadores que o mínimo necessário a sua execução, já que o tempo, $T(n)$, de um algoritmo é diretamente ligado ao número de processadores existentes. Em contra-partida a análise realizada pelo o método de custo por número de operações, $W(n)$, independe do número de processadores existentes para a execução do algoritmo. Note que este último método de avaliação pode ser considerado mais realista já que o teorema de Brent[7] nos garante uma certa independência do número de processadores.

[7]Este teorema mostra como dividir uma tarefa para menos processadores que o necessário sem uma grande perda no tempo de execução [Bre74].

### Achar o máximo

O problema consiste em se achar o máximo de $n$ números. Apesar de simples, o problema é bastante interessante para exemplificarmos um caso onde algoritmos CRCW possuem vantagens claras sobre outros mais fracos.

Apresentaremos na figura 4.3 o algoritmo CRCW para resolução deste problema. Tal algoritmo é baseado no apresentado em [CLR90].

**Entrada**: Uma lista ligada, $A$, dos elementos e uma outra lista ligada, $aux$ de elementos lógicos, inicializados com TRUE, com o mesmo tamanho de $A$.

**Saida**: O índice na lista $A$ do elemento máximo, dado por $k$.

```
(1)          for i := 1 to n do in parallel
(2)             for j:= 1 to n do in parallel
(3)                if A[i] < A[j] then aux[i] := FALSE
(4)          for i := 1 to n do in parallel
(5)             if aux[i] = TRUE then k:=i
```

Figura 4.3: Algoritmo CRCW PRAM para o problema de achar o máximo de $n$ números.

Ao analisarmos o algoritmo da figura 4.3 utilizando o método de avaliação Tempo vs. Processadores notamos que no mínimo $n^2$ processadores são necessários para que o algoritmo possa executar no menor tempo possível, devido aos laços encadeados iniciados em `(1)` e `(2)`. Dado que temos este número de processadores podemos afirmar que o tempo necessário para execução do algoritmo é $O(1)$. Os passos `(1)` - `(3)` levam tempo $O(1)$, pois cada processador executará uma comparação e se necessário escreverá na posição `aux[i]`. Veja que neste laço existe escrita concorrente: existem $n$ processadores escrevendo ao mesmo tempo em cada uma das $n$ posições do vetor `aux`. Da mesma forma, podemos dizer que o laço `(4)` - `(5)` executa em tempo $O(1)$ se os $n$ processadores necessários para esta execução estiverem disponíveis.

Sendo assim podemos confirmar que o algoritmo acima possui realmente tempo total $T(n) = O(1)$ e custo $C(n) = O(n^2)$. Como falamos anteriormente, mostramos este algoritmo apenas para exemplificar um caso onde um algoritmo CRCW executa muito mais rapidamente que quaisquer outros. Vale salientar porém, que este algoritmo não é ótimo com relação ao custo, pois é possível se achar o máximo a um custo $C(n) = O(n)$.

A análise usando o método de número de operações é mais simples de ser feita. Ao olharmos para o algoritmo apresentado notamos claramente que os passos `(1)` - `(3)` equivalem a execução de $O(n^2)$ operações seqüenciais e que o laço `(4)` - `(5)` equivale a

execução de $O(n)$. Sendo assim podemos afirmar que o algoritmo possui um número de operações total dado por: $W(n) = O(n^2)$.

### 4.3.4 Vantagens do Uso do PRAM

Por ter sido o primeiro modelo de computação paralela no qual se pôde desenvolver algoritmos utilizando estrutura de dados e proposto como uma extensão paralela do modelo RAM, o PRAM rapidamente se tornou conhecido. Atualmente dispomos de uma vasta literatura sobre algoritmos paralelos e suas técnicas de desenvolvimento.

O uso do modelo PRAM nos traz uma série de benefícios. Suas principais vantagens são:

- **Estudo da Estrutura do Problema:** Por ser um modelo de computação paralela simples, o PRAM possibilita que conheçamos a estrutura do problema mais facilmente, já que não precisamos nos ater a detalhes não relacionados a esta estrutura.
- **Fácil Análise de Complexidade:** Por ser uma extensão paralela do modelo RAM, o PRAM nos permite analisar a complexidade de algoritmos de uma maneira quase tão simples quanto no modelo RAM. Com apenas algumas informações adicionais relativas a paralelismo, um usuário com prática em análise de complexidade no modelo RAM, consegue fazer a de um algoritmo paralelo. Da mesma forma que no modelo RAM, a complexidade de um algoritmo PRAM é expressa, geralmente, usando notação de análise assintótica.
- **Fácil adaptação por Parte dos Usuários:** A cultura adquirida através dos anos de desenvolvimento de algoritmos seqüenciais no modelo RAM permite uma fácil adaptação, em direção ao PRAM, por parte dos usuários. Devido a sua simplicidade e facilidade de uso o PRAM serve como um bom modelo para introduzir um usuário em computação paralela. Karp [Kar91] chega mais adiante e nos propõe usar o modelo PRAM em uma fase inicial de um processo de desenvolvimento de algoritmos paralelos em duas fases. Ele nos mostra que o uso do PRAM nesta fase inicial de desenvolvimento, além de nos fazer conhecer melhor o problema, nos leva a uma implementação em uma máquina real usando um processo mais adequado que o do desenvolvimento diretamente para esta máquina.

### 4.3.5 Problemas com o Uso do Modelo PRAM

Além dos problemas relacionados com a distância entre modelo e máquina (descritos na seção 2.2) e que de certa forma podem ser aplicados ao modelo PRAM, temos alguns referentes a implementabilidade deste modelo:

- **Sincronismo:** Devido a fatores como sistema operacional, meios de armazenamento com velocidades de acesso diferentes e diferença na velocidade de execução das instruções, exigirmos que os processadores se comportem de uma forma síncrona pode gerar ineficiência em uma máquina paralela, pois poderia sempre ocorrer de processadores ficarem esperando por algum outro processador que, por algum dos fatores citados acima, levasse mais tempo para terminar seu passo. Além disso temos que a construção de uma máquina que garanta sincronização por *hardware* a cada passo[8], como ocorre no modelo PRAM, pode ser muito cara.

- **Custo de Acesso à Memória:** Em máquinas paralelas reais com memória compartilhada, o custo de acesso à memória pode ser muito alto, principalmente pela possível existência de contenção no barramento ou pela própria distância que o módulo de memória desejado esteja do processador. Desta forma não é realista a contabilização de um acesso à memória como ocorrendo em um único ciclo. Algumas máquinas possuem memória local aos processadores objetivando armazenar dados temporariamente. O uso desta memória pode diminuir substancialmente os acessos à memória global, caso os dados sejam trazidos à memória local em blocos de dados consecutivos. Esta diminuição de acessos à memória global é garantida pelo princípio da localidade de referência. Porém, o modelo PRAM não considera a existência de memórias locais com este objetivo.

- **Distância entre Modelo e Arquitetura:** De uma forma geral, o modelo PRAM não possui um bom nível de precisão em suas previsões de desempenho, pois não existe uma arquitetura que predomine em computação paralela que seja com ele relacionada, como acontece no caso seqüencial entre o modelo RAM e a arquitetura *von Neumann*.

## 4.4 Modelo PRAM, Nova Visão

Apesar da sua distância às máquinas paralelas, recentemente o modelo PRAM voltou a ser bastante discutido no meio devido, principalmente, ao trabalho de Vishkin [Vis92, Vis93]. Este trabalho apresenta diversos fatores que fazem do PRAM um modelo atrativo para ser usado na prática.

Muitos trabalhos tentam incluir novos parâmetros no modelo PRAM para uma melhor representação de características existentes em máquinas paralelas, mas isto torna mais complexa sua utilização. Por outro lado, a utilização do PRAM como um modelo de computação paralela seria bastante razoável pela sua simplicidade e por permitir uma fácil adaptação dos usuários de computação seqüencial à computação paralela.

[8] *lock-step sincronization*

Atualmente poucas são as máquinas estritamente seqüenciais [Vis92]. Quase todas as máquinas ditas seqüenciais modificam mais de um dado por ciclo, seja pelo uso de *pipelining* ou por outras características de *hardware*. Isto não é compatível com a definição de máquina seqüencial, que supõe a modificação de um único dado por ciclo.

Segundo Vishkin, para um maior desenvolvimento da computação paralela, necessitamos de cursos de análise e projeto de algoritmos que incluam em sua ementa tópicos na área, tornando os usuários mais íntimos deste assunto. Além disso, é necessário que as indústrias de *hardware* tornem possível a utilização de simulação de paralelismo em suas máquinas hoje ditas seqüenciais. Isto possibilitaria uma introdução da computação paralela de uma forma menos brusca, pois os usuários poderiam gradativamente incluir em seus programas rotinas em paralelo.

Quatro são os argumentos usados por Vishkin para nos mostrar que o modelo PRAM pode ser usado na prática:

- **Algoritmos Seqüenciais Mais Rápidos:** Alguns algoritmos escritos para o modelo PRAM podem ser executados seqüencialmente de maneira mais rápida que os escritos diretamente para o modelo RAM. Isto é conseguido porque quando desenvolvemos algoritmos paralelos a distribuição dos dados entre os processadores é considerada passo a passo, possibilitando que o transformemos em um algoritmo seqüencial que se utilize de busca antecipada.

- **Emulação Eficiente:** O modelo PRAM pode ser emulado eficientemente em uma máquina paralela síncrona onde cada processador está conectado a um pequeno número de outros [MV84].

- **Sucesso da Teoria:** Dispomos atualmente de um grande número de técnicas de desenvolvimento de algoritmos para o modelo PRAM. Conseqüentemente, algoritmos para os mais diversos problemas já foram desenvolvidos neste modelo [GR88, Akl89, Jáj92, KR90].

- **Por *Default*:** A não existência de um outro modelo de computação paralela que seja mais desenvolvido e difundido entre os usuários de computação em geral, faz do modelo PRAM uma das melhores escolhas atuais.

Mesmo com o surgimento de outros modelos, possivelmente mais complexos, o modelo PRAM não deve ser esquecido. Ele poderia ser utilizado, em um passo inicial, por usuários menos preparados em computação paralela; ou em aplicações para as quais o uso de outro modelo mais complexo não nos traga benefícios. Porém, nada impede que um usuário opte por desenvolver seus algoritmos em modelos mais complexos se ele assim achar necessário ou se isto for vantajoso com relação a desempenho.

Vale salientar que, ao referir-se ao modelo PRAM, Vishkin está na realidade referindo-se a um conjunto de modelos que possui o PRAM como base. Logo qualquer uma de suas extensões pode ser utilizada como modelo de programação segundo o autor.

## 4.5 Algoritmos PRAM em uma Máquina Real

Na seção anterior mostramos pontos que podem fazer do modelo PRAM um modelo mais aplicável na prática. Além destes fatores defendidos principalmente por Vishikin, existem trabalhos que reforçam a teoria de que o PRAM pode co-existir como outros modelos padrão [HRD93a, HRD93b, KLH93]. Estes trabalhos nos mostram casos onde algoritmos PRAM são implementados eficientemente em máquinas paralelas com memória distribuída.

Um dos trabalhos citados acima é o de Hsu *et. al* [HRD93a, HRD93b] que mostra implementações de algoritmos PRAM para grafos não direcionados em uma máquina MasPar MP-1. Este trabalho é interessante pois caracteriza bem o caso onde o PRAM se faz um bom modelo de computação paralela na prática. Descreveremos sucintamente a seguir o trabalho.

Apesar de o PRAM ser um modelo que não expressa bem os custos existentes em máquinas paralelas reais, os autores usaram-no principalmente por acreditarem na viabilidade de um modelo abstrato como o PRAM modelar bem, na prática, algumas máquinas paralelas como a MasPar MP-1. Hsu *et. al* [HRD93a, HRD93b] argumentam que os algoritmos desenvolvidos em PRAM possuem, em geral, uma boa estruturação fazendo com que a implementação seja facilitada.

Um dos principais problemas existentes no mapeamento de um algoritmo PRAM para uma máquina paralela é o custo decorrente dos acessos à memória global. No PRAM sabemos que o custo de tal acesso é $O(1)$ enquanto que no caso da MP-1 o custo é de $O(n)$, onde $n$ é o número de processadores.

Para tentar resolver tal problema os autores propõem um mapeamento da memória da MP-1 para a memória do PRAM. A proposta consiste em simular a memória do PRAM pelo conjunto de parte da memória local a cada processador MP-1 juntamente com a memória global desta[9] (Vide figura 4.4).

Com o mapeamento representado na figura 4.4, é possível distribuir os dados na memória da MasPar MP-1 de tal forma a minimizar a diferença de custo de acesso à memória global existente entre o modelo PRAM e a máquina MP-1. Em geral é possível se realizar acesso à memória global da MP-1 com custo na ordem de $o(n)$.

Os autores propõem que os dados mais usados por um determinado processador sejam colocados, preferencialmente, na parte da memória global do PRAM referente a memória

[9]Ao nos referirmos a memória global da MasPar MP-1, estamos nos referindo a um componente da máquina chamado ACU (*Array Control Unit*)

local a este processador. Caso o dado seja usado bastante por todos os processadores, eles devem ser preferencialmente alocados na parte da memória global do PRAM referente a memória global da MasPar MP-1.

Já com relação a programação de algoritmos para grafos, é usada uma estratégia de alocar um processador para cada nó ou aresta do grafo a ser usado. Note então que variáveis globais ao sistema devem ser armazenadas na parte da memória global do PRAM referente a memória global da MP-1.

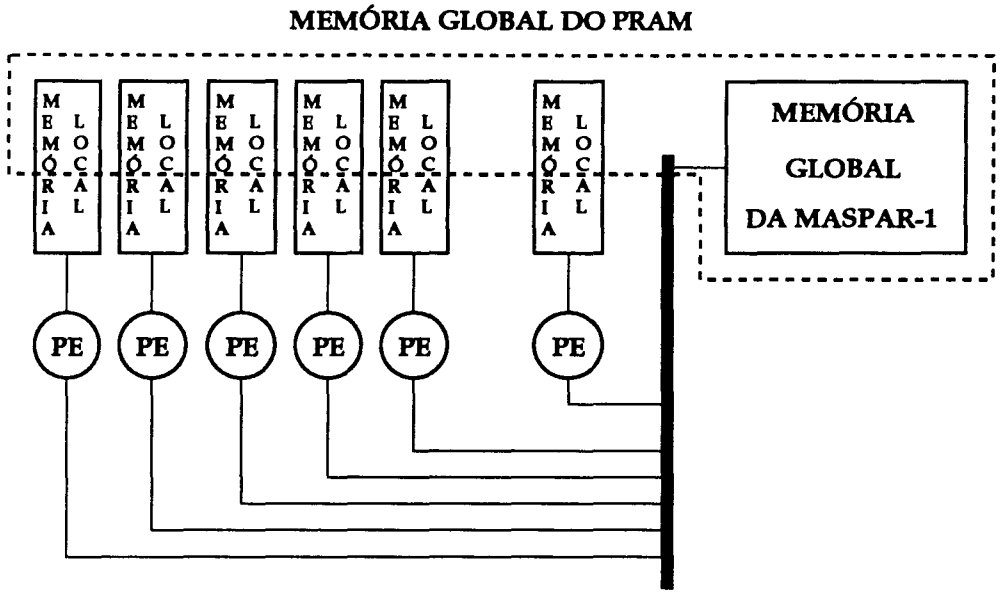


Figura 4.4: Mapeamento da memória da MasPar MP-1 para o modelo PRAM. Adaptado de [HRD93a].

A estratégia usada para mostrar a viabilidade dos resultados obtidos no modelo PRAM foi implementar algoritmos seqüenciais usando a mesma biblioteca utilizada para os algoritmos paralelos. A partir dos algoritmos seqüenciais e paralelos foi possível considerar a aceleração prática e teórica como forma de avaliação dos resultados.

Como exemplo, suponhamos a resolução do problema da floresta espalhada. Tal problema consiste em acharmos uma árvore para cada componente conexo de um grafo dado. A resolução deste problema seqüencialmente é trivial, basta que apliquemos uma busca no grafo dado. Este método possui tempo de execução $O(n + m)$, onde $n$ é o número de vértices do grafo e $m$ o número de arestas.

O algoritmo paralelo implementado no MasPar MP-1 é uma variação do algoritmo para componentes conexos apresentado por Awerbuch e Shiloach [AS87], que se utiliza de $O(n + m)$ processadores e possui tempo de execução $O(log\ n)$. Como este algoritmo é

originalmente CRCW temos que transformá-lo em EREW para que possa ser implementado em uma máquina real. Desta forma, o tempo do algoritmo aumenta por um fator multiplicativo de $O(log\ n)$, resultando em um algoritmo com tempo total $O(log^2 n)$.

No caso de grafos esparsos, podemos afirmar que o tempo do algoritmo seqüencial é $O(n)$. A aceleração teórica do algoritmo é, a menos de fatores constantes:

$$\frac{n}{log^2 n} = \frac{P}{log^2 P}$$

Na implementação deste algoritmo os autores supuseram a existência de um processador para cada vértice do grafo, logo temos que $P = n$.

Tendo implementado tanto a versão seqüencial quanto a versão paralela usando os mesmo recursos, foi possível comparar o resultado obtido com o resultado teórico.

Hsu *et. al* observaram que sua implementação do algoritmo PRAM obteve uma aceleração bastante próxima daquela prevista pela teoria. O mesmo fato ocorreu para implementações de vários outros algoritmos paralelos para problemas em grafos. Estes resultados vêm reforçar as considerações de Vishkin sobre a viabilidade do modelo PRAM.

É preciso se fazer a ressalva de que os resultados de Hsu *et. al* são válidos apenas quando o tamanho do problema casa com o número de processadores disponíveis.

# Capítulo 5

# Extensões do modelo PRAM

## 5.1 Histórico

A partir da verificação da não aplicabilidade prática do modelo PRAM, várias propostas surgiram, principalmente extensões, tentando suprir necessidades geradas pelos seus pontos fracos. Alguns destes pontos foram descritos na seção 4.3.5.

Mostramos a seguir algumas extensões do modelo PRAM que observam pontos fracos distintos. Tentaremos abordar as principais extensões para cada ponto.

Vale salientar que não tomamos todos estes modelos para um estudo mais aprofundado na dissertação porque optamos por estudar mais detalhadamente duas propostas que não têm o PRAM como modelo base. Outro ponto importante é que preferimos abordar mais profundamente modelos de computação paralela que se propõem a resolver, senão todos os problemas existentes no modelo PRAM, pelo menos aqueles considerados mais sérios. Nós veremos que estas propostas de extensão se preocupam, na maioria dos casos, com apenas um problema específico.

Apesar disso, as extensões têm seu papel, pois na maioria dos casos é possível se ter um pouco mais de proximidade com as arquiteturas sem perda de simplicidade do modelo. Acontece como se tivéssemos um modelo mais realista com a mesma simplicidade do modelo PRAM.

## 5.2 APRAM: Incorporando Assincronismo ao PRAM

No modelo APRAM [CZ89] os custos referentes à sincronização são explícitos, o que torna o modelo um pouco mais complexo. A justificativa para o uso deste modelo é que como as máquinas paralelas existentes no mercado são potencialmente assíncronas (MIMD), acredita-se que o uso de um modelo assíncrono seja mais apropriado.

O APRAM é um modelo onde a memória é compartilhada e serve como meio de comu-

nicação assíncrono: um processador que deseje comunicar-se com outros escreve o valor em uma posição de memória e não necessita esperar pela leitura do mesmo.

O modelo é definido como uma coleção de processos onde cada um é constituído de um conjunto de eventos. São eles: evento de leitura, evento de escrita e evento local (qualquer evento que ocorra na memória local, como uma leitura ou um cálculo). Estes eventos seguem uma relação de precedência definida pela relação **executado antes** e representada pelo sinal $\rightarrow$. Dados dois eventos $e_1$ e $e_2$, se $e_1 \rightarrow e_2$, então $e_1$ é executado antes de $e_2$. A mesma representação poderia ser usada para nos mostrar um caso onde $e_1$ necessitasse escrever um dado na memória compartilhada para que $e_2$ o lesse. Desta forma podemos dizer que uma computação no modelo APRAM é definida como um seqüenciamento de eventos onde estes obedecem a relação **executado antes.**

## 5.3 *Phase* PRAM: Usando Sincronização em Fases

O *Phase* PRAM [Gib89] é um modelo que elimina o sincronismo rígido existente no PRAM e adota o uso de fases onde os processadores trabalham assincronamente. Entre cada fase existe uma instrução de sincronização que não envolve obrigatoriamente todos os processadores, $P$, e sim um conjunto $S$ de processadores tal que $S \subseteq P$. A execução de uma instrução de sincronização consiste em uma parada dos processadores até que todos do conjunto $S$ tenham terminado suas tarefas. Só então, os processadores entram numa nova fase de execução assíncrona.

A consistência do modelo *Phase* PRAM é garantida pois não são permitidos acessos arbitrários à memória. Tal afirmação é ilustrada pelo fato de um processador somente poder ler uma posição de memória que outro escreveu, caso haja uma instrução de sincronização antes. Veja que, como o modelo é assíncrono, um processador não tem conhecimento da execução de outros, daí a necessidade de sincronização.

Na proposta do modelo *Phase* PRAM é definida uma família de modelos que diferem somente em pontos específicos:

- *Phase* PRAM **com sincronização por subconjunto:** Os processadores estão divididos em subconjuntos e o custo de sincronização é proporcional ao número de processadores de cada subconjunto.

- *Phase* PRAM **com sincronização de todos os processadores:** O custo de sincronização é relativo a todos os processadores. Seria similar ao anterior, onde o subconjunto compreende todos os processadores.

- *Phase* PRAM **contabilizando latência de comunicação:** Objetiva fazer uma melhor estimativa do tempo de execução dos algoritmos. Considera a latência de comunicação fixa para acessos de leitura e escrita, onde $2t$ é o tempo de acesso para

leitura (já que são necessários tanto uma requisição de leitura quanto o retorno do resultado da leitura) e $t$ é o tempo de acesso para escrita.

Nos submodelos acima citados, a execução de instruções pode se utilizar de *pipelining*, mas este somente pode ocorrer dentro de uma mesma fase.

## 5.4 BPRAM: Incorporando Transferência em Blocos de Dados

Tendo em vista o princípio da localidade de referência, podemos visualizar um modelo de computação paralela que utilize suas vantagens. No modelo *Block* PRAM [ACS89], ou simplesmente BPRAM, as transferências de dados em blocos são permitidas objetivando diminuir a latência de comunicação, que em algumas máquinas reais chega a ser centenas de vezes maior que o tempo de execução de uma operação aritmética em memória local.

O BPRAM é um modelo de memória compartilhada, ilimitada, onde cada processador possui sua memória local. São definidos dois parâmetros usados na análise de complexidade dos algoritmos:

- $l$, latência de comunicação
- $p$, número de processadores

Cada operação em memória local, executada por um processador, gasta uma unidade de tempo, enquanto que uma operação em memória global gasta $l + b$ unidades de tempo, onde $b$ é o tamanho do bloco ou a granularidade da memória. A vantagem principal é que se cada posição de memória de $b$ fosse lida individualmente gastaríamos $l \times b$ unidades de tempo para isso. Os acessos à memória global são exclusivos tanto para leitura quanto para escrita.

## 5.5 H-PRAM: Uma Hierarquia de PRAMs

Uma constante nas extensões do modelo PRAM é o fato de estas modificarem características fundamentais do modelo, dificultando que algoritmos já desenvolvidos para o PRAM sejam aproveitados em tais extensões. Face a isto, o modelo H-PRAM [HR92] segue um caminho diferente, fazendo do PRAM seu submodelo. O H-PRAM é um modelo que utiliza uma hierarquia de PRAMs configurável dinamicamente.

De uma forma geral, o modelo H-PRAM introduz uma nova função ao PRAM, que é subdividir uma tarefa em tarefas menores que possam ser resolvidas por PRAMs com menos processadores. Esta divisão pode ser feita até que as subtarefas atinjam o seu

limite mínimo, quando cada uma pode ser resolvida por um processador. Feita a divisão adequada da tarefa base, as subtarefas são executadas em sub-PRAMs de níveis cada vez mais baixos na hierarquia (visualizando a hierarquia como uma árvore) (vide figura 5.1).

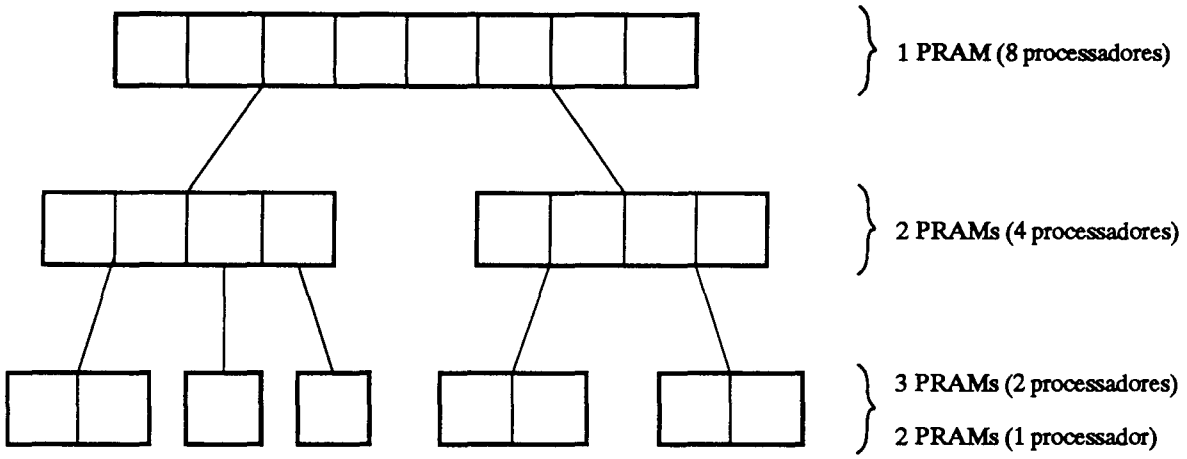


Figura 5.1: Macro estrutura do modelo H-PRAM.

Quando há uma divisão de tarefas em subtarefas de níveis inferiores na hierarquia, os sub-PRAMs que as executarão trabalham assincronamente entre si, e a execução só retornará ao nível superior desta hierarquia após uma sincronização.

Com relação aos acessos à memória compartilhada, o modelo H-PRAM pode ser dividido em duas categorias:

- H-PRAM **Privado:** A memória compartilhada é dividida igualmente entre os processadores, e cada PRAM possui, na hierarquia, direito de acesso apenas à parte da memória associada aos seus processadores.
- H-PRAM **Compartilhado:** Caso em que um PRAM de qualquer nível hierárquico possui direito de acesso a toda memória compartilhada do H-PRAM.

Vários são os pontos interessantes deste modelo. Podemos citar alguns como:

- O modelo pode ser visto como de propósito mais geral que outros, pois as subdivisões não necessariamente devem ser em PRAMs básicas, podendo existir subdivisões em BPRAMs, por exemplo. A única exigência é que o modelo usado na subdivisão seja síncrono.
- A capacidade da hierarquia de PRAMs poder ser configurada dinamicamente, permitindo que o modelo se adeqüe a problemas de granularidade variada.

- Fatores como latência de memória e custo de sincronização estão contabilizados implicitamente no custo da função de divisão de tarefas, não sendo necessário o uso de parâmetros que possam vir a diminuir a simplicidade do modelo.

## 5.6 QRQW PRAM: Contabilizando Contenção

Apresentaremos aqui uma extensão do modelo PRAM bastante recente, o que nos mostra que o PRAM não é de forma alguma um modelo a ser esquecido. Esta nova proposta veio também reforçar as considerações de Vishkin apresentadas no capítulo anterior. Faremos uma descrição mais longa deste modelo tentando deixar claro o objetivo geral de uma extensão do PRAM.

### 5.6.1 Definição do Modelo

A proposta de Gibbons *et. al* [GMR94c, GMR94a, GMR94d, GMR94b] consiste basicamente da inclusão de uma nova regra para tratamento de acessos concorrentes àquelas já descritas na seção 4.3.1, possibilitando que os custos decorrentes da contenção por memória sejam contabilizados (vide figura 5.2). À nova regra proposta foi dado o nome de **fila**[1].

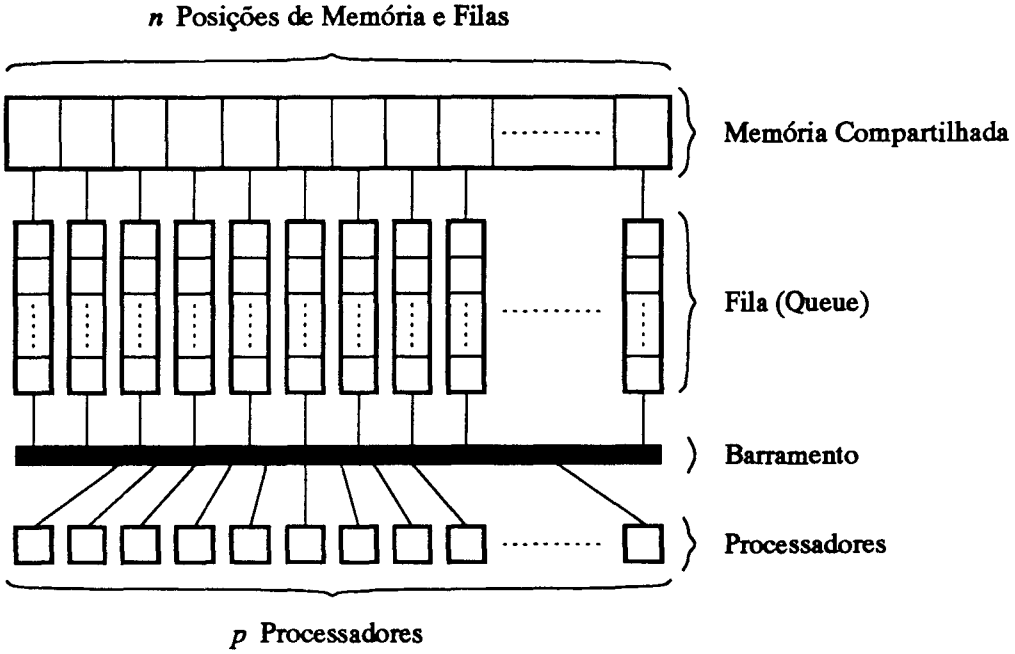


Figura 5.2: Estrutura do modelo QRQW PRAM

[1] *queue*

**Definição 5.1** *O Grau Máximo da Contenção por Memória é igual ao número máximo de processadores que estão lendo ou escrevendo em uma mesma posição de memória em um determinado passo.*

O QRQW PRAM é definido como um modelo de memória compartilhada onde cada processador possui sua própria memória local. Como na maioria dos modelos de memória compartilhada, toda comunicação é feita através da memória compartilhada.

A execução no modelo QRQW PRAM consiste de um conjunto de passos onde cada passo está dividido em subpassos menores:

- **Subpasso de leitura:** $l_i$ células da memória compartilhada são lidas pelo processador $i$.
- **Subpasso de cálculo:** $c_i$ cálculos locais são executados por cada processador $i$.
- **Subpasso de escrita:** $e_i$ valores são escritos em $e_i$ posições da memória compartilhada por cada processador $i$

Note que o modelo é síncrono e portanto entre cada passo existe uma sincronização. Esta divisão em subpassos possibilita um pouco mais de liberdade de programação, já que não existe sincronização entre subpassos. Em particular o modelo permite que o processador realize outras operações enquanto espera que suas leituras ou escritas terminem. Sendo assim, dado que $t_{p_i}$ é o tempo de um passo em um processador $i$ tal que $t_{p_i} = \text{Max}\{l_i, c_i, e_i\}$, seja $t_s = \text{Max}\{t_{p_1}, t_{p_2}, t_{p_3}, \ldots, t_{p_p}\}$ e $k$ o tempo máximo decorrente da contenção por memória, então o custo de um passo neste modelo é dado por $\text{Max}\{t_s, k\}$. Definido o custo de cada passo, as medidas de desempenho $T(n)$ e $C(n)$ são calculadas normalmente.

Além deste modelo, vários outros podem ser definidos através do uso da fila independentemente para leitura e escrita, podendo surgir modelos como: CRQW, ERQW, entre outros.

## 5.6.2 Motivação

A regra de acessos através do uso de fila é bastante próxima da realidade. Várias máquinas paralelas se utilizam deste tipo de regra para solução de seus problemas de acesso concorrente, como podemos verificar na tabela 5.1.

A proposta da regra de fila também se baseou no fato de que ao analisarmos as regras para tratamento de acessos concorrentes propostas no modelo PRAM, podemos facilmente verificar que as mesmas não são muito realistas quando comparadas com máquinas paralelas:

- A regra de **exclusividade** de acessos é extremamente rígida, pois a existência de contenção por memória é comum na maioria das máquinas paralelas. Além disso, os algoritmos resultantes tornam-se um pouco mais complexos, pois estes, em hipótese alguma, poderão conter acessos concorrentes à memória.

- A regra de **concorrência** de acessos, ao contrário da anterior, propõe uma facilidade que é extremamente irreal. Nesta regra não existe custo adicional pela existência de contenção por memória. Os acessos concorrentes podem ocorrer e são contabilizados da mesma forma que os acessos exclusivos, que custam uma unidade de tempo. Ocorre que até o momento não conhecemos uma máquina paralela cujos acessos concorrentes tenham mesmo custo que acessos não concorrentes, e ambos iguais ao de uma operação local. Em contraste a isso, temos que o trabalho do projetista de algoritmos é facilitado, pois o mesmo não necessita se preocupar com custos relacionados com contenção por memória.

| **Regras de Acessos Concorrente em algumas Máquinas** | |
|---|---|
| Kendall Square KSR1 | CRQW |
| Stanford DASH | QRQW |
| MasPar MP-1, MP-2 | |
| roteador global | QRQW |
| Xnet | EREW |
| Thinking Machines CM-5 | |
| rede de dados | QRQW |
| rede de controle | QRQW |
| Intel Paragon | QRQW |
| NCUBE 3 | QRQW |

Tabela 5.1: Regras para tratamento de acessos concorrentes em algumas máquinas paralelas

# Capítulo 6

# Modelo BSP

## 6.1 Descrição do Modelo

### 6.1.1 Descrição Básica

O BSP [Val90] surgiu como proposta de um modelo de computação paralela que não fosse nem arquitetural nem de programação, mas algo que pudesse ser considerado entre as duas áreas, como uma abstração de ambos os lados, ligando-os.

O modelo BSP, em linhas gerais, consiste das seguintes partes:

- Processadores e módulos de memória chamados indistintamente de **componentes.**
- Um **roteador** que transporta mensagens, tipicamente operações de leitura e escrita, ponto a ponto entre os componentes. Mais especificamente, a principal tarefa do roteador é realizar **h-relações**, que são passos do algoritmo onde cada componente envia ou recebe no máximo $h$ mensagens.
- Mecanismos para sincronização dos componentes no estilo **Barreira de Sincronização.**

O esquema de sincronização através de barreira, referido acima, deve ser provido por hardware no modelo BSP. Neste esquema de sincronização os processadores trabalham assincronamente dentro de intervalos de tempo $L$, e a cada final de intervalo há uma parada para sincronização. Apesar de este esquema ser provido por *hardware*, o programador pode, via *software*, desabilitar seu uso para que determinados processos, como processos estritamente seqüenciais, não sejam atrasados desnecessariamente. Uma vez desabilitado, os processos podem se utilizar de envios de mensagens caso desejem sincronizar.

O modelo BSP se propõe a ser mais realista contabilizando alguns custos existentes nas máquinas paralelas através de parâmetros que são usados na avaliação de complexidade de seus algoritmos. Estes parâmetros são:

- O número de componentes, $p$.
- O tempo mínimo, $L$, entre operações de sincronização que pode ser controlado via *software*, de acordo com o tamanho das operações, com o objetivo de conseguir a máxima utilização dos processadores. Este período $L$ é intimamente relacionado com a latência da rede, já que $L$ deve ser pelo menos igual ao tempo requerido para acessos não locais [McC94b].
- A razão $g$ entre o número total de operações elementares de todo sistema por unidade de tempo e o número total de palavras de dados transportados pelo roteador por unidade de tempo. Este parâmetro é melhor entendido como sendo o inverso da largura de banda de comunicação do sistema. Em geral, $g$ é fixado de tal forma que possamos, em uma análise, associar um custo de $gh$ para uma h-relação.
- A latência ou *startup*, $s$, está ligada à velocidade dos processadores, pois quanto mais rápidos forem os processadores mais rápido eles conseguirão responder às requisições, diminuindo desta forma a latência do sistema.

A execução em um computador BSP consiste de um conjunto de **superpassos**. Em cada superpasso, tarefas como operações locais, transmissão de mensagens e recebimento de mensagens são atribuídas a cada componente. Ao final de $L$ unidades de tempo é executada uma sincronização global (barreira de sincronização) para verificar se todos os componentes encerraram suas tarefas; caso nem todos os processadores tenham terminado, outras $L$ unidades de tempo são alocadas para o superpasso.

Agora suponha, para qualquer componente, que $O_l$ é o número máximo de operações locais executadas (ou $O_l = \text{Max}\{O_{l_i}\}$, $i = 1, 2, ..., p$), $M_e$ é o número máximo de mensagens enviadas (ou $M_e = \text{Max}\{M_{e_i}\}$, $i = 1, 2, ..., p$) e $M_r$ é o número máximo de mensagens recebidas(ou $M_r = \text{Max}\{M_{r_i}\}$, $i = 1, 2, ..., p$); então a complexidade de um superpasso é $\text{Max}\{L, O_l, gM_e, gM_r\}$.

### 6.1.2 Acessos Concorrentes

Como podemos notar, o BSP é um modelo de memória distribuída onde não interessa qual é a particular rede de interconexão entre os processadores (vide figura 6.1). O que faz parte do modelo é o roteador, que deve transportar mensagens ponto a ponto entre os componentes, independente da topologia da rede de interconexão e sem requerer o uso de mecanismos especiais para combinação de acessos concorrentes; o objetivo é diminuir a latência provocada pela contenção por memória. O fato de não existir nenhum mecanismo de *hardware* para combinação destas mensagens não impede que consigamos melhorar o atraso provocado pela contenção.

Quando nenhum mecanismo de combinação de mensagens concorrentes está disponível, a latência de um acesso concorrente será linearmente proporcional ao número de processadores que participam deste acesso, já que eles serão atendidos um por vez. O grande problema é que este atraso pode ser inaceitável caso o número de processadores seja muito grande.

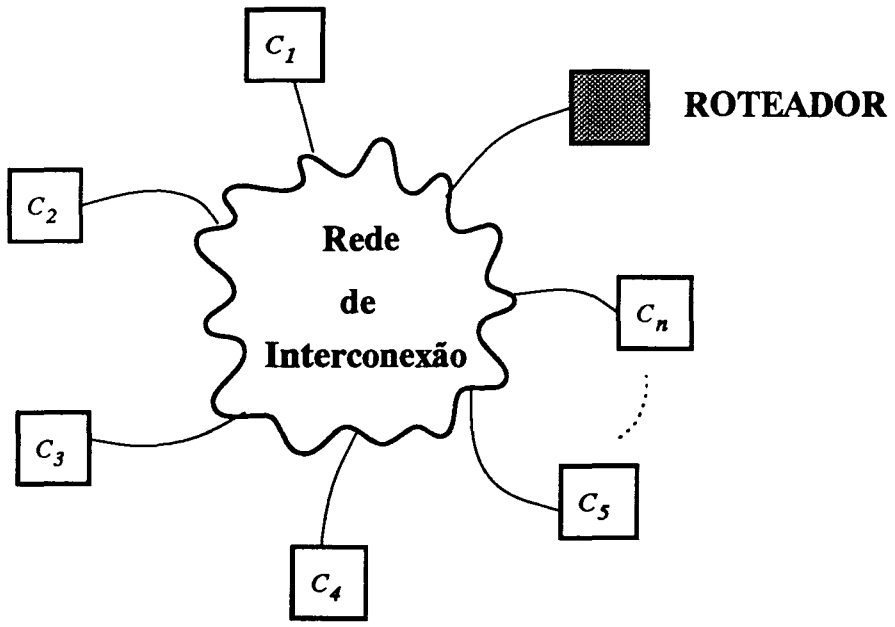


Figura 6.1: Macro estrutura do modelo BSP.

Umas das soluções existentes para implementação de acessos concorrentes, em tempo não linearmente proporcional ao número de processadores, é o uso de redes de combinação[1] [GGK+83, Ran87], que são mecanismos de *hardware* que podem combinar mensagens concorrentes ou replicar mensagens para diversos processadores destino, e não somente enviar mensagens ponto a ponto entre os processadores.

Uma rede de combinação nada mais é do que ligações físicas entre os componentes. O problema do uso deste tipo de solução é que, em geral, não conhecemos o padrão de comunicação entre os componentes, e desta forma não podemos usar uma rede de combinação fixa (com poucas ligações). Sendo assim, temos que fazer uso de mais ligações para permitir uma maior flexibilidade de comunicação. Tal flexibilidade, porém, somente é alcançada a um custo muito alto devido ao grande número de ligações necessárias.

A proposta apresentada por Valiant [Val92], e que ele supõe estar implementada no modelo BSP, é o uso de um mecanismo de combinação de mensagens concorrentes em duas fases implementado via *software*, e conseqüentemente com custos menores. Como

[1] *combining networks*

um determinado processador somente consegue atender requisições em tempo linearmente proporcional ao número de requisições concorrentes, o mecanismo em duas fases procura, em uma primeira fase, espalhar, via uma função de *hashing*, os acessos concorrentes a mesmas posições de memória em processadores intermediários, os quais têm por objetivo combinar as mensagens recebidas em uma só; a segunda fase consiste simplesmente de enviar a mensagem combinada para o destino apropriado. Podemos visualizar este mecanismo com uma árvore associada a cada endereço de memória, onde o próprio endereço é sua raiz e quanto maior for o número de acessos concorrentes maior sua profundidade.

O modelo BSP possui esquemas de gerenciamento automático de memória e comunicação, mas o uso destes não é obrigatório. O programador pode não fazer uso de tais esquemas objetivando, por exemplo, melhorar o desempenho da execução de um algoritmo com relação ao tempo.

### 6.1.3 Folga Paralela

O desempenho de algoritmos projetados no modelo BSP pretende ser ótimo dentro de fatores multiplicativos constantes e independente do número de processadores. Isto significa que os algoritmos projetados para $v$ processadores virtuais podem ser implementados em uma máquina com $p$ processadores efetivos, onde $p < v$, tendo fatores constantes de diferença no tempo de execução. Tal fato é possível graças a folga paralela[2], a qual é usada para esconder a latência decorrente de operações de grande latência.

A exploração desta diferença entre o número de processadores virtuais ($v$) e processadores efetivos ($p$) pode permitir que se atinja a otimalidade de execução de algoritmos, mas caso tivéssemos $v = p$ esta otimalidade não seria possível em todos os casos. A existência de menos processadores efetivos permite que estes estejam sempre ocupados. Suponha a execução de uma operação de longa latência como, por exemplo, leitura de memória, que na prática chega a custar até 10 vezes o tempo de uma operação comum. Se $p = v$, o mapeamento dos processadores virtuais para os efetivos terá uma relação de 1:1, logo caso um processador necessite executar uma leitura ele ficará latente até que o dado esteja disponível. Se $p < v$, cada processador efetivo responderá pela execução de $\lceil \frac{v}{p} \rceil$ processadores virtuais. Neste caso, quando um processador efetivo executa uma operação de longa latência, ele pode usar o tempo da latência para executar operações escalonadas para outros processadores virtuais, escondendo desta maneira a latência da operação. A isso chamamos de exploração da folga paralela.

O desempenho ótimo dentro de fatores multiplicativos refere-se ao fato de que, se garantirmos que $v = p \log p$, podemos garantir que a razão $\frac{v}{p}$ é proporcional a razão $\frac{t_p}{t_v}$, onde $t_p$ é o tempo de execução do algoritmo implementado e $t_v$ o tempo de execução do algoritmo no modelo BSP [Val90]. Esta proporcionalidade significa que quanto menos

[2] *parallel slackness* - diferença existente entre o número de processadores virtuais e efetivos

processadores efetivos existirem maior será o tempo de execução do seu algoritmo, mas permanece a equivalência entre as duas razões. Caso tenhamos $v = p$, temos que fazer algumas modificações no algoritmo para, se possível, utilizar mais processadores virtuais.

## 6.2 Motivação

A proposta do modelo BSP surgiu como uma tentativa de se conseguir uma padronização no mundo da computação paralela. Valiant [Val90, Val93] argumenta que o sucesso da computação seqüencial se deve, em grande parte, ao surgimento da arquitetura *von Neumann*. Esta arquitetura e o modelo que ela supõe conseguiram se portar, segundo Valiant, como uma ponte ligando as áreas de *hardware* e *software*.

A partir destas observações realizadas no mundo da computação seqüencial, Valiant resolveu propor o modelo BSP, argumentando que este poderia representar na computação paralela o mesmo papel da arquitetura *von Neumann* para a computação seqüencial.

Em [Val93], Valiant nos apresenta alguns fatores (sumarizados abaixo) que mostram o porquê da necessidade de tal modelo ponte na computação paralela:

- Grande parte dos problemas tratáveis computacionalmente possuem um grau de paralelismo implícito. Um modelo de computação nos ajudaria a explorar tal paralelismo, que difere de problema para problema.
- Não existe nada que impeça que a exploração deste paralelismo gere algoritmos transportáveis entre as diversas arquiteturas existentes, alcançando assim, uma computação paralela de propósito geral.
- É pouco provável que um modelo de computação paralela padrão surja apenas de considerações de linguagens. Note que a arquitetura *von Neumann* surgiu sem a existência de uma linguagem padrão.
- Também é pouco provável que um modelo de computação paralela surja apenas de considerações de *hardware*. As rápidas mudanças tecnológicas, que podem ocorrer no mercado, forçam a existência de um modelo que seja de mais alto nível.
- Uma vez proposto, um modelo ponte deve permitir que antecipemos restrições tecnológicas sem que percamos um bom nível de programabilidade.
- Não existe nenhum impedimento fundamental que nos mostre a impossibilidade de padronização da computação paralela através de um único modelo de computação.

Considerando todos estes pontos, o modelo BSP foi proposto como um modelo ponte para a computação paralela que influencia tanto a área de *hardware*, com a proposta de uma arquitetura (BSP *computer*), quanto a de *software*, com métodos de programação.

## 6.3 Algoritmos

Antes de falarmos nos algoritmos é oportuno falarmos um pouco de formas de avaliação de algoritmos BSP.

A forma mais usada é a utilização do conceito de contagem de superpassos desprezando constantes multiplicativas e termos de menor grau (análise assintótica).

O conceito de notação assintótica pode ser usado, principalmente, devido a existência de um tipo de sincronização que, neste caso, é feita por barreira. Para analisar um algoritmo temos que descobrir o custo de cada superpasso e contar o número de superpassos existentes. Podemos, se necessário, expressar o custo deste superpasso em termos dos parâmetros do modelo BSP.

Descreveremos a seguir um algoritmo para este modelo. Nosso objetivo é mostrar como é um algoritmo BSP. Desta forma, escolhemos um problema bastante simples para que as características do modelo não fossem ofuscadas pela dificuldade de um problema.

### 6.3.1 Difusão de Dados

Um dos problemas mais comuns e mais simples em computação paralela é o problema de difusão[3]. O problema consiste em enviar um dado, inicialmente localizado em um processador, para todos outros através de transmissões de mensagens.

O objetivo é resolver o problema da forma mais rápida possível. Para tanto, temos que evitar alguns fatores adversos como o fato de um determinado processador receber o dado mais de uma vez.

Valiant [Val90] nos apresenta um algoritmo para difusão onde o processo de comunicação se comporta como uma árvore de grau $d$. Ele nos mostra que o problema de difusão pode ser resolvido através desta árvore em tempo $d\ g\ log_d p$.

Antes de apresentarmos o algoritmo em si, precisamos definir a seguinte operação:

- SEND(VALOR,DESTINO): Consiste em enviar um VALOR para um processador DESTINO. Faz-se necessário devido ao BSP ser um modelo de memória distribuída.

Na figura 6.2 apresentamos uma formalização do algoritmo BSP para difusão, com apenas uma pequena modificação. No algoritmo original os processadores ficam parados após enviar os dados para seus respectivos receptores; na nossa variante, os processadores permanecem enviando dados em todos os passos até que todos os processadores tenham recebido o dado. A diferença de desempenho entre estas duas variações não passa de valores constantes.

No algoritmo da figura 6.2 existem $d-1$ SENDs, cada um corresponde a uma 1-relação, e que conseqüentemente possui custo $g$. Como cada SEND equivale a uma h-relação,

[3] *broadcast*

**Entrada:** Um processador, $P_1$, contendo o dado a ser enviado para os outros processadores. O grau, $d$, da árvore a ser usada e as $n$ posições de memória onde os dados devem ser colocados. O vetor $A$ é local a cada processador

**Saída:** Ao final de $log_d p$ passos, todos os processadores conterão $n$ cópias do dado inicial nos $p$ vetores locais.

```
(1)              for i := 1 to p do in parallel
(2)                 k := 1
(3)                 while k <= p/d do
(4)                    if (i <= k) then
(5)                       for j:=1 to d-1 do
(6)                          SEND(DADO, P_(i+(d-j)k))
(7)                    k := k * d
(8)                 for j := 1 to n/p do
(9)                    A[j] := DADO
```

Figura 6.2: Algoritmo BSP para difusão de um dado

existe uma barreira de sincronização entre cada um. Ainda olhando para a função SEND, podemos afirmar que cada um gasta tempo constante, pois como neste modelo a rede de interconexão não é importante, podemos contabilizar o custo de um SEND igualmente para qualquer nó da rede. O laço representado por (3)-(7) é executado $log_d p$ vezes, já que este simula uma árvore $d$-ária, já o laço (8)-(9) é executado $\frac{n}{p}$ vezes. Daí dizemos que o algoritmo possui tempo total, $T(n)$, proporcional a $(d-1)\ g\ log_d p + \frac{n}{p}$.

Note que a forma de apresentação deste algoritmo não o torna diferente dos algoritmos PRAM. Tal diferença começa a ficar clara quando fazemos suposições sobre os valores dos parâmetros do modelo.

Desta forma, podemos verificar que como cada SEND possui custo $g$ (1-relação) e como são executados $(d-1)$ destes, o ideal seria que o valor de $L$ fosse $\Theta(dg)$, pois desta forma o intervalo entre sincronizações não seria nem pequeno demais nem grande.

O outro objetivo é alcançarmos um tempo ótimo para o algoritmo que neste caso é dado por $O(\frac{n}{p})$. Observe que isto é possivel se tivermos $O(d\ g\ log_d p + \frac{n}{p}) = O(\frac{n}{p})$, o que nos mostra que o grau da árvore, $d$, a ser usada deve ser $O((\frac{n}{p\ g\ log\ p}) \times log(\frac{n}{p\ g\ log\ p}))$.

No capitulo 8 apresentaremos uma implementação paralela no modelo BSP do algoritmo para Eliminação de Gauss.

# Capítulo 7

# Modelo LogP

## 7.1 Descrição do Modelo

Os modelos de computação paralela são geralmente propostos tentando descobrir o futuro da computação paralela ou influenciar na construção de uma arquitetura que venha a se tornar padrão. O modelo LogP [CKP+93], ao contrário dos outros, tenta resolver o problema atual de desenvolvimento de bons algoritmos para um grande número de máquinas existentes.

O modelo LogP se baseou tanto em uma convergência de arquiteturas paralelas quanto no modelo BSP [Val90] que serviu como ponto de partida. Assim como no modelo BSP, o modelo LogP possibilita a contabilização de fatores existentes nas máquinas paralelas através de parâmetros que são utilizados na expressão da complexidade de um algoritmo. Este porém, não provê nenhum mecanismo de sincronização, caracterizando-se como um modelo totalmente assíncrono.

Antes, porém, de mostrarmos os parâmetros do modelo necessitamos da seguinte definição:

**Definição 7.1** *Custo Adicional*[1] *de Comunicação é o custo decorrente da transmissão ou recepção de informações adicionais necessárias a realização da transmissão ou recepção de mensagens.*

São os seguintes, os parâmetros definidos no modelo LogP:

- $L$, um limite superior para latência;
- $o$, custo adicional de comunicação incorrido por um processador para transmitir ou receber uma mensagem;

[1] *Overhead*

- $g$ (*gap*), que define o intervalo mínimo entre transmissões consecutivas ou recepções consecutivas de mensagens;
- um certo número $P$ de módulos, cada um composto de um processador e sua memória local.

Como podemos verificar pelos parâmetros do modelo, ele não supõe a existência de nenhuma rede de interconexão particular, podendo, desta forma, os processadores estarem organizados em qualquer topologia. Além disso, o modelo supõe que a rede de interconexão tem uma capacidade finita, isto é, no máximo um certo número de mensagens podem estar circulando na rede num dado instante. Esse número é $\lceil \frac{L}{g} \rceil$ para garantir que todas mensagens poderão ser processadas mesmo que destinadas a um único processador. As previsões de desempenho no modelo LogP só são válidas se este limite imposto à rede de interconexão for respeitado. Veja também que a latência individualmente existente em cada mensagem é imprevisível mas é limitada por $L$.

Os proponentes do modelo LogP acreditam que este conjunto de parâmetros dificilmente poderia ser menor caso realmente desejemos uma boa expressão da realidade. O uso de todos os parâmetros está sujeito a cada aplicação e a cada máquina. Em alguns casos alguns parâmetros podem ser desprezados pois não influenciam no comportamento total do algoritmo.

## 7.2 Motivação

Atualmente já podemos observar que diversos fatores tecnológicos estão fazendo com que máquinas paralelas atuais de grande porte possuam características semelhantes [CKP+93, GKLP93].

Por questões estritamente econômicas as máquinas paralelas tendem a possuir no máximo milhares de nós, pois um sistema com mais do que esse limite se tornaria economicamente inviável quando comparado com uma rede de estações de trabalho[2] [Pat93, KFW94]. Além disso, os nós de uma máquina paralela tendem a se tornar praticamente estações de trabalho completas, dado seu poder (vide figura 7.1). Esse poder dos processadores se deve ao crescimento da capacidade de processamento que cresce em torno de 50% ao ano em relação a operações com inteiros e 75% ao ano em relação a operações de ponto flutuante (vide figura 7.2*(a)*); estes processadores tendem a ser não customizados[3], já que os custo de desenvolvimento de processadores tornou-se muito alto.

Além dos processadores as memórias locais também tendem a crescer, mas a uma taxa menor que a dos processadores (vide figura 7.2*(b)*). O *gap* existente entre a velocidade dos

[2] *workstations*
[3] *off-the-shelf*

processadores e das memórias é muito grande e tende a crescer; acredita-se que no ano 2000 a velocidade de uma CPU deverá ser em torno de 10 a 12 vezes maior que a velocidade da memória (vide figura 7.2*(c)*). As memórias locais tendem a estar em torno de centenas de *MegaBytes*. Já a tecnologia de interconexão de nós, apesar de ter evoluído bastante, não consegue acompanhar as crescentes velocidades dos processadores e das memórias, o que nos faz acreditar que a largura de banda continuará baixa em relação a velocidade dos processadores, assim como a latência de comunicação continuará sendo relativamente alta. Quanto a topologia da rede de interconexão, modernas técnicas de roteamento indicam que uma particular forma de organização não será fundamental no futuro das máquinas paralelas. Algumas máquinas que fazem parte desta convergência são: CM-5, Intel iPSC e NCube.

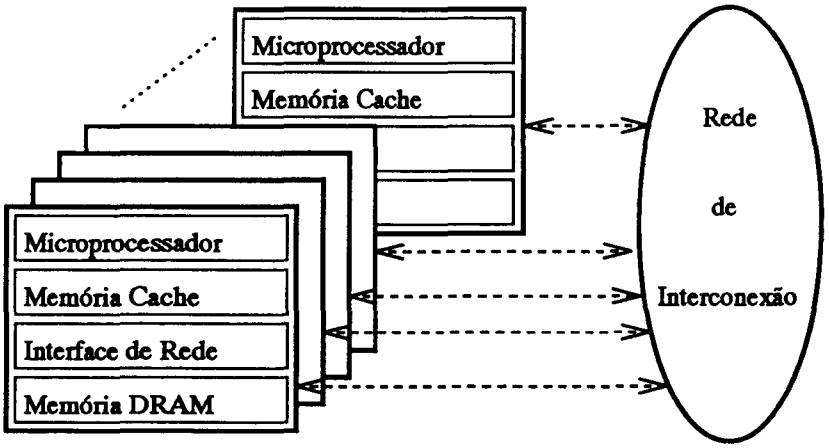


Figura 7.1: Macro estrutura de arquiteturas pertencentes a convergência. Adaptado de [CKP+93].

Em outro trabalho apresentado por McColl [McC94b], pode-se observar afirmações a respeito da convergência de arquiteturas. Segundo McColl, existem atualmente três tipos de arquiteturas paralelas disponíveis comercialmente, que parecem convergir em direção a uma arquitetura comum:

- **Arquiteturas com Memória Distribuída:** Este tipo de arquitetura caminha em direção a uma forma de abstração que permitirá aos usuários visualizarem a memória como um único espaço de endereçamento.
- **Arquiteturas com Memória Compartilhada:** Caminha em direção a uma maior escalabilidade, substituindo sua arquitetura baseada em barramento por redes de interconexão de alto desempenho.

- **Redes de Estações de Trabalho:** As estações de trabalho tendem a se ligar através de redes de alto desempenho onde a latência será bastante reduzida quando comparada com a existente atualmente neste tipo de arquitetura.

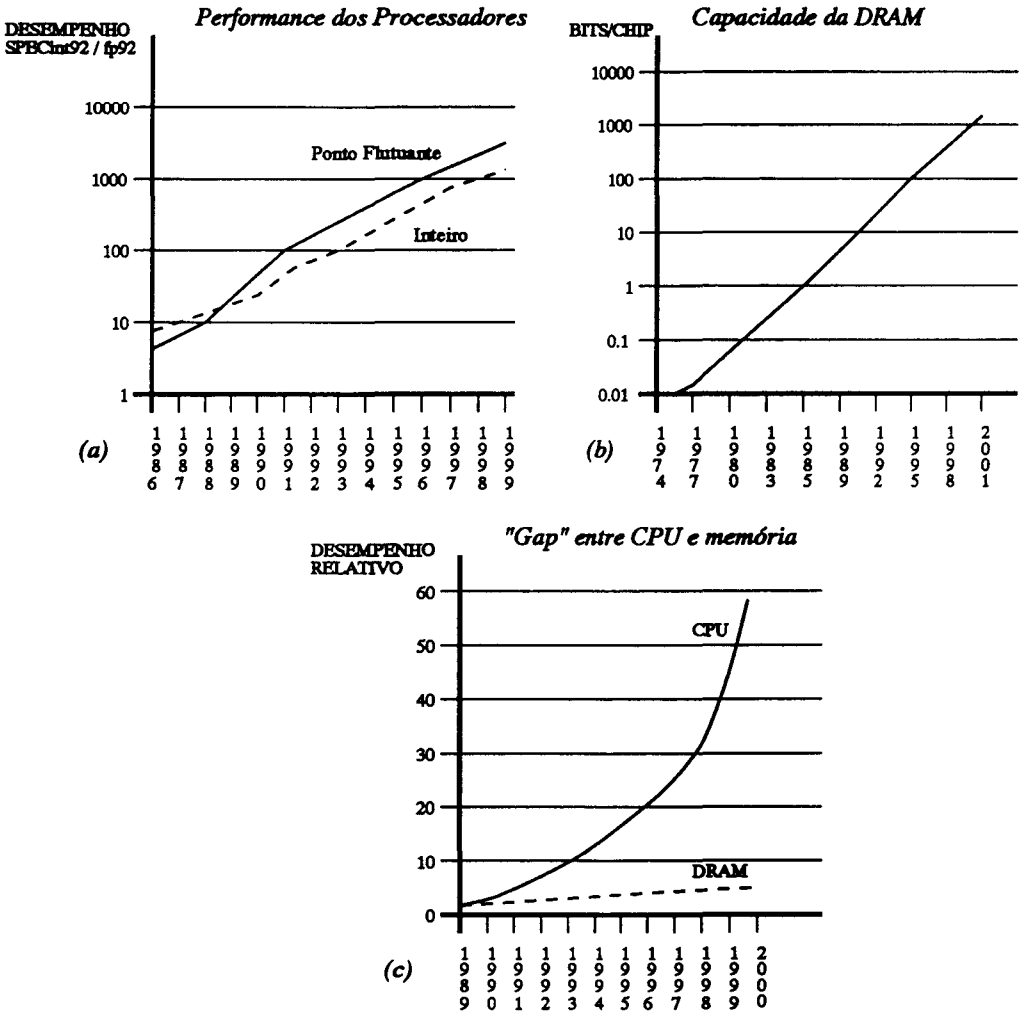


Figura 7.2: Gráficos indicando o desempenho dos processadores e memória. Adaptado de [Pat93]

De uma forma geral, a tendência visualizada por McColl vai ao encontro da visualizada por Paterson em [Pat93]. McColl defende que em poucos anos uma arquitetura padrão surgirá; teremos pares processador-memória conectados ponto a ponto em uma rede de interconexão eficiente que suporte o uso de espaço de endereçamento único.

## 7.3 Algoritmos

O primeiro passo antes de mostrarmos algoritmos em LogP é falarmos de como estes podem ser projetados e analisados.

Diferentemente de outros modelos, não existe uma formalização de um algoritmo LogP. O modelo serve como suporte para que o usuário projete diretamente o programa para a máquina destino. Não podemos, assim como no BSP e outros modelos, definir um algoritmo como um metacódigo em termos de alguns parâmetros que se adaptam a cada situação.

Neste modelo é fundamental que antes de pensarmos em como resolver o problema em si, termos todos parâmetros para a máquina destino, pois nossa forma de agir e toda a estrutura do programa dependerá de seus valores.

Com relação a análise de desempenho do programa, o que temos é um cálculo do tempo de execução, que se propõe a ser preciso, pois dado o valor de cada parâmetro e conseqüentemente como é o comportamento do programa, podemos predizer precisamente o tempo de execução. Novamente não é possível se apresentar uma função geral dos parâmetros, que indique o tempo de execução para qualquer máquina. Existe uma função, mas específica a cada máquina.

### 7.3.1 Difusão de Um Único Dado

A resolução do problema de difusão de dados no modelo LogP se dá de uma maneira completamente diferente da vista no capitulo 6. Como não é possível formalizar um algoritmo genérico, pois o que temos são praticamente técnicas que nos ajudam a resolver o problema, descreveremos a solução para um exemplo particular e mostraremos como essa solução inicial nos ajuda na resolução geral.

O problema de difusão consiste em enviar um dado (ou um conjunto de dados) inicialmente localizado em um processador inicial $P_0$ para todos os outros $P-1$ processadores no menor tempo possível. Pode-se notar que é ideal garantirmos que nenhum processador receberá o dado mais de uma vez, já que isto seria um custo adicional desnecessário.

O primeiro passo é visualizar a estrutura de comunicação entre estes processadores. Neste problema temos a figura 7.3 que nos mostra uma possível estrutura de comunicação. Podemos notar que nesta figura cada nó está associado a um valor, em função dos parâmetros $L$, $o$ e $g$ do sistema, o que nos mostra claramente que esta estrutura está extremamente ligada as arquiteturas alvo.

A figura também nos mostra que estamos supondo que o dado a ser difundido já se encontra no processador $P_0$ no tempo 0. A partir deste instante o processador $P_0$ já está apto a enviar o dado para outro processador e pode reenviar o dado a outros processadores a cada intervalo de $g$ unidades de tempo, conseqüentemente $P_0$ enviará um dado nos tempos

0, $g$, $2g$, $3g$, e assim por diante. Quando o dado sai de $P_0$ no tempo 0 ele estará disponível no seu destino no tempo $L + 2o$, correspondendo a um custo adicional de comunicação, $o$, para o envio; uma latência, $L$, correspondendo ao tempo que o dado leva do nó fonte para o nó destino; e finalmente outro custo adicional de comunicação para o recebimento do dado.

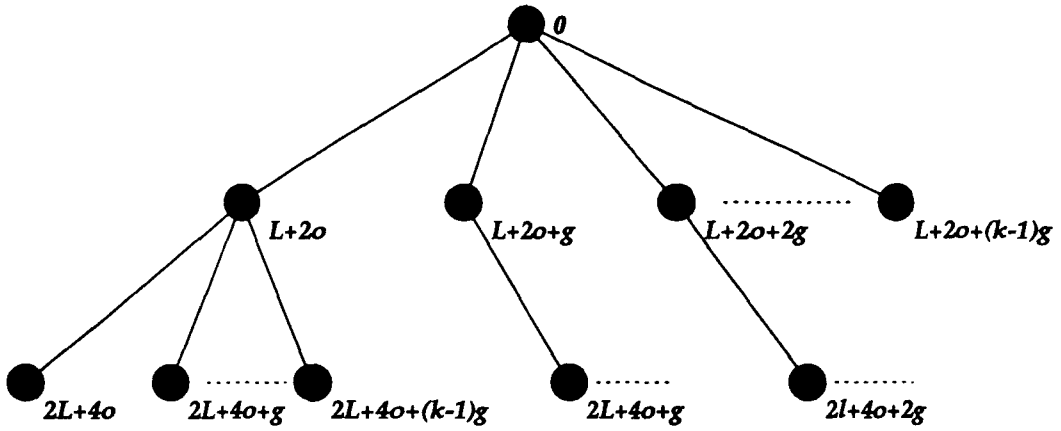


Figura 7.3: Árvore para difusão de um único dado

Podemos generalizar estes tempos exemplificados para o processador $P_0$ para qualquer outro processador. Suponha que um processador recebeu o dado a ser difundido no tempo $a$, então este mesmo processador poderá enviar este dado para outros processadores nos tempos $a$, $a + g$, $a + 2g$ e assim sucessivamente. Similarmente, se um processador envia um dado no tempo $b$ ele estará disponível em seu destino no tempo $b + L + 2o$.

Logo para resolvermos o problema de difusão no modelo LogP temos que primeiro construir a árvore da figura 7.3 e substituir os valores dos parâmetros para a máquina alvo. Com a árvore construída temos que escolher os $P$ menores nós desta árvore. O tempo de execução do algoritmo será igual ao nó escolhido de maior valor.

### 7.3.2 Soma

O problema de soma consiste em, dado um conjunto de valores, obter a soma destes valores no menor tempo possível.

A solução clássica em arquiteturas de memória distribuída consiste em espalhar os valores pelos processadores e executar a soma, de tal forma que cada processador ao executar sua soma local passe o valor a outro processador. Neste tipo de solução temos que garantir que nenhum dos processadores transmitirá o resultado de sua soma a mais de um processador, pois caso isto ocorra seu resultado poderá ser somado mais de uma vez.

Podemos observar que este problema possui similaridades como o problema de difusão, pois o que temos aqui praticamente o processo inverso. Se tomarmos a árvore de soma veremos que esta tem praticamente a mesma estrutura da árvore de difusão. A cada soma executada em um processador, este transfere seu resultado ao nó pai.

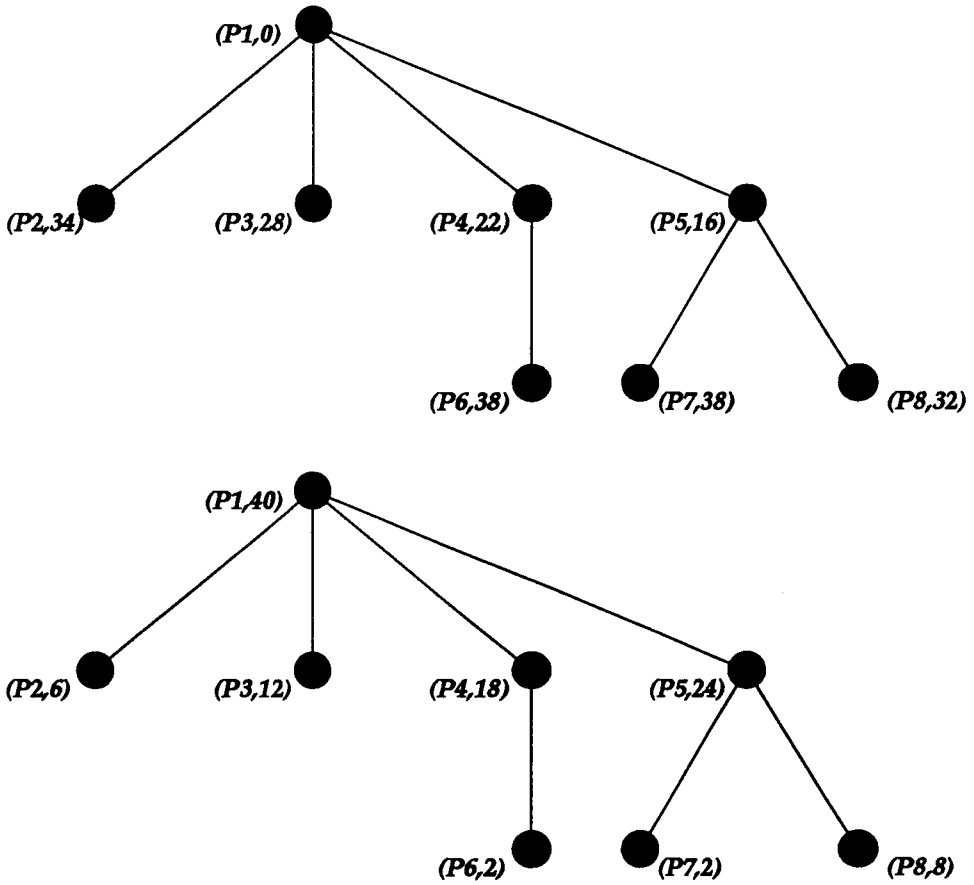


Figura 7.4: Método para transformação da árvore de difusão com parâmetros $L = 8$, $o = 4$, $g = 6$ e $P = 8$ na árvore de soma com parâmetros: $L = 7$, $o = 4$, $g = 6$ e $P = 8$; e tempo $T = 40$. Os nós da árvore superior (difusão) são representados por $(Px, y)$ onde $x$ indica o número do processador e $y$ o tempo restante para uma possível difusão a partir deste nó (o tempo cresce de cima para baixo). Os nós da árvore inferior (soma) também possuem representação dada por $(Px, y)$ mas neste caso o valor de $y$ representa o tempo gasto na soma até este nó (o tempo cresce de baixo para cima).

Existem duas abordagens do problema de soma, sendo uma o inverso da outra: a primeira que pretende somar o maior número de valores, $n$, no tempo dado e uma segunda abordagem que pretende descobrir qual o menor tempo necessário para se somar $n$ números dados. Em [CKP+93, KSSS93] a primeira abordagem é mostrada como que introdutória

para a solução da segunda a qual podemos chamar de mais geral. Mostraremos a primeira abordagem detalhadamente.

Antes de descrevermos os algoritmos para as duas abordagens do problema, precisamos apresentar um resultado de Karp *et. al* [KSSS93], sobre a similaridade da árvore de difusão e a de soma:

**Teorema 7.1** *A árvore de soma ótima é igual ao inverso árvore de difusão de um único dado, com relação ao tempo.*

O teorema 7.1, nos mostra que dada uma árvore ótima de difusão de um único dado cujo tempo de execução é $T$, podemos construir a árvore ótima de soma para o mesmo tempo $T$, simplesmente substituindo o tempo de cada nó, $T_n$ (mostrado na figura 7.3), por um nó com tempo $T - T_n$. A única diferença existente é que, ao considerarmos a árvore inversa da difusão para obtermos a de soma, temos que contabilizar o tempo necessário para execução da soma no nó (vide figura 7.4).

Para resolvermos a primeira abordagem do problema de soma no modelo LogP, necessitamos, como em qualquer outra resolução, dos valores dos parâmetros $L$, $o$, $g$, $P$ e evidentemente o tempo $T$ que temos para trabalhar. Com estes valores disponíveis podemos mostrar qual o comportamento da resolução através da árvore ótima de soma. Especificamente neste exemplo, faremos a suposição de que uma soma local a um processador pode ser efetuada em uma unidade de tempo.

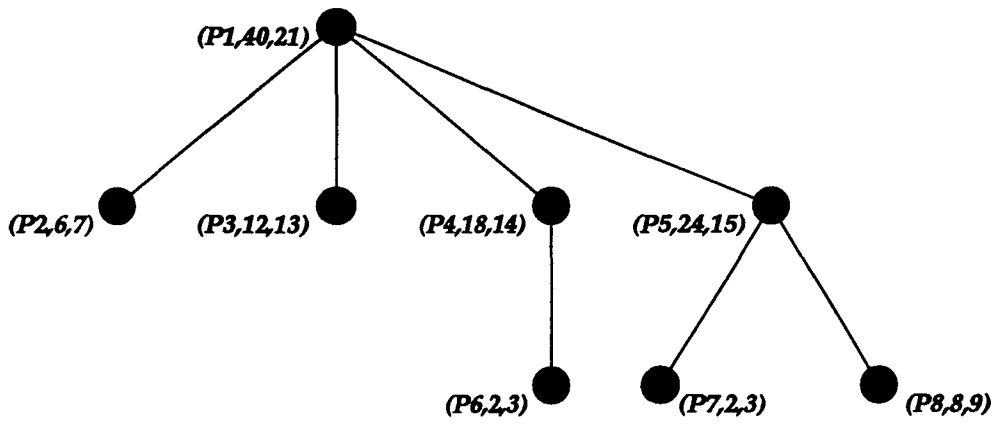


Figura 7.5: Árvore de comunicação ótima para soma com os mesmos parâmetros da figura 7.4. Neste exemplo os nós da árvore soma são representados por $(Px, y, z)$; note que os parâmetros $x$ e $y$ possuem mesmo significado dos da árvore de soma da figura 7.4. O parâmetro $z$ representa o número máximo de valores que podem ser somados pelo nó localmente.

Suponha como exemplo que os valores dos parâmetros do modelo LogP para uma determinada máquina sejam: $L = 7$, $o = 4$, $g = 6$ e $P = 10$; suponha também que o valor do tempo disponível para soma é $T = 40$. A partir destes valores temos que construir a árvore de soma (a partir da similar de difusão), pois esta nos mostra a forma de comunicação entre os processadores. A figura 7.5 nos mostra esta árvore, onde os valores internos a cada nó representam o tempo em que cada um envia a soma de seus resultados ao seu nó pai.

Dada a árvore de comunicação ótima da figura 7.5, podemos calcular quantos valores poderão ser somados no tempo dado. Nos processadores representados por nós folhas na árvore de comunicação temos que cada processador consegue somar $t_p + 1$ valores, onde $t_p$ é o tempo interno de cada nó. No exemplo apresentado, temos que os processadores $P_2$, $P_3$,$P_6$, $P_7$ e $P_8$ conseguem somar respectivamente 7, 13, 3, 3 e 9 valores. Já os processadores representados por nós internos na árvore de comunicação temos que cada processador consegue somar $t_p - f_p(o + 1) + 1$, onde $f_p$ representa o número de filhos do nó na árvore de comunicação. Note que o custo de cada recebimento de um valor do nó filho é $o + 1$, pois necessita-se de uma unidade a mais de tempo para se executar a soma do valor recebido. No exemplo apresentado temos que os processadores $P_1$, $P_4$ e $P_5$ conseguem somar respectivamente 21, 14 e 15 valores.

Na figura 7.6 mostramos uma explosão de alguns nós onde pode ser verificado o porquê das fórmulas apresentadas.

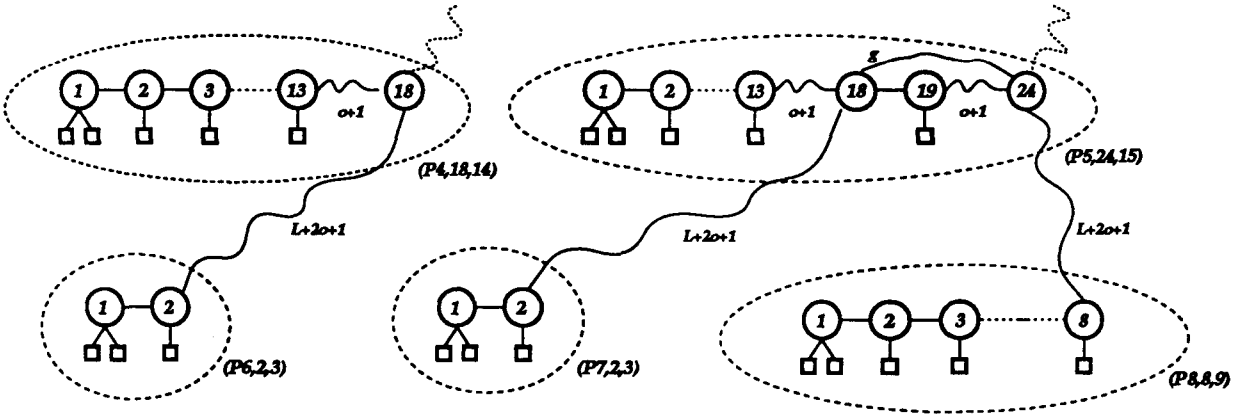


Figura 7.6: Expansão dos nós $P_4$, $P_5$, $P_6$, $P_7$ e $P_8$ da árvore de comunicação apresentada na figura 7.5

Concluindo, podemos dizer que o número máximo de valores que podem ser somados no exemplo apresentado é igual a 85, que equivale a soma dos valores que podem ser somados em cada processador.

A partir da solução da primeira abordagem do problema podemos facilmente mostrar

como resolver a segunda abordagem. Aqui também temos todos os parâmetros do modelo, mas ao invés de desejarmos o número máximo de elementos que podem ser somados em um tempo dado, queremos descobrir o tempo mínimo para se executar a soma de um conjunto de valores dados.

Neste problema temos três casos relacionados com o tamanho do conjunto de valores a ser somado:

- **O tamanho do conjunto é menor que o tamanho ótimo que pode ser somado pela árvore de comunicação:** Neste caso podemos resolver o problema de duas maneiras: a primeira seria utilizar uma subárvore ótima que fosse suficiente para fazer a soma do conjunto de valores; a segunda, e a que acreditamos ser mais precisa, seria construir uma árvore de comunicação utilizando menos processadores com um tempo menor, da mesma forma que fizemos no exemplo anterior.

- **O tamanho do conjunto é igual ao tamanho ótimo que pode ser somado na árvore de comunicação:** Aqui basta que usemos a árvore diretamente. Neste caso, o tempo mínimo para se somar o conjunto de valores será, evidentemente, o usado para resolver a primeira abordagem do problema.

- **O tamanho do conjunto é maior que o tamanho ótimo que pode ser somado pela árvore de comunicação:** Neste caso temos que nos utilizar da árvore ótima dividindo o excesso do número máximo que pode ser somado pela árvore entre todos os processadores de forma a conseguir um mínimo atraso.

# Capítulo 8

# Estudos de Casos

## 8.1 Transformada Rápida de Fourier

### 8.1.1 Definição

Uma das maiores descobertas de algoritmos rápidos da história da computação foi a **Transformada Rápida de Fourier**[CT65]. Introduzida em 1965 por Cooley e Tukey, sua descoberta proporcionou um decréscimo muito grande do número de operações necessárias ao cálculo da transformada de Fourier. Em geral, o uso da TRF[1] diminui o número de operações executadas de $O(n^2)$ para $O(n\ log\ n)$.

O impacto da descoberta desta técnica se deve principalmente ao fato de a transformada de Fourier ser aplicável em diversas áreas como: problemas aritméticos comuns, processamento de sinais, física qüântica, teoria da probabilidade, ótica, sistemas lineares, entre outros. Além disso a Transformada Rápida de Fourier possibilitou a aplicação da transformada de Fourier na resolução de vários outros problemas que de outra forma eram quase que intratáveis. Atualmente a TRF é um dos algoritmos aritméticos mais usados em toda computação [Sed88].

Na demonstração do algoritmo utilizaremos um problema aritmético comum de multiplicação de polinômios e mostraremos como TRF pode ser usada para diminuir a complexidade da operação de $O(n^2)$ para $O(n\ log\ n)$.

### 8.1.2 Multiplicação de Polinômios e TRF

Em álgebra existem quatro operações fundamentais: adição, subtração, multiplicação e divisão. Estas operações podem ser aplicadas, principalmente, em três diferentes domínios: inteiros, polinômios e matrizes. O campo de estudo de algoritmos eficientes para essas

[1]Transformada Rápida de Fourier; em inglês *FFT, Fast Fourier Trasform*

operações é chamado de **álgebra algorítmica** [Yap94]; mostraremos aqui a importância da TRF para este campo.

Para exemplificarmos uma aplicação da TRF, usaremos um dos problemas básicos em álgebra algorítmica: multiplicação de polinômios. Mostraremos inicialmente a ligação existente entre este problema e a TRF de uma maneira informal, mostrando uma boa formalização desta ligação no final da seção.

De um modo geral, um polinômio em $x$ é uma função do tipo:

$$P(x) = \sum_{i=0}^{n} a_i x^i,$$

onde os valores de $a_i$, para $i = 0, 1, 2, 3, \ldots, n-1, n$, são chamados coeficientes. O grau de um polinômio é dado por $n$.

Computacionalmente falando, os polinômios podem ser representados de duas formas: representação por coeficientes e representação por valores de pontos.

A representação por coeficientes de um polinômio $P(x)$ é dada por um vetor $A$ de todos os coeficientes do polinômio, sejam estes zero ou não, logo $A = [a_0, a_1, a_2, \ldots, a_{n-1}, a_n]$. Já a representação por valores de pontos é dada por um conjunto de $n+1$ pontos distintos entre si, onde $n$ é o grau do polinômio, logo o conjunto é dado por:

$$\{(x_0, y_0), (x_1, y_1), (x_2, y_2), \ldots, (x_{n-1}, y_{n-1}), (x_n, y_n)\}$$

onde $y_i = P(x_i)$.

Pode-se provar facilmente que só existe um polinômio de grau $n$ que passa por $n+1$ pontos distintos. Para tanto supomos que existem dois e chegamos a uma contradição aplicando um método de interpolação. Daí podemos dizer que este tipo de representação pode ser mapeado em um polinômio. Como exemplo suponha um polinômio de grau 1 do tipo $a_0 + a_1 x$. Note que este polinômio nada mais é do que uma reta no plano, e sabemos que só existe uma reta passando por dois pontos distintos entre si. Note também que são necessários 2 pontos para a representação de um polinômio de grau 1.

O uso destas representações depende das características de cada aplicação, mas de qualquer forma podemos transformar uma representação em outra, quando assim desejarmos, através do uso dos métodos de avaliação e interpolação.

Quando desejamos transformar a representação por coeficientes para a representação de valores de pontos podemos utilizar a regra de Horner como método de avaliação. Tal regra é descrita pela seguinte expressão:

$$P(x_0) = a_0 + x_0(a_1 + x_0(a_2 + \ldots + x_0(a_{n-1} + x_0(a_n))\ldots))$$

Esta regra consegue avaliar um polinômio em um determinado ponto em tempo $O(n)$.

Como são necessários $n+1$ pontos para representação por valores de pontos de um polinômio de grau $n$, necessitamos fazer $n+1$ avaliações, o que nos dá um tempo total de $O(n^2)$.

Na transformação da representação por valores de pontos para a por coeficientes, utilizamos a fórmula de Lagrange, dada por:

$$
\begin{aligned}
P(x) = {} & Y_0 \left(\frac{x-x_1}{x_0-x_1} \quad \frac{x-x_2}{x_0-x_2} \quad \frac{x-x_3}{x_0-x_3} \quad \cdots \quad \frac{x-x_{n-1}}{x_0-x_{n-1}} \quad \frac{x-x_n}{x_0-x_n}\right)+ \\
& Y_1 \left(\frac{x-x_0}{x_1-x_0} \quad \frac{x-x_2}{x_1-x_2} \quad \frac{x-x_3}{x_1-x_3} \quad \cdots \quad \frac{x-x_{n-1}}{x_1-x_{n-1}} \quad \frac{x-x_n}{x_1-x_n}\right)+ \\
& Y_2 \left(\frac{x-x_0}{x_2-x_0} \quad \frac{x-x_1}{x_2-x_1} \quad \frac{x-x_3}{x_2-x_3} \quad \cdots \quad \frac{x-x_{n-1}}{x_2-x_{n-1}} \quad \frac{x-x_n}{x_2-x_n}\right)+ \\
& \qquad \vdots \\
& Y_{n-1} \left(\frac{x-x_0}{x_{n-1}-x_0} \quad \frac{x-x_1}{x_{n-1}-x_1} \quad \frac{x-x_2}{x_{n-1}-x_2} \quad \cdots \quad \frac{x-x_{n-2}}{x_{n-1}-x_{n-2}} \quad \frac{x-x_n}{x_{n-1}-x_n}\right)+ \\
& Y_n \left(\frac{x-x_0}{x_n-x_0} \quad \frac{x-x_1}{x_n-x_1} \quad \frac{x-x_2}{x_n-x_2} \quad \cdots \quad \frac{x-x_{n-2}}{x_n-x_{n-2}} \quad \frac{x-x_{n-1}}{x_n-x_{n-1}}\right)
\end{aligned}
$$

O resultado obtido com o uso desta regra é um polinômio na forma de coeficientes. A execução desta fórmula gasta tempo $O(n^2)$.

A representação por coeficientes é aconselhável quando os polinômios que queremos representar possuem muitos coeficientes com valor zero, pois podemos usar uma lista ligada onde somente os valores dos coeficientes diferentes de zero seriam armazenados. Tal representação permite que somemos dois polinômios em tempo $O(n)$, onde $n$ é o grau do polinômio de maior grau, pois basta somarmos os coeficientes dos termos de mesmo grau. Dados dois vetores $A = [a_0, a_1, a_2, \ldots, a_{m_0-1}, a_{m_0}]$ e $B = [b_0, b_1, b_2, \ldots, b_{m_1-1}, a_{m_1}]$, o vetor resultante da soma de $A+B$ é dado por $S = [s_0, s_1, s_2, \ldots, s_{n-1}, s_n]$, onde $s_i = a_i + b_i$ para $i = 0, 1, 2, 3, \ldots, n-1, n$ e onde $n = max\{m_0, m_1\}$.

Um problema com este tipo de representação é que o método básico para multiplicação de polinômios é feito em tempo $O(n^2)$. Dados dois vetores $A$ e $B$ de graus respectivamente $m_0$ e $m_1$, o método básico se dá através multiplicação de cada elemento de $A$ por todos os elementos de $B$, resultando um vetor $R = \{r_0, r_1, r_2, \ldots, r_{n-1}, r_n\}$, onde $r_i = \sum_{j=0}^{i} a_j b_{i-j}$ e onde $n = m_0 + m_1$.

Já a representação de um polinômio por valores de pontos é excelente para operações tanto de soma quanto de multiplicação de polinômio, mas requer $n+1$ pares de pontos para a representação de um polinômio de grau $n$, independente das características deste.

Nesta representação, tanto a adição quanto a multiplicação podem ser feitas em tempo $O(n)$. Na adição de $A(x)$ e $B(x)$ suponha, sem perda de generalidade, que o grau de $A(x)$ é igual ao grau de $B(x)$ e que:

$$A(x) = \{(x_0, y_0), (x_1, y_1), \ldots, (x_{m-1}, y_{m-1}), (x_m, y_m)\}$$
$$B(x) = \left\{(x_0, y'_0), (x_1, y'_1), \ldots, \left(x_{m-1}, y'_{m-1}\right), (x_m, y'_m)\right\}$$

então a representação de $S(x)$ é dada por:

$$S(x) = \left\{(x_0, y_0 + y'_0), (x_1, y_1 + y'_1), \ldots, \left(x_{m-1}, y_{m-1} + y'_{m-1}\right), (x_m, y_m + y'_m)\right\}$$

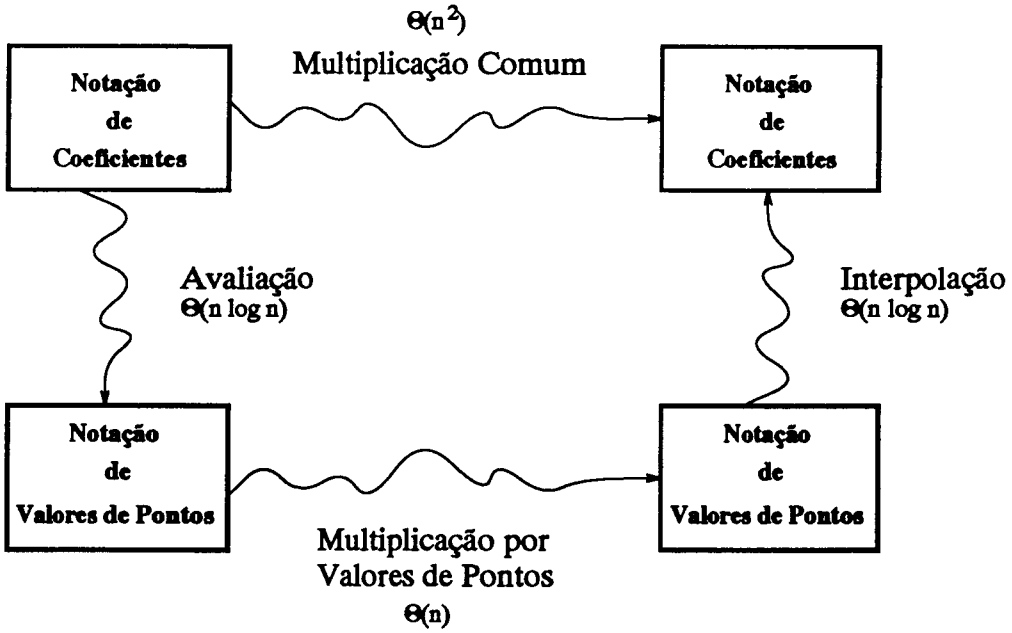


Figura 8.1: Processo para a multiplicação usando notação de coeficientes

Este método também é válido para a multiplicação com o adendo de que os polinômios a serem operados necessitam estar representados no número de pares de pontos necessários à representação do polinômio resultante. Desta forma suponha $n$, o grau do polinômio resultante igual a $2m$, logo teremos a seguinte representação para a multiplicação:

$$A(x) = \{(x_0, y_0), (x_1, y_1), \ldots, (x_{n-1}, y_{n-1}), (x_n, y_n)\}$$
$$B(x) = \left\{(x_0, y'_0), (x_1, y'_1), \ldots, \left(x_{n-1}, y'_{n-1}\right), (x_n, y'_n)\right\}$$

então a representação de $R(x)$ é dada por:

$$R(x) = \left\{(x_0, y_0 \times y'_0), (x_1, y_1 \times y'_1), \ldots, \left(x_{n-1}, y_{n-1} \times y'_{n-1}\right), (x_n, y_n \times y'_n)\right\}$$

Dado a facilidade de multiplicarmos um polinômio via notação de valores de pontos, podemos visualizar uma opção de fazermos multiplicação de polinômios representados por

coeficientes através de uma transformação de notações (vide figura 8.1).

Como vimos anteriormente, tanto a avaliação quanto a interpolação possuem tempos de execução $O(n^2)$. Note porém, que na avaliação utilizamos pontos quaisquer; desta forma, podemos imaginar que alguns pontos são triviais de serem avaliados.

Após verificação, vemos que alguns pontos são mais convenientes para a avaliação polinomial. Este conjunto de número são chamados de raízes de unidade[2]. Os números pertencentes a este conjunto, quando elevados a uma determinada potência, resultam um valor unitário.

Usaremos um conjunto especial de raízes de unidade que facilitarão nosso trabalho posteriormente. Para um dado $n$, tal conjunto é formado por um número complexo $c \in C$, tal que $c^n = 1$ e por suas potências $c^m$ para $0 < m < n$, logo nosso conjunto é $\{c^0, c^1, c^2, \ldots, c^{n-2}, c^{n-1}\}$, note que $c^n = c^0$. Chamamos este conjunto de enésimas raízes unitárias complexas[3].

Usaremos a notação $\omega_n^m$ para identificar um conjunto de enésimas raízes unitárias complexas para um valor $n$. Desta forma para $k = 8$ teríamos o conjunto denominado oitavas raízes unitárias complexas, dado por: $\{\omega_8^0, \omega_8^1, \omega_8^2, \omega_8^3, \omega_8^4, \omega_8^5, \omega_8^6, \omega_8^7\}$. Este conjunto está espalhado uniformemente no plano complexo (veja figura 8.2*(b)*).

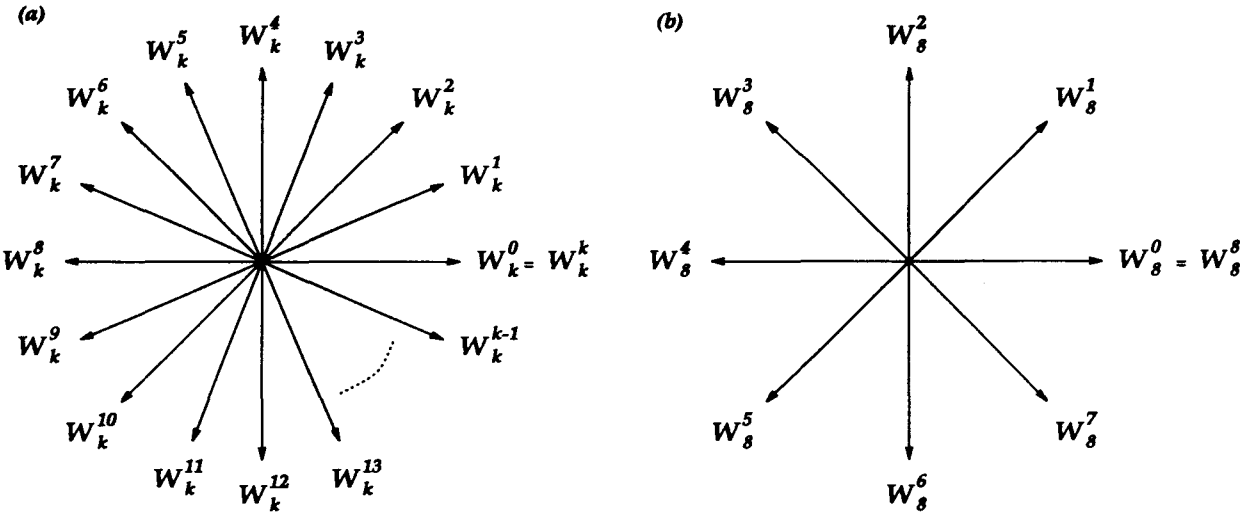


Figura 8.2: *(a)* Disposição das enésimas raízes unitárias complexas no plano complexo e *(b)* disposição das oitavas raízes unitárias complexas também no plano complexo

O valor de cada $\omega_n^m$ é dado por $e^{\frac{2\pi im}{n}}$ que pela fórmula de exponenciais de números complexos temos:

[2] *roots of unity*
[3] *complex nth root of unity*

$$\omega_n^m = \cos\left(\frac{2\pi m}{n}\right) + i \sin\left(\frac{2\pi m}{n}\right)$$

Como já citamos a TRF é uma forma de resolver a TDF[4]. Sendo assim, nesta aplicação, dizemos que a TDF é a avaliação de um polinômio nas raízes unitárias complexas. Seja o vetor de coeficientes $A = \{a_0, a_1, a_2, \ldots, a_{n-1}, a_n\}$, dizemos que o resultante $Y$ formado pelos $y_m$, é dado por:

$$y_m = A(\omega_n^m) = \sum_{i=0}^{n-1} a_i(\omega_n^m)^i,$$

dizemos então vetor $Y$ é a **Transformada Discreta de Fourier** do vetor $A$, ou $Y = TDF_n(A)$.

A TRF é usada nesta aplicação para melhorar o tempo necessário à avaliação de um polinômio nas raízes complexas. Com o uso desta técnica podemos chegar a um tempo $O(n\ log\ n)$. A TRF utiliza o método de divisão-e-conquista e se faz valer da seguinte propriedade das raízes complexas.

**Propriedade 8.1** *Se $j > 0$ é um número par, então o quadrado das $j$-ésimas raízes unitárias complexas resulta em $\frac{k}{2}$ $\frac{k}{2}$-ésimas raízes unitárias complexas.*

Para esclarecer esta propriedade, suponha um exemplo com as oitavas raízes unitárias complexas:

$$\Omega_8 : \omega_8^0, \omega_8^1, \omega_8^2, \omega_8^3, \omega_8^4, \omega_8^5, \omega_8^6, \omega_8^7$$

Pela figura 8.2*(b)*, podemos observar que $\omega_8^4 = -1$, logo podemos escrever:

$$\Omega_8 : \omega_8^0, \omega_8^1, \omega_8^2, \omega_8^3, -\omega_8^0, -\omega_8^1, -\omega_8^2, -\omega_8^3$$

desta forma, elevando cada termo ao quadrado, nos da 2 cópias de quartas raízes unitárias complexas:

$$\Omega_8^2 : \omega_4^0, \omega_4^1, \omega_4^2, \omega_4^3, \omega_4^0, \omega_4^1, \omega_4^2, \omega_4^3$$

Para usarmos tal propriedade temos que modificar a representação do polinômio de tal forma a termos somente potências pares:

$$P(x) = \sum_{i=0}^{k-1} a_i x^i \Rightarrow$$
$$P(x) = \sum_{i=0}^{\lfloor \frac{k-1}{2} \rfloor} a_{2i} x^{2i} + x \sum_{i=0}^{\lfloor \frac{k-1}{2} \rfloor} a_{2i+1} x^{2i}$$

[4]Transformada Discreta de Fourier. Em inglês DFT, *Discrete Fourier Tranform*.

Com isto, o problema se reduz a fazermos uma avaliação de dois polinômios de grau $\frac{n}{2}$ nos pontos $\omega_{\frac{n}{2}}^{m}$ e então combinar os resultados.

Com a propriedade 8.1 e a redução mostrada acima, temos tudo que necessitamos para aplicação da técnica de divisão-e-conquista usada na TRF. Resumidamente, para avaliarmos um polinômio de graus $n-1$ em $n$ pontos, dividimos o polinômio em dois outros de grau $\frac{k-1}{2}$ e os avaliamos em $\frac{k}{2}$ pontos (raízes unitárias complexas).

A partir desta descrição podemos dizer que o tempo de execução do algoritmo é proporcional a equação de recorrência da técnica de divisão-e-conquista, dada por:

$$T(n) = 2T\left(\frac{n}{2}\right) + O(n)$$

cuja solução é $O(n\ log\ n)$.

Um algoritmo paralelo pode se fazer valer da divisão-e-conquista na divisão de tarefas. Uma solução trivial no modelo PRAM executaria em tempo $O(log\ n)$, utilizando $n$ processadores. Conseqüentemente teríamos custo $C(n) = O(n\ log\ n)$.

Dada a figura 8.1, podemos verificar que para obtermos um algoritmo rápido para a multiplicação de polinômios basta que encontremos um algoritmo para interpolação pelo menos tão rápido quando o existente para avaliação que acabamos de descrever informalmente.

O uso da raízes unitárias complexas faz com que a operação de interpolação seja exatamente a inversa da avaliação[5]. Isto nos permite usar, na interpolação, o mesmo método usado na avaliação. Neste caso usamos os inversos das raízes unitárias complexas, isto é, suponha que temos os seguintes valores com resultado de avaliação polinomial:

$$\begin{aligned} P(\omega_n^0) &= s_0 \\ P(\omega_n^1) &= s_1 \\ P(\omega_n^2) &= s_2 \\ &\vdots \\ P(\omega_n^m) &= s_m \end{aligned}$$

desta forma podemos montar um polinômio da forma:

$$Q(x) = \sum_{i=0}^{m} s_i x^i$$

que quando avaliado nos devidos pontos (inverso das raízes unitárias complexas) produzirá como resultado os coeficientes do polinômio original. Desta forma, temos uma interpolação polinomial em tempo $O(n\ log\ n)$.

Como citamos, usamos na interpolação o inverso das raízes complexas:

[5]propriedade fundamental da inversão de uma Transformada de Fourier (veja [Yap94]).

$$\Omega_8^{-1}: \ \omega_8^0, \ \omega_8^{-1}, \ \omega_8^{-2}, \ \omega_8^{-3}, \ \omega_8^{-4}, \ \omega_8^{-5}, \ \omega_8^{-6}, \ \omega_8^{-7}$$
$$\Omega_8^{-1}: \ \omega_8^0, \ \omega_8^7, \ \omega_8^6, \ \omega_8^5, \ \omega_8^4, \ \omega_8^3, \ \omega_8^2, \ \omega_8^1$$

Note que os pontos são os mesmos usados na avaliação em uma ordem diferente.

Apresentaremos agora uma formalização, tal como apresentada em [Yap94], que nos mostra bem a ligação entre multiplicação de polinômios e TRF.

Seja $n$ um número par, e $A = [a_0, a_1, a_2, \ldots, a_{n-2}, a_{n-1}]$ e $B = [b_0, b_1, b_2, \ldots, b_{n-2}, b_{n-1}]$ dois vetores de tamanho $n$, a **convolução** de dois vetores é dada por:

$$\begin{aligned} C &= A \times B \\ &= [c_0, c_1, c_2, \ldots, c_{n-2}, c_{n-1}] \end{aligned}$$

onde $c_i = \sum_{j=0}^{i} a_j b_{i-j}$.

Suponha agora que os vetores $A$ e $B$ representam os coeficientes de dois polinômios, $P(x)$ e $Q(x)$ de grau menor que $\frac{n}{2}$, inserindo zeros para completar a representação com $n$ pontos . Então podemos afirmar que o vetor $C$ é o vetor do produto de polinômios $R(x) = P(x) \times Q(x)$. Note que este polinômio resultante possui grau menor que $n$.

Podemos então concluir que o convolução de dois vetores representa uma multiplicação de polinômios.

O teorema abaixo resume formalmente os processos de avaliação e interpolação de polinômios. Sua prova pode ser encontrada em [Yap94]. Para que este teorema se tornasse mais claro fizemos algumas modificações a partir de observações de Knuth [Knu81].

**Teorema 8.1** *Sejam $A$ e $B$ dois vetores de tamanho $n$ ($n$ par), onde em cada um deles existem $\frac{n}{2}$ posições com o valor zero. Logo:*

$$C = A \times B$$

$$TDF(A \times B) = TDF(A) \times TDF(B)$$

$$C = A \times B = TDF^{-1}(TDF(A) \times TDF(B))$$

Com esta formalização mostramos a ligação existente entre Multiplicação de Polinômios e TRF. Nas próximas seções mostraremos algoritmos para TRF em diversos modelos de computação paralela.

### 8.1.3 Desenvolvimento de um Algoritmo TRF

A seqüência de cálculos de uma TRF pode ser expressa por uma rede conhecida como *butterfly* (veja figura 8.3). Nesta figura representamos a execução de uma transformada rápida de Fourier de tamanho 8, pois temos como entrada um vetor de $X$ de tamanho 8 e como saída um vetor $Y$ (que é a transformada) também de tamanho 8. Cada vértice desta estrutura efetua operações que são partes de um cálculo total.

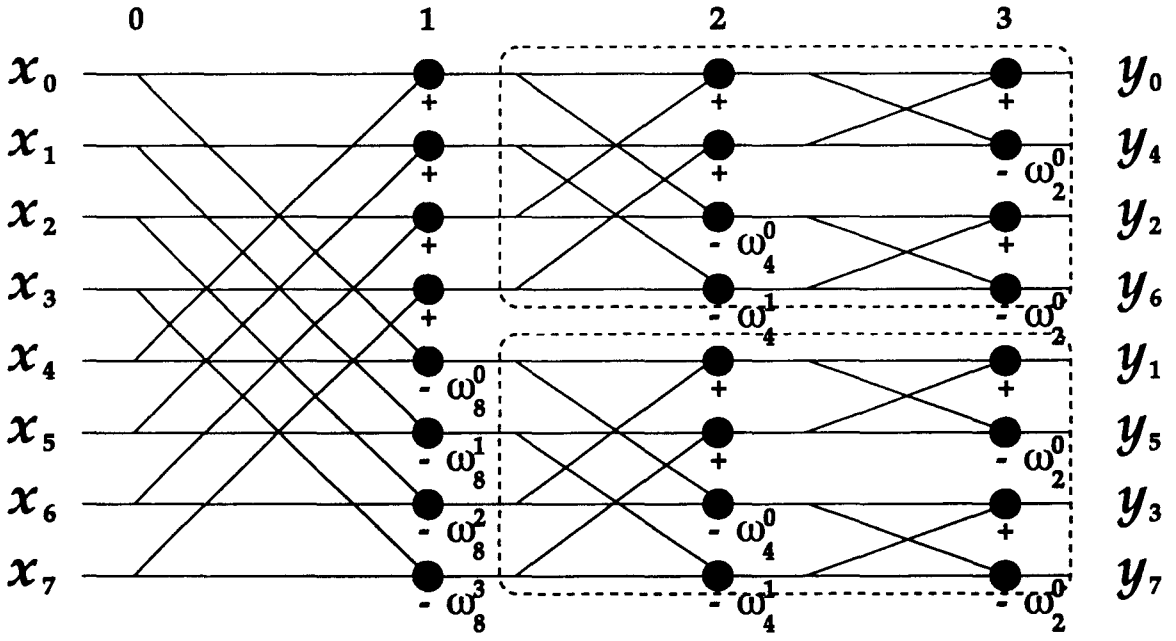


Figura 8.3: Estrutura da rede de interconexão *butterfly* mostrando sua característica de TRF

Uma rede deste tipo pode ser visualizada como um grafo direcionado com $n(log\ n+1)$ vértices distribuídos em $n$ linhas e $log\ n+1$ colunas. Cada vértice ($l$,$c$) está ligado a um outro vértice ($l$,$c+1$) e ($l'$,$c+1$) onde $l$ e $l'$ diferem apenas pelo $c+1$-ésimo bit mais significante.

Visualizando o grafo *butterfly* em termos de linhas e colunas e a representação de um vértice dada por ($Lin$,$Col$) temos a seguinte representação para os vértices $(0,2)$ e $(1,2)$:

$y_0 = (0,2) = (0,1) + (1,1)$
$y_4 = (1,2)\omega_2^0 = ((0,1) - (1,1))\omega_2^0$

Note também a natureza recursiva da estrutura do *butterfly*. Os dois círculos pontilhados na figura 8.3 representam dois pequenos *butterflies* que representam o cálculo de duas TRF de tamanho 4. Desta forma podemos concluir que uma TRF de tamanho $n$ é

formada por duas TRFs de tamanho $\frac{n}{2}$. Isto apenas vem confirmar a afirmação feita na seção anterior sobre a aplicabilidade da técnica de divisão-e-conquista.

O objetivo principal de um algoritmo paralelo para a TRF é alocar os vértices nos $p$ processadores existentes e fazer um escalonamento de comunicação de tal forma que minimizemos o tempo de saída da última resposta (lado direito do grafo representado na figura 8.3).

Faremos uma análise comparativa entre a resolução do modelo LogP e do modelo PRAM sob o ponto de vista de projeto de algoritmos. Estes dois modelo possuem algoritmos ótimos para resolução de Transformada de Fourier. Este tempo é dado por $\frac{n\ log\ n}{p}$ onde $n\ log\ n$ é o número de de vértices do *butterfly*, e $p$ é o número de processadores (poderosos).

### 8.1.4 Algoritmos Paralelos para TRF

Faremos a seguir uma análise comparativa de dois algoritmos para Transformada Rápida de Fourier desenvolvidos usando dois modelos bastante distintos: LogP e PRAM. Tentaremos, com este problema, mostrar as diferenças entre estes dois modelos com relação a nível de abstração.

Começaremos por mostrar um algoritmo EREW PRAM para TRF apresentado por Jájá [Jáj92]. Logo após, mostraremos um outro algoritmo para TRF desenvolvido para o modelo LogP.

Antes porém de mostrarmos o algoritmo PRAM para TRF faremos alguns comentários sobre tal algoritmo.

O que desejamos calcular é a TDF $y = W_n x$. Para tanto, devemos supor que $y$ e $x$ são vetores de $n$ posições e que $W_n$ é uma matriz construída a partir de $\omega = e^{i\frac{2\pi}{n}} = cos\frac{2\pi}{n} + i \times sen\frac{2\pi}{n}$, onde cada elemento $W(j,k) = \omega^{jk}$, e onde $j,k = (0,1,2,\ldots,n-1)$. Desta forma o valor de $W_4$, por exemplo, é representado por uma matriz $4 \times 4$ com os seguintes valores:

$$W_4 = \begin{Vmatrix} 1 & 1 & 1 & 1 \\ 1 & i & -1 & -i \\ 1 & -1 & 1 & -1 \\ 1 & -i & -1 & i \end{Vmatrix}$$

Seguindo com o exemplo podemos afirmar então que o valor do vetor $y$ é dado por:

$$\begin{aligned} y_0 &= x_0 + x_1 + x_2 + x_3 \\ y_1 &= x_0 + ix_1 - x_2 - ix_3 \\ y_2 &= x_0 - x_1 + x_2 - x_3 \\ y_3 &= x_0 - ix_1 - x_2 + ix_3 \end{aligned}$$

Note então que o cálculo da TDF pode ser representado como uma multiplicação de matrizes que é um algoritmo que no modelo PRAM possui custo de $O(n^2)$ operações com números complexos. O algoritmo que apresentaremos para resolução da TDF possui tempo $T(n) = O(log\ n)$ e custo $W(n) = O(n\ log\ n)$. Este custo coincide com o tempo do algoritmo seqüencial mais rápido e desta forma considerado ótimo [Jáj92].

Seguindo o que foi mostrado na seção 8.1.2, podemos afirmar que a técnica de divisão-e-conquista e bastante aplicável na formulação de um algoritmo PRAM, principalmente pelas características das raízes unitárias (vide figura 8.2).

Na figura 8.4 apresentamos um algoritmo EREW PRAM para TRF encontrado em [Jáj92]. Este algoritmo é mostrado recursivo para deixar claro que estamos usando a técnica de divisão-e-conquista.

**Entrada:** Um vetor $x$ de elementos complexos e $n$ suposto ser potência de 2, sem perda de generalidade.

**Saída:** Um vetor $y$ que é a TDF de $x$

TRF($x$,$n$)

```
(1)     if n = 2 then
(2)        y[1] = x[1] + x[2]
(3)        y[2] = x[1] - x[2]
(4)     else
(5)        for k:=0 to n/2 - 1 do in parallel
(6)           u[k] := x[k] + x[n/2 + k]
(7)           v[k] := ω^k (x[k] - x[n/2 + k])
(8)        z(1) := TRF(u, n/2)
(9)        z(2) := TRF(v, n/2)
(10)       for j:=0 to n - 1 do in parallel
(11)          if j mod 2 = 0 then
(12)             y[j] := z(1)[j/2]
(13)          else
(14)             y[j] := z(2)[(j-1)/2]
```

Figura 8.4: Algoritmo EREW PRAM para TRF

Note que este algoritmo PRAM expressa claramente a estrutura do problema apresentada informalmente na seção 8.1.2 como também expressa a estrutura *butterfly* apresentada na figura 8.3. Observe que as linhas (6) e (7) do algoritmo são generalizações do cálculo apresentado como exemplo na seção 8.1.3

Resumiremos a seguir um estudo realizado por Sahay [Sah93], sobre paralelização no modelo LogP de tarefas do tipo TRF, isto é, tarefas que podem ser representadas através de uma rede de dependência do tipo *butterfly*. Alguns problemas que se enquadram neste tipo de aplicação são: Avaliação de Polinômios e Ordenação Bitônica[6].

O objetivo principal de Sahay é tentar minimizar os custos de comunicação incorridos devido a limitações da largura de banda existentes em certas arquiteturas. O modelo LogP foi escolhido por Sahay pois este permite que explicitemos o escalonamento de comunicação, o que não é possível no modelo BSP, por exemplo.

A solução apresentada por Sahay [Sah93] se baseia exclusivamente na estrutura do problema de TRF que é representada por um *butterfly*. O algoritmo particiona os $n \log n$ vértices de um *butterfly* com $n$ entradas em $P$ processadores, organizando a seqüência de cálculo e comunicação de forma a conseguir um tempo de execução próximo de $\frac{n \log n}{P}$ (que é uma aceleração ideal). Esta aceleração só é alcançada quando o tamanho de $n$ é grande quando comparado com $P$. O autor cita, porém, que mesmo para valores de $n$ não muito grandes, a aceleração atingida chega a $\frac{P}{1+\frac{1}{\log n}}$.

Descreveremos informalmente o algoritmo apresentado por Sahay. Primeiramente descreveremos a alocação dos vértices para os processadores. Logo após, descreveremos as características do escalonamento da execução do algoritmo e conseqüentemente mostraremos seu tempo de execução em função dos parâmetros $L$, $g$ e $P$. O parâmetro $o$ foi desprezado nesta simplificação do modelo, pois o autor está supondo que a largura de banda das máquinas paralelas é bem pequena e desta forma $g$ é muito grande. Segundo Culler *et. al* [CKP+93] quando isto acontece o parâmetro $o$ pode ser desprezado para trabalharmos com um modelo mais simples.

O algoritmo apresentado por Sahay objetiva processar uma estrutura do tipo *butterfly* da forma mais rápida possível, minimizando o custo de operações de comunicação, que são compararativamente, as operações mais caras. Desta forma, um dos pontos mais importantes de um algoritmo LogP é a divisão dos dados entre os processadores.

Especificamente com relação a estrutura *butterfly*, existem duas maneiras principais de alocação dos elementos para os processadores:

- **Cíclica**: As linhas do *butterfly* são alocadas ciclicamente entre os processadores; desta forma caso tivéssemos 8 linhas e 4 processadores teríamos uma possível alocação dada por:
  - linha 0 e 4 - processador 0
  - linha 1 e 5 - processador 1
  - linha 2 e 6 - processador 2

[6] *Bitonic Sort*

- linha 3 e 7 - processador 3

- **Bloco**: Neste caso, um conjunto de linhas é alocado para cada processador, e desta forma teríamos uma alocação possível dada por:

  - linha 0 e 1 - processador 0
  - linha 2 e 3 - processador 1
  - linha 4 e 6 - processador 2
  - linha 6 e 7 - processador 3

Ao analisarmos a estrutura do *butterfly* para o cálculo da TRF vemos que as primeiras comunicações da estrutura seriam locais a cada processador se utilizássemos uma alocação cíclica. Em contra-partida as últimas comunicações somente seriam locais caso utilizássemos uma alocação em bloco[7]

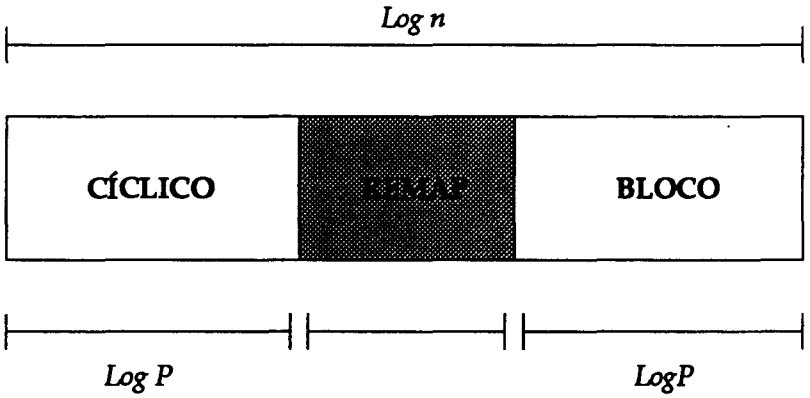


Figura 8.5: Estrutura abstrata da rede de interconexão *butterfly*, mostrando o mapeamento híbrido de nós aos processadores.

Sendo assim, Sahay [Sah93] nos propõe uma forma de alocação híbrida iniciando cíclico e terminando em bloco. Devemos também observar que esta divisão entre o que será cíclico e o que será bloco, não deve ser feita aleatoriamente já que desejamos minimizar as comunicações não locais entre processadores. Chamaremos de **Remapeamento** o processo de mudança de alocação cíclica para em bloco.

O problema agora é descobrir o local onde o remapeamento deve ocorrer. Este local depende diretamente da diferença entre o número de processadores, $P$, e o tamanho da entrada do *butterfly*, $n$.

[7]chamamos de primeiras comunicações as representadas nas primeiras colunas do grafo e conseqüentemente as últimas comunicações as representadas pelas últimas colunas do grafo.

Se garantirmos que $n \geq P^2$, podemos usar o mapeamento híbrido de tal forma que tenhamos somente uma comunicação não-local, ocorrendo justamente durante o remapeamento (vide figura 8.5).

Para vermos porque é necessários garantirmos que $n \geq P^2$ basta verificarmos que cada processador cuidará $\frac{n}{P}$ nós. A cada coluna $i$, o número de nós envolvidos em comunicação dobra. Assim, a comunicação deixa de ser local quando $2^i > \frac{n}{P}$ ou quando $i > log\ n - log\ P$. Isto mostra que precisamos de pelo menos $log\ P$ colunas para manter a comunicação local no caso cíclico. Raciocínio análogo mostra que também precisamos das últimas $logP$ colunas para manter a comunicação local no caso de alocação em bloco. Portanto é preciso que $log\ P + log\ P \leq log\ n$.

Como estamos utilizando um método híbrido, temos que garantir que todos os processadores cheguem a fase de remapeamento no mesmo instante. Como o modelo LogP é totalmente assíncrono temos que nos utilizar de uma barreira de sincronização antes do remapeamento.

O algoritmo funciona portanto em duas fases: uma antes do remapeamento (cíclica) e outra após (em bloco). Desta forma sejam $m = \frac{n}{P}$ e $\ell = \frac{m}{P}$. As duas fases do algoritmo são:

- **FASE I**: Para $c = 1 \ldots logm$, os vértices $(r, c)$ estão associados ao processador $r\,(modP)$;
- **FASE II**: para $c = logm \ldots logn$, os vértices $(r, c)$ estão associados ao processador $\left\lfloor \frac{r}{m} \right\rfloor$.

Esta alocação híbrida não influencia o tempo de computação do algoritmo, mas por outro lado, Culler *et. al* [CKP+93], nos afirmam que o tempo gasto para comunicação é diminuído por um fator de $log\ P$. Desta forma, o tempo total do algoritmo também é melhorado. Veja que caso a alocação não fosse híbrida (digamos totalmente cíclica), necessitaríamos de mais $log\ P$ comunicações totais para terminamos o algoritmo, e portanto, nosso tempo de comunicação seria multiplicado por $log\ P$.

Como mostramos, o tempo de comunicação total do algoritmo acima é igual ao tempo de uma comunicação total entre os processadores (difusão de todos para todos). Tal comunicação pode ser executada em tempo:

$$g(m - \ell - 1) + L$$

Esta equação pode ser facilmente entendida se pensarmos antes na equação de uma rede não híbrida. Supondo uma alocação totalmente cíclica, por exemplo, necessitamos enviar durante $log\ P$ fases (não locais) $\frac{n}{P}$ mensagens para todos os processadores. Neste caso temos um tempo de comunicação dado por:

$\left(g\frac{n}{P} + L\right) log\ P$

Já no modelo híbrido não temos necessidade de $log\ P$ comunicações, o que já nos dá um tempo:

$g\frac{n}{P} + L$

Como um bloco não precisa enviar mensagens para ele mesmo, podemos subtrair este tempo e chegar a equação final dada por:

$g\left(\frac{n}{P} - \frac{n}{P^2} - 1\right) + L = g(m - \ell - 1) + L$

Agora como o tempo de computação total do algoritmo é dado por:

$\frac{n}{P} \times log\ n = \frac{n}{P} \times (logm + log\ P) = m\ logm + m\ log\ P$

Podemos afirmar que o tempo total do algoritmo é dado por:

| | |
|---|---|
| $m\ log\ m + m\ log\ P + L + (m - \ell - 1)g$ | [1] |
| $\frac{n}{P} log\ \frac{n}{P} + \frac{n}{P}\ log\ P + L + (\frac{n}{P} - \frac{n}{P^2} - 1)g$ | [2] |
| $\frac{n}{P}\ log\ n - \frac{n}{P}\ log\ P + \frac{n}{P}\ log\ P + L + (\frac{n}{P} - \frac{n}{P^2} - 1)g$ | [3] |
| $\frac{n}{P}\ log\ n + L + (\frac{n}{P} - \frac{n}{P^2} - 1)g$ | $[FINAL]$ |

Com mais algumas considerações extra, esta previsão teórica fica bem próxima do desempenho na prática em uma CM-5 [CKP+93].

### 8.1.5 PRAM vs. LogP

Com as duas abordagens acima podemos ver mais claramente as diferenças existentes entre estes dois modelos. Faremos abaixo uma comparação entre tais modelos.

No algoritmo PRAM implementamos diretamente a técnica de divisão-e-conquista para TRF. Embora o algoritmo seja recursivo, é fácil de vermos que ele é equivalente ao comportamento da rede *butterfly* (vide figura 8.3).

Observe que embora o modelo PRAM resolva o problema baseado na rede *butterfly*, em nenhum ponto foram feitas considerações relacionadas ao número de processadores máximo ou mínimo, alocação dos nós nos processadores, ou seja, o modelo PRAM está apenas preocupado com a clareza do algoritmo e sua eficiência teórica, já que grande parte dos fatores práticos importantes não são considerados.

Ao nos referirmos a fatores práticos importantes, estamos nos referindo exatamente aos modelados pelos parâmetros do modelo LogP. Desta forma, podemos afirmar que,

independentemente do problema, este modelo tende a ser mais realista.

Ao verificarmos o papel do modelo LogP para o problema de TRF vemos que ele supõe encontrar a solução do problema através de uma rede *butterfly* abstrata (já que nenhuma máquina foi especificada), onde a maior preocupação está relacionada à alocação dos dados aos processadores. Ademais, o modelo permite que outros custos, como o de comunicação, sejam quantificados e conseqüentemente considerados na análise de desempenho deste.

Ao descrevermos o modelo LogP afirmamos que este modelo não fixa uma rede de interconexão, visto que estas tendem a possuir o mesmo desempenho. Neste exemplo podemos ver claramente que o modelo não fixou uma rede concreta e usou apenas uma rede abstrata.

Sendo assim, a conclusão principal que podemos tirar é que, neste caso, o modelo LogP funciona essencialmente como uma máquina abstrata que permite mapear mais facilmente um algoritmo baseado na rede *butterfly* a uma máquina paralela de memória distribuída.

Quando desejamos aplicar nosso algoritmo para TRF em uma máquina existente, poderemos ter que fazer algumas pequenas modificações para cada máquina alvo. Note a diferença de um algoritmo projetado neste nível de abstração e um possivelmente projetado para a máquina destino. Um algoritmo específico certamente conteria características particulares da máquina alvo, dificultando assim, sua transportabilidade para outras máquinas com arquiteturas diferentes. Neste ponto vale lembrar que transportabilidade é um das principais características de um bom modelo de computação paralela.

## 8.2 Eliminação de Gauss

### 8.2.1 Definição

Operações com matrizes aparecem constantemente como subpartes de grandes problemas que necessitam de resolução computacional.

Um dos problemas onde usamos matrizes, são os de sistemas de equações lineares. Um sistema de equações é um conjunto de equações com variáveis onde queremos, a partir destas equações, calcular o valor das variáveis, caso isto seja possível. Um sistema de equações é apresentado normalmente na seguinte forma:

$$\left.\begin{array}{ccccccccccccc}
a_{1,1}x_1 & + & a_{1,2}x_2 & + & a_{1,3}x_3 & + & \ldots & + & a_{1,n-1}x_{n-1} & + & a_{1,n}x_n & = & b_1 \\
a_{2,1}x_1 & + & a_{2,2}x_2 & + & a_{2,3}x_3 & + & \ldots & + & a_{2,n-1}x_{n-1} & + & a_{2,n}x_n & = & b_2 \\
a_{3,1}x_1 & + & a_{3,2}x_2 & + & a_{3,3}x_3 & + & \ldots & + & a_{3,n-1}x_{n-1} & + & a_{3,n}x_n & = & b_3 \\
\vdots & & \vdots & & \vdots & & \vdots & & \vdots & & \vdots & = & \vdots \\
a_{n,1}x_1 & + & a_{n,2}x_2 & + & a_{n,3}x_3 & + & \ldots & + & a_{n,n-1}x_{n-1} & + & a_{n,n}x_n & = & b_n
\end{array}\right\}$$

O sistema de equações lineares acima é equivalente a seguinte operação com as matrizes

$A$, $X$ e $B$:

$$AX = B$$

$$\underbrace{\begin{Vmatrix} a_{1,1} & a_{1,2} & a_{1,3} & a_{1,4} & a_{1,5} & \cdots & a_{1,n} \\ a_{2,1} & a_{2,2} & a_{2,3} & a_{2,4} & a_{2,5} & \cdots & a_{2,n} \\ a_{3,1} & a_{3,2} & a_{3,3} & a_{3,4} & a_{3,5} & \cdots & a_{3,n} \\ a_{4,1} & a_{4,2} & a_{4,3} & a_{4,4} & a_{4,5} & \cdots & a_{4,n} \\ a_{5,1} & a_{5,2} & a_{5,3} & a_{5,4} & a_{5,5} & \cdots & a_{5,n} \\ \vdots & \vdots & \vdots & \vdots & \vdots & \ddots & \vdots \\ a_{n-1,1} & a_{n-1,2} & a_{n-1,3} & a_{n-1,4} & a_{n-1,5} & \cdots & a_{n-1,n} \\ a_{n,1} & a_{n,2} & a_{n,3} & a_{n,4} & a_{n,5} & \cdots & a_{n,n} \end{Vmatrix}}_{A} \underbrace{\begin{Vmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \\ \vdots \\ x_{n-1} \\ x_n \end{Vmatrix}}_{X} = \underbrace{\begin{Vmatrix} b_1 \\ b_2 \\ b_3 \\ b_4 \\ b_5 \\ \vdots \\ b_{n-1} \\ b_n \end{Vmatrix}}_{B}$$

Podemos fazer certas operações em sistemas de equações lineares sem que os resultados sejam alterados. Desta forma podemos tentar, através destas operações, obter sistemas de equações mais simples. As operações que utilizaremos são:

- **Multiplicação por uma Constante:** Podemos multiplicar qualquer equação de um sistema de equações lineares por uma constante.
- **Soma de Equações:** Podemos somar indiscriminadamente quaisquer duas equações e substituir uma delas pela soma.
- **Permuta de Equações:** Podemos permutar a posição de duas equações do sistema.
- **Permuta de Variáveis:** Podemos também permutar a posição de duas váriáveis dentro do sistema. Porém, temos que executar isto para todas as equações.

Suponha, sem perda de generalidade, que temos o mesmo número de equações e variáveis. Desta forma, o método de Eliminação de Gauss propõe que eliminemos cada variável $x_i$ , das equações $i + 1$ até $n$. Mostraremos um exemplo onde podemos observar mais claramente tal método.

Suponha o seguinte sistema de equações lineares na forma de matrizes:

$$\left.\begin{Vmatrix} 2 & 1 & -3 \\ 1 & -1 & -1 \\ 5 & -2 & 2 \end{Vmatrix} \begin{Vmatrix} x_1 \\ x_2 \\ x_3 \end{Vmatrix} = \begin{Vmatrix} -11 \\ 1 \\ 24 \end{Vmatrix}\right\}$$

Suponha agora as seguintes operações nas matrizes sabendo que $[i]$ representa a $i$-ésima linha destas:

- $[2] = [2] + [1] \times -\frac{1}{2}$
- $[3] = [3] + [1] \times -\frac{5}{2}$

Obtemos então:

$$\left.\begin{Vmatrix} 2 & 1 & -3 \\ 0 & \frac{-3}{2} & \frac{1}{2} \\ 0 & \frac{-9}{2} & \frac{19}{2} \end{Vmatrix} \begin{Vmatrix} x_1 \\ x_2 \\ x_3 \end{Vmatrix} = \begin{Vmatrix} -11 \\ \frac{13}{2} \\ \frac{103}{2} \end{Vmatrix} \right\}$$

Note que a primeira coluna da matriz $A$ foi totalmente zerada com exceção da primeira linha. Fazendo agora a seguinte operação chegaremos a matriz desejada:

- $[3] = [3] + [2] \times -3$

$$\left.\begin{Vmatrix} 2 & 1 & -3 \\ 0 & \frac{-3}{2} & \frac{1}{2} \\ 0 & 0 & 8 \end{Vmatrix} \begin{Vmatrix} x_1 \\ x_2 \\ x_3 \end{Vmatrix} = \begin{Vmatrix} -11 \\ \frac{13}{2} \\ 32 \end{Vmatrix} \right\}$$

Temos agora, como desejado, a matriz $A$ totalmente simplificada. Chegamos então ao seguinte sistema de equações lineares:

$$\left.\begin{array}{rcrcrcl} 2x_1 & + & x_2 & - & 3x_3 & = & -11 \\ & & \frac{-3}{2}x_2 & + & \frac{1}{2}x_3 & = & \frac{13}{2} \\ & & & & 8x_3 & = & 32 \end{array}\right\}$$

que possui solução igual a:

$$\left|\begin{array}{rcl} 8x_3 = 32 & \Rightarrow & \\ x_3 = 4 & & \end{array}\right| \left|\begin{array}{rcrcrcl} \frac{-3}{2}x_2 & + & \frac{1}{2}x_3 & = & \frac{13}{2} & \Rightarrow & \\ \frac{-3}{2}x_2 & + & 2 & = & \frac{13}{2} & \Rightarrow & \\ \frac{-3}{2}x_2 & & & = & \frac{9}{2} & \Rightarrow & \\ x_2 & & & = & -3 & & \end{array}\right| \left|\begin{array}{rcrcrcl} 2x_1 & + & x_2 & - & 3x_3 & = & -11 \\ 2x_1 & - & 3 & - & 12 & = & -11 \\ 2x_1 & & & & & = & 4 \\ x_1 & & & & & = & 2 \end{array}\right|$$

Podemos representar este processo de eliminações de acordo como os sistemas de equações lineares abaixo, onde o coeficiente $a_{i,j}^{(k)}$ representa o termo $a_{i,j}$ resultante após a $k$-ésima iteração do algoritmo. A diferença básica desta representação para a apresentada acima é o fato de que nesta estamos pensando em termos cada coeficiente e não de linhas:

$$\left.\begin{array}{rcrcrcrcrcl}
a_{1,1}x_1 & + & a_{1,2}x_2 & + & a_{1,3}x_3 & + & \ldots & + & a_{1,n-1}x_{n-1} & + & a_{1,n}x_n & = & b_1 \\
 & & a^{\langle 1\rangle}_{2,2}x_2 & + & a^{\langle 1\rangle}_{2,3}x_3 & + & \ldots & + & a^{\langle 1\rangle}_{2,n-1}x_{n-1} & + & a^{\langle 1\rangle}_{2,n}x_n & = & b^{\langle 1\rangle}_2 \\
 & & a^{\langle 1\rangle}_{2,3}x_2 & + & a^{\langle 1\rangle}_{3,3}x_3 & + & \ldots & + & a^{\langle 1\rangle}_{3,n-1}x_{n-1} & + & a^{\langle 1\rangle}_{3,n}x_n & = & b^{\langle 1\rangle}_3 \\
 & & \vdots & & \vdots & & \vdots & & \vdots & & \vdots & = & \vdots \\
 & & a^{\langle 1\rangle}_{n,2}x_2 & + & a^{\langle 1\rangle}_{n,3}x_3 & + & \ldots & + & a^{\langle 1\rangle}_{n,n-1}x_{n-1} & + & a^{\langle 1\rangle}_{n,n}x_n & = & b^{\langle 1\rangle}_n
\end{array}\right\}$$

$$\left.\begin{array}{rcrcrcrcrcl}
a_{1,1}x_1 & + & a_{1,2}x_2 & + & a_{1,3}x_3 & + & \ldots & + & a_{1,n-1}x_{n-1} & + & a_{1,n}x_n & = & b_1 \\
 & & a^{\langle 1\rangle}_{2,2}x_2 & + & a^{\langle 1\rangle}_{2,3}x_3 & + & \ldots & + & a^{\langle 1\rangle}_{2,n-1}x_{n-1} & + & a^{\langle 1\rangle}_{2,n}x_n & = & b^{\langle 1\rangle}_2 \\
 & & & & a^{\langle 2\rangle}_{3,3}x_3 & + & \ldots & + & a^{\langle 2\rangle}_{3,n-1}x_{n-1} & + & a^{\langle 2\rangle}_{3,n}x_n & = & b^{\langle 2\rangle}_3 \\
 & & & & \vdots & & \vdots & & \vdots & & \vdots & = & \vdots \\
 & & & & a^{\langle 2\rangle}_{n,3}x_3 & + & \ldots & + & a^{\langle 2\rangle}_{n,n-1}x_{n-1} & + & a^{\langle 2\rangle}_{n,n}x_n & = & b^{\langle 2\rangle}_n
\end{array}\right\}$$

$$\left.\begin{array}{rcrcrcrcrcl}
a_{1,1}x_1 & + & a_{1,2}x_2 & + & a_{1,3}x_3 & + & \ldots & + & a_{1,n-1}x_{n-1} & + & a_{1,n}x_n & = & b_1 \\
 & & a^{\langle 1\rangle}_{2,2}x_2 & + & a^{\langle 1\rangle}_{2,3}x_3 & + & \ldots & + & a^{\langle 1\rangle}_{2,n-1}x_{n-1} & + & a^{\langle 1\rangle}_{2,n}x_n & = & b^{\langle 1\rangle}_2 \\
 & & & & a^{\langle 2\rangle}_{3,3}x_3 & + & \ldots & + & a^{\langle 2\rangle}_{3,n-1}x_{n-1} & + & a^{\langle 2\rangle}_{3,n}x_n & = & b^{\langle 2\rangle}_3 \\
 & & & & & & \vdots & & \vdots & & \vdots & = & \vdots \\
 & & & & & & \ldots & + & a^{\langle n\rangle}_{n,n-1}x_{n-1} & + & a^{\langle n\rangle}_{n,n}x_n & = & b^{\langle n\rangle}_n
\end{array}\right\}$$

Podemos generalizar as operações executadas em cada elemento do sistema acima diretamente para equações abaixo:

$$a^{\langle k\rangle}_{i,j} = a^{\langle k-1\rangle}_{i,j} - \frac{a^{\langle k-1\rangle}_{i,k}}{a^{\langle k-1\rangle}_{k,k}} a^{\langle k-1\rangle}_{k,j} \qquad\qquad b^{\langle k\rangle}_{i} = b^{\langle k-1\rangle}_{i} - \frac{a^{\langle k-1\rangle}_{i,k}}{a^{\langle k-1\rangle}_{k,k}} b^{\langle k-1\rangle}_{k}$$

Para que estas equações sejam calculáveis é preciso que em nenhum momento haja divisão por zero. Pode-se provar que se a matriz for inversível é sempre possível encontrar uma permutação das colunas tal que não ocorram tais divisões.

### 8.2.2 Algoritmos Paralelos para Eliminação de Gauss

Nesta seção faremos com a Eliminação de Gauss algo semelhante ao feito com TRF; usaremos o problema para estudar as diferenças existentes entre dois modelos. Primeiramente apresentaremos algoritmos nos modelos PRAM e BSP. Na próxima seção faremos uma comparação entre os modelo BSP e LogP. Mostraremos a solução PRAM para este problema pois ela pode ajudar em um melhor entendimento do algoritmo no outro modelo.

Como afirmamos na seção 8.2.1, a Eliminação de Gauss consiste em eliminar cada variável $x_i$ das equações $i+1$ até $n$. Estas operações podem ser representadas em um algoritmo PRAM diretamente. Mostramos na figura 8.6 um algoritmo CRCW PRAM para tal problema adaptado de [Akl89].

**Entrada:** Uma matriz $A[n, n+1]$ onde a $n+1$-ésima coluna representa o vetor $B$.

**Saída:** A matriz $A$ com todos os coeficiente das variáveis $x_i$, das equações $i+1$ até $n$, zerados.

`Gauss(`$A$`)`

```
(1)          for k := 1 to n do
(2)             for i := 1 to n do in parallel
(3)                for j := k to n+1 do in parallel
(4)                   if i ≠ k then
(5)                      a_{i,j} := a_{i,j} - (a_{i,k}/a_{k,k}) × a_{k,j}
```

Figura 8.6: Algoritmo CRCW PRAM para o problema de Eliminação de Gauss.

Podemos ver claramente que realmente o algoritmo apresentado possui leitura concorrente no passo (5). Tal algoritmo possui tempo de execução, $T(n) = O(n)$ devido ao laço (1) que não é executado em paralelo. Seguindo o mesmo pensamento, vemos que o algoritmo necessita de $O(n^2)$ processadores para se executado devido aos laço encadeados (4) e (5). Desta forma o algoritmo acima possui custo, $C(n) = O(n^3)$.

Como afirmamos, o algoritmo apresentado acima para o modelo PRAM possui custo $O(n^3)$. Se detalhássemos este custo veríamos que o algoritmo executa $n^3 + O(n^2)$ operações devido a matriz $B$ que é adicionada como última coluna de $A$.

Nos algoritmos BSP, geralmente, temos que fazer suposições a respeito dos parâmetros. Na seção 6.1.3 mostramos a importância da folga paralela nos algoritmos BSP. No algoritmo que apresentaremos a seguir é necessário que asseguremos que o tamanho da entrada, $n$, esteja relacionado ao número de processadores de tal forma que $p \leq n^2$ (na pior das hipóteses teremos um processador para cada elemento).

Apresentaremos o algoritmo para Eliminação de Gauss de forma semelhante a apresentada por Gerbessiotis e Valiant [GV93], mas de uma maneira mais informal.

O algoritmo está dividido em três procedimentos que são executados $n$ vezes (vide figura 8.7).

Mostraremos mais detalhadamente cada um dos procedimentos do algoritmo (figura 8.7) após fazermos uma descrição sumária do algoritmo.

O objetivo de Gerbessiotis e Valiant ao propor este algoritmo não é o de alcançar o melhor desempenho possível e sim mostrar que o projeto de um algoritmo utilizando os parâmetros $p$, $g$ e $L$ não é uma tarefa difícil.

O algoritmo consiste, em linhas gerais, dividir o número de processadores $p$ em $\sqrt{p}$ gru-

```
(1)          for k := 1 to n do
(2)             Pivô(k)
(3)             Comunica(k)
(4)             Calcula(k)
```

Figura 8.7: Macro código de um algoritmo para Eliminação de Gauss no modelo BSP.

pos de tamanho $\sqrt{p}$, e onde cada processador $p_i$ é responsável por, no máximo, $\frac{n}{\sqrt{p}}\left(\frac{n}{\sqrt{p}}+1\right)$ elementos. Note que nesta soma, o tempo 1 somado, representa a matriz $B$ que foi adicionada como $n+1$-ésima coluna da matriz $A$. Feita esta divisão o algoritmo está apto a executar os três procedimentos básicos.

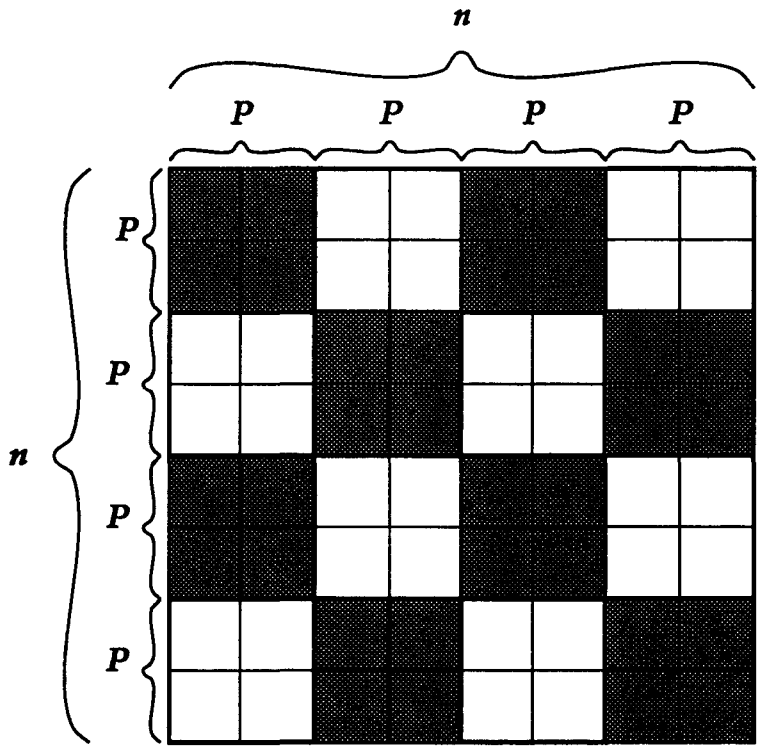


Figura 8.8: Divisão de elementos de uma matriz de $n \times n$ elementos ($n = 8$) entre $P$ processadores ($P = 16$)

Antes porém de mostrarmos os procedimentos, necessitamos introduzir algumas notações importantes, tais notações podem ficar mais claras pela figura 8.8.

- $Coluna^{P(k)}$: Conjunto dos $\sqrt{p}$ processadores que contêm elementos da coluna $k$.

- $Linha^{P(k)}$: Conjunto dos $\sqrt{p}$ processadores que contêm elementos da linha $k$.
- $Proc(a_{k,j})$: Processador que contém o elemento $a_{k,j}$ da matriz $A$.

Com estas notações podemos agora descrever os procedimentos do algoritmo de Eliminação de Gauss para BSP mais detalhadamente:

- **Pivô($k$)**: Este procedimento é composto de 3 passos que visam, no todo, achar o elemento pivô que será usado no cálculo de eliminação de Gauss. Este procedimento se faz necessário pois o elemento pivô deve ser, ao menos, não nulo, para que as equações fundamentais da eliminação sejam calculáveis. Os passos deste procedimento são:
  - **Passo 1**: Cada processador pertencente ao conjunto $Coluna^{P(k)}$ procura, dentre os elementos locais a ele, aquele que possui maior valor absoluto e que está simultaneamente na coluna $k$ e linha $j$, para $j \geq k$. Note que estamos tomando somente os $j$s com índice maior que $k$, pois a eliminação de Gauss transforma a matriz A em uma matriz triangular superior. Após se achar este máximo ele é comunicado ao processador $Proc(a_{k,k})$, que é o processador do elemento da diagonal.
    **Análise**: Este passo é composto de duas partes: uma computação que gasta tempo $\frac{n}{\sqrt{P}}$, pois cada processador está procurando o máximo de seus elementos locais; e uma comunicação de $\sqrt{P}-1$ mensagens. Estas mensagens correspondem a uma $\sqrt{P}-1$-relação que gasta tempo $g(\sqrt{P}-1)$.
  - **Passo 2**: O processador $Proc(a_{k,k})$ procura o maior elemento dentre aqueles recebidos no passo anterior. Tendo este elemento, identifica sua linha, $l$, e comunica este valor para o conjunto de processadores $Linha^{P(k)}$ e $Linha^{P(l)}$.
    **Análise**: Neste passo temos que o tempo para computação é $\sqrt{P}$, já que o processador está procurando o máximo de $\sqrt{P}$ elementos. Com relação a comunicação, são enviados $2\sqrt{P}$ mensagens que gasta tempo $g(2\sqrt{P})$.
  - **Passo 3**: Os processadores pertencentes ao conjunto $Linha^{P(k)}$ e $Linha^{P(l)}$ trocam seus elementos para toda coluna $k$.
    **Análise**: Aqui só temos comunicação. O que estamos fazendo é permutando todos os elementos de duas linhas, logo necessitamos de $\frac{n}{\sqrt{P}}$ mensagens gastando tempo $g(\frac{n}{\sqrt{P}})$
- **Comunica($k$)**: Composto de $\lceil log\sqrt{P} \rceil$ passos que visam comunicar todos os elementos necessários ao cálculo da eliminação para cada elemento da matriz dado pela equação apresentada no final da seção 8.2.1.

- **Passo 1**: O processador $Proc(a_{k,k})$ executa uma difusão de seu dado para os processadores do conjunto $Linha^{P(k)}$.
  **Análise**: Temos aqui o tempo de comunicação dado por $g\sqrt{P}$ equivalente ao processamento de uma $\sqrt{P}$-relação.
- **Passo 2**: Cada processador do conjunto $Linha^{P(k)}$ divide o elemento que possui pelo elemento $a_{k,k}$ recebido no passo anterior. Este cálculo consiste na execução da divisão $\frac{a_{k,j}^{<k-1>}}{a_{k,k}^{<k-1>}}$ pertencente a equação principal de eliminação. Note que a divisão $\frac{b_k^{<k-1>}}{a_{k,k}^{<k-1>}}$ é executada também já que $B$ foi acrescentada na matriz $A$ como ultima coluna.
  **Análise**: Veja que cada processador possui $\frac{n}{\sqrt{P}}$ elementos e desta forma possui tempo de execução $\frac{n}{\sqrt{P}}$ referente as divisões efetuadas.
- **Passo 3, Passo 4, Passo 5, ..., Passo** $\lceil log\sqrt{P} \rceil + 2$: Nestes passos cada processador $j$ do conjunto $Linha^{P(k)}$ executa uma difusão do elemento computado no passo 2 para todos processadores da $Coluna^{P(j)}$. Simultaneamente todo $j$-ésimo processador do conjunto $Coluna^{P(k)}$ executa uma difusão do elemento da coluna $k$ para todos os $j$-ésimos processadores de todos os outros grupos. Observe que o objetivo principal destes passos é disponibilizar todos os dados necessários ao cálculo da equação de eliminação de Gauss.
  **Análise**: Em cada um dos passos acima gastamos um tempo de comunicação equivalente a uma $\frac{2n}{\sqrt{P}}$-relação, já que são duas transmissões simultâneas. Isto nos leva a um tempo gasto de $\left(\frac{2n}{\sqrt{P}}\right) g$. Apesar das transmissões ocorrerem para no máximo $\sqrt{P}$ processadores, utilizamos somente $log\ \sqrt{P}$ passos, pois podemos utilizar uma difusão com replicação binária.

- **Calcula($k$)**: Possui somente um passo que é o cálculo da eliminação propriamente dita, já que todos os dados necessários estão disponíveis.

  - **Passo 1**: Todos os processadores executam a equação para cálculo da eliminação. Note que um parte do cálculo já foi executada no passo 2 do procedimento Comunica($k$).
    **Análise**: Neste passo necessitamos calcular a equação para todos os elementos do proecssador. Desta forma temos que, devido a fórmula da eliminação, gastamos tempo:

    $$T_k = \frac{2n}{\sqrt{P}} \left[\frac{n-k}{\sqrt{P}}\right] + \frac{2n}{\sqrt{P}}$$

    O primeiro termo se deve às 2 operações necessárias para todas as linhas a

partir da coluna $k$. O segundo termo se refere às operações sobre a coluna que representa o vetor $B$.

Tendo em vista o algoritmo apresentado acima se suas respectivas análises a cada passo, podemos resumir os tempos de cada procedimento:

- Pivô
  - Computação: $\frac{n}{\sqrt{P}} + \sqrt{P}$
  - Comunicação: $g\left(\sqrt{P} - 1 + 2\sqrt{P} + \frac{n}{\sqrt{P}}\right)$
- Comunica
  - Computação: $\frac{n}{\sqrt{P}}$
  - Comunicação: $g\left(\sqrt{P} + log\sqrt{P}\left(\frac{2n}{\sqrt{P}}\right)\right)$
- Calcula
  - Computação: $T_k$

Antes de mostrarmos a análise total do algoritmo resaltamos que em cada superpasso gastamos em computação no mínimo o tamanho do intervalo entre barreiras de sincronizações, $L$. Da mesma forma, em comunicação gastamos no mínimo o número de mensagens que podemos rotear entre as barreiras de sincronização, $\frac{L}{g}$.

Sendo assim podemos dizer que nos $n$ passos do algoritmo teríamos uma expressão de computação dada por:

$$2n\ max\left\{L, \frac{n}{\sqrt{P}}\right\} + n\ max\left\{L, \sqrt{P}\right\} + \sum_{k=1}^{n} max\left\{L, T_k\right\}$$

Da mesma forma o tempo gasto com comunicações é:

$$ng\ \left(4\ max\left\{\frac{L}{g}, \sqrt{P}\right\} + max\left\{\frac{L}{g}, \frac{n}{\sqrt{P}}\right\} + log\sqrt{P}\ max\left\{\frac{L}{g}, \frac{2n}{\sqrt{P}}\right\}\right)$$

Gerbessiotis e Valiant observam que a comunicação de $\sqrt{P}$ valores, que aparece na expressão acima, pode ser otimizada através de um processo de difusão.

Após termos feito toda esta análise do algoritmo BSP, o passo seguinte seria aplicar as equações para valores específicos de $P$, $L$, $g$ e $n$. Em [GV93] nos é apresentado um exemplo para $P = 100$, $n = 1000$, $L = 100$ e $g = 15$. Neste caso os autores afirmam que a parte computacional fica aproximadamente 7% mais lenta que o ótimo e que o tempo de comunicação é sempre menor que o de computação.

### 8.2.3 PRAM vs. BSP

Se olharmos atentamente para os algoritmos apresentados, veremos facilmente que os dois refletem claramente as características do problema. O algoritmo BSP consiste basicamente do algoritmo PRAM com considerações a respeito de memória distribuída. Estas considerações estão mostradas quando incluimos os parâmetros do modelo BSP na análise de seu algoritmo.

Note porém, que só a consideração destes fatores adicionais, apesar de tornar o algoritmo mais realista, o tornou mais complexo. Desta forma devemos fazer um bom estudo dos nossos objetivos para avaliarmos o que precisamos mais: realidade ou simplicidade.

# Capítulo 9

# Conclusão

## 9.1 Conclusão Geral

Apresentamos neste trabalho alguns modelos de computação com características bem distintas. Descrevemos o modelo PRAM e suas extensões e dois outros modelos que têm uma visão diferenciada do problema: BSP e LogP.

Apresentaremos abaixo um resumo das conclusões tiradas sobre cada um destes modelos.

### 9.1.1 PRAM e Suas Extensões.

Como foi afirmado diversas vezes neste trabalho, o modelo PRAM se apresenta como um ótimo modelo para projeto e análise de algoritmos, principalmente em uma primeira fase.

Suas extensões têm como objetivo geral torná-lo mais realista e, se possível, deixando-o tão simples quanto antes. O modelo QRQW PRAM [GMR94c], por exemplo, consegue modelar de uma forma mais realista as arquiteturas paralelas existentes sem se tornar por demais complexo.

Utilizando o conceito de Vishkin, que considera o PRAM uma família de modelos que possua o PRAM como base, faremos considerações sobre o modelo QRQW PRAM, visando mostrar a viabilidade de modelos baseados em PRAM. Optamos em fazer este detalhamento com o QRQW PRAM por este ter sido proposto mais recentemente e também por ser a proposta que consideramos mais simples e realista, dentre as extensões do PRAM.

A figura 9.1 nos mostra como se situa o QRQW PRAM dentro de submodelos do PRAM. Esta figura também nos mostra como é equilibrado este modelo com relação a simplicidade e realidade, pelo menos comparado com submodelos do PRAM.

Outra boa característica deste modelo é sua possibilidade de emulação pelo modelo BSP, onde as características das máquinas estão mais realisticamente modeladas[1]. Esta

[1] Esta emulação também é possível com o modelo PRAM original

característica reforça o argumento de que os modelos de fila são boas escolhas para projeto e análise de algoritmos. Resultados mais objetivos com relação a esta simulação serão mostrados na seção 9.1.2.

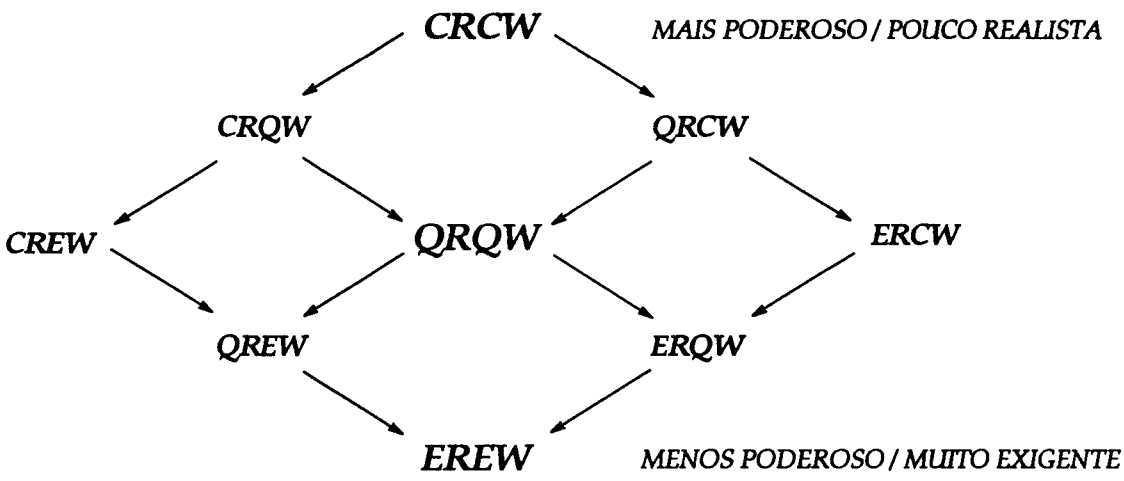


Figura 9.1: Estrutura mostrando a relação entre modelos PRAM mais fracos e mais fortes. Adaptado de [GMR94d]

Apesar destas características, há alguns problemas relacionados com a contabilização da contenção por memória e com a não contabilização de outros fatores.

Por ser um modelo considerado de alto nível, o QRQW PRAM não prevê a utilização de mecanismos especiais para combinação de mensagens existentes em algumas máquinas paralelas. Mecanismos deste tipo possibilitam diminuições substanciais no grau máximo da contenção por memória. No modelo QRQW PRAM a contabilização de custos é, em geral, realizada pelo pior caso independentemente da existência ou não de tais mecanismos nas máquinas onde o algoritmo será implementado. Desta forma a contabilização da contenção por memória de $k$ processadores será $O(k)$, note então que se $k$ for muito grande o algoritmo pode ser considerado inviável.

Já a contabilização de fatores como latência, custo de sincronização, entre outros, não é realizada no modelo QRQW PRAM por questões de simplicidade e também pelo fato de seus proponentes acharem que o problema de contenção por memória é um dos mais sérios em computação paralela. O fato é que assim como no modelo PRAM, algumas previsões de desempenho podem não refletir a realidade.

### 9.1.2 BSP

O modelo BSP se mostra como um excelente modelo para projeto e análise de algoritmos paralelos de uma forma independente de arquitetura. Note que a diferença deste para os modelos da família PRAM é que defendemos o uso do PRAM e suas extensões apenas em um primeiro passo do projeto de algoritmos. O BSP é um modelo que devido a existência de sincronismo não dificulta demasiadamente este trabalho de projeto, podendo ser usado desde o início do projeto.

Uma das melhores características do BSP é o seu papel de modelo intermediário, não sendo muito de alto nível, para não abstrair fatores essenciais ao desenvolvimento de algoritmos paralelos; nem de baixo nível, se detendo a detalhes algumas vezes desnecessários e difíceis de serem tratados por um projetista de algoritmos.

A proposta do modelo BSP apresentada por Valiant [Val90] é bastante fundamentada e suas idéias não surgiram repentinamente. Várias propostas anteriores de extensões do PRAM tornam o modelo BSP consistente, já que algumas idéias implementadas neste modelo podem ser observadas nessas extensões. Em 1989, Gibbons [Gib89] propôs o modelo *Phase* PRAM (descrito sucintamente na seção 5.3) que usa o conceito de sincronização em fases, semelhante ao conceito de barreira de sincronização do modelo BSP. Isto nos mostra que, de certa forma, o modelo BSP pode ser visualizado como uma extensão do modelo PRAM onde o intervalo de sincronização não é rígido a um passo. Para comprovação desta afirmação basta usarmos os parâmetros do modelo BSP com valores adequados.

Por poder ser visto como uma extensão do modelo PRAM, a proposta do BSP pode ser mais facilmente entendida pelos usuários. A existência de um tipo de sincronização facilita o trabalho do projetista de algoritmos, pois podemos ter um controle mais efetivo da execução do algoritmo. Além de ser relativamente simples, o modelo BSP aproveita todo trabalho desenvolvido para o modelo PRAM, pois este pode ser simulado no modelo BSP. Dada suficiente folga paralela, um algoritmo CRCW PRAM (se considerarmos $g$ constante) pode ser simulado otimamente no modelo BSP.

Com relação ao modelo QRQW PRAM, a simulação é possível mas a outros custos. Dado um algoritmo QRQW PRAM que possui tempo de execução $T(n)$ e custo $C(n)$, sua simulação no modelo BSP terá tempo de execução $O\left(\frac{C(n)}{p} + T(n) \times \log\ p\right)$ [GMR94c].

Apesar desta possível simulação de um modelo pelo outro, o modelo BSP é considerado de mais baixo nível que o QRQW PRAM, pois supõe a existência de arquiteturas que implementem os mecanismos usados no modelo, como a barreira de sincronização [Val90], enquanto que o modelo QRQW PRAM ou outra extensão do modelo PRAM não faz imposições a respeito de mecanismos no nível arquitetural.

A suposição de arquiteturas que implementem as características necessárias ao modelo BSP é talvez seu principal problema. Atualmente não existe nenhuma arquitetura

com características do BSPC[2]. Mesmo assim acreditamos que sua construção deverá ter custos menores que a de uma arquitetura baseada em PRAM, já que no modelo BSP não possuímos problemas relacionados a memória compartilhada e com pequena granularidade de sincronização. Apesar disso, Valiant nos afirma que se estes custos forem pagos, novas máquinas paralelas com melhores níveis de eficiência, flexibilidade e programabilidade podem ser alcançadas [Val90, McC94b].

Atualmente o modelo BSP tem sido bastante estudado em diversos grupos de pesquisa. Está sendo atualmente desenvolvido em Oxford uma linguagem baseada no modelo BSP; este desenvolvimento já se encontra bastante avançado e já existem várias rotinas implementadas em uma biblioteca chamada de MLIB [BM94]. Uma boa explicação deste projeto pode ser encontrado em [McC94a]. O modelo BSP é pesquisado como parte do projeto *ESPRIT 9072 - GEPPCOM (Foundations of General Purpose Parallel Computing)*, onde o objetivo final é construção de software paralelo que possa ser transportável e executável com boa performance em arquiteturas de memória distribuída, memória compartilhada ou rede de estações de trabalho. Também é do nosso conhecimento que o grupo de pesquisa responsável pela proposta do modelo H-PRAM [HR92] possui interesse em vários aspectos do modelo BSP, objetivando verificar as melhores características deste modelo e, se possível, incluí-las no H-PRAM.

### 9.1.3 LogP

Diferente de todos os modelos apresentados, o LogP é um modelo de bastante baixo nível e conseqüentemente de difícil uso quando comparado como o PRAM.

Nós tivemos a oportunidade de observar que o projeto de um algoritmo no modelo LogP não é uma tarefa trivial. Seus proponentes defendem que, apesar desta dificuldade, o modelo serve como um bom meio de desenvolvimento de algoritmos para as máquinas paralelas disponíveis no mercado.

Um aspecto importante do modelo LogP é o seu direcionamento para as máquinas pertencentes a convergência. Apesar de previsões [CKP+93, Pat93, McC94b], não podemos seguramente afirmar que elas ocorrerão. O mercado é muito instável e sujeito a diversos fatores que podem levar os projetos de máquinas paralelas comerciais a seguirem outras direções. Além disso nada garante que todas as máquinas seguirão esta convergência e caso isto não ocorra teremos a necessidade de outros modelos direcionados a estas arquiteturas.

Ao ser comparado com o modelo BSP, que de certa forma já é de nível mais baixo que outros modelos, é que notamos o quanto o modelo LogP é de baixo nível.

Diferentemente do modelo BSP, o LogP não possui qualquer necessidade de sincronização fixa; caso seja necessária, a sincronização é efetuada através de passagens de mensagens. Os proponentes do modelo LogP, apesar de admitirem que o modelo BSP serviu

[2] BSP *computer* [Val90]

como base para sua proposta, fazem uma série de críticas a este modelo. Especificamente sobre a questão da sincronização do modelo BSP, o fato de este modelo supor a existência de mecanismos especiais de *hardware* o tornam menos realista que o modelo LogP. Nós particularmente não entendemos que o fato da não existência de tais mecanismos de hardware possa invalidar a proposta do modelo BSP. Na seção 8.1.4 pudemos observar que mesmo o modelo LogP necessita de sincronizações. O algoritmo para TRF exige uma barreira de sincronização. Desta forma, assim como no modelo LogP, o mecanismo de barreira de sincronização, suposto no BSP, pode ser simulado através de passagem de mensagens e acreditamos que até com custos semelhantes. O fato de esta sincronização ser periódica no modelo BSP pode até ter um custo elevado quando computada todas as sincronizações, mas por outro lado, temos uma maior facilidade de programação.

Outra crítica feita ao modelo BSP é que ele utiliza poucos parâmetros na modelagem das máquina paralelas, pois seriam necessários alguns mais, introduzidos no modelo LogP, para se poder fazer uma boa modelagem do mundo real. Esta crítica não nos parece bem fundamentada, principalmente pelo fato de existir uma certa contradição quando é afirmado na proposta do modelo LogP que, felizmente para os usuários, em muitas aplicações alguns parâmetros podem ser desprezados. Se isto é verdade, seria necessário um estudo mais aprofundado para se saber qual a quantidade maior, se a dos problemas que necessitam de mais parâmetros ou se a dos que são bem modelados pelo modelo BSP.

Na proposta do modelo LogP ainda é feita outra crítica com relação ao fato de o modelo BSP permitir simulações de PRAM. É afirmado, e nos parece correto, que as constantes embutidas nesta simulação podem ser muito grandes e inviabilizar a otimalidade proposta. De qualquer forma para uma parte dos problemas as constantes podem ser pequenas viabilizando a simulação. Achamos que mesmo com constantes elevadas, é importante para um modelo de computação paralela permitir tal simulação, pois isto pode servir até para uma melhor adaptação por parte dos usuários. Esta possibilidade de simulação de PRAM não existe, ou não foi apresentada juntamente com a proposta do modelo LogP, pelo fato de este estar muito distante do modelo PRAM.

Apesar de todas as críticas que fizemos ao modelo LogP, achamos que a proposta do modelo é válida e tem grande importância, principalmente pelo fato de ela se ater a uma questão atual, pois os problemas necessitam ser resolvidos e não tínhamos uma boa proposta para resolvê-los.

## 9.2 Contribuições

Algumas contribuições deste trabalho são:

- comparação teórica de vários modelos de computação paralela;
- comparação de classes de arquiteturas paralelas;

- exposição organizada dos problemas existentes em computação paralela;
- comparação entre computação seqüencial e paralela.

## 9.3 Futuros Trabalhos

Algumas extensões possíveis deste trabalho são:

- estudar de um novo problema profundamente e projetar algoritmos paralelos usando os três modelos principais abordados;
- implementar os algoritmos usando uma linguagem paralela conhecida para verificação da realidade do modelo;
- verificar experimentalmente a proximidade entre o modelo LogP e as arquiteturas da convergência.

# Bibliografia


[ACS89] Alok Aggarwal, Ashok K. Chandra, e Marc Snir. On communication latency in PRAM computations (preliminary version). In *Proc. of ACM Symp. on Parallel Algorithms and Architectures*, pp. 11–21, 1989.

[AG94] George S. Almasi e Allan Gottlieb. *Highly Parallel Computing.* Series in Computer Science. Benjamin/Cummings, 2a. edição, 1994.

[AHU74] A. V. Aho, J. E. Hopcroft, e J. D. Ullman. *The Design and Analysis of Computer Algorithms.* Addison-Wesley, 1974.

[Akl89] Salim Akl. *The Design and Analysis of Parallel Algorithms.* Prentice-Hall, 1989.

[AKP93] Grama Y. Ananth, Vipin Kumar, e Panos Pardalos. Parallel processing of discrete optimization problems. Technical Report TR-93-38, DIMACS - Rutgers University, 1993.

[AS87] B. Awerbuch e Y. Shiloach. New connectivity and MSF algorithms for shuffle-exchange network and PRAM. *IEEE Tran. on Computers*, pp. 1258–1263, 1987.

[BM94] R. H. Bisseling e W. F. McColl. Scientific computing on bulk synchronous parallel architectures (short version). In B. Pehrson e I Simon, editores, *Proc. 13th IFIP World Computer Congress. Volume 1.* Elsevier, 1994. To appear.

[Bre74] R. P. Brent. The parallel evaluation of general arithmetic expressions. *Journal of the ACM*, 21(2):201–206, 1974.

[Cam94] Duncan K. G. Campbell. Towards a unified parallel architecture class. Technical Report TR-287, University of Exeter - Departament of Computer Science, Janeiro de 1994.

[CKP+93] 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 *Proc. of 4th*

*ACM PPOPP*, pp. 1– 12, Maio de 1993. Também como Technical Report UCB/CSD 92/713 - University of California, Berkeley - Computer Science Division (EECS).

[CLR90] Thomas H. Cormen, Charles E. Leiserson, e Ronald L. Rivest. *Introduction to Algorithms.* (Mit Electrical Engineering and Computer Science Series). MIT Press, 1990.

[CT65] James W. Cooley e John W. Tukey. An algorithm for the machine calculation of complex Fourier series. *Math. Comp.*, 19:297–301, 1965.

[CZ89] Richard Cole e Ofer Zajicek. The APRAM: Incorporating asynchrony into the PRAM model. In *Proc. of Symp. on Parallel Algorithms and Architectures 89*, pp. 1–10, 1989.

[Fly72] Michael J. Flynn. Some computer organizations and their effectiveness. *IEEE Transactions on Computers*, 21(9):948–960, Setembro de 1972.

[GGK+83] A. Gottlieb, R. Grishman, C. P. Kruskal, K. P. McAuliffe, L. Rudolph, e M. Snir. The NYU ultracomputer - designing an MIMD, shared-memory parallel machine. *IEEE Transactions on Computers*, 32:75–89, 1983.

[Gib89] Phillip B. Gibbons. A more practical PRAM model. In *Proc. of 1st Symp. on Parallel Algorithms and Architectures*, Junho de 1989.

[GKLP93] Phillip B. Gibbons, Richard Karp, Charles E. Leiserson, e Gregory Papadopoulos, editores. *Proc. of the DIMACS Workshop on Models, Architectures and Technologies for Parallel Computation*, Setembro de 1993. DIMACS Technical Report 93-87.

[GMR94a] Phillip B. Gibbons, Yossi Matias, e Vijaya Ramachandran. Efficient low-contention parallel algorithms. Technical report, AT&T, Setembro de 1994.

[GMR94b] Phillip B. Gibbons, Yossi Matias, e Vijaya Ramachandran. Efficient low-contention parallel algorithms. In *Proc. 5th ACM-SIAM Symp. on Discrete Algorithms*, Janeiro de 1994.

[GMR94c] Phillip B. Gibbons, Yossi Matias, e Vijaya Ramachandran. The QRQW PRAM: Accounting for contention in parallel algorithms. In *Proc. 5th ACM-SIAM Symp. on Discrete Algorithms*, Janeiro de 1994.

[GMR94d] Phillip B. Gibbons, Yossi Matias, e Vijaya Ramachandran. The queue-read queue write PRAM model: Accounting for contention in parallel algorithms. Technical report, AT&T, Setembro de 1994.

[GR88] Allan M. Gibbons e W. Rytter. *Efficient Parallel Algorithms.* Cambridge University Press, 1988.

[GV93] Alexandros V. Gerbessiotis e Leslie G. Valiant. Direct bulk-syncronous parallel algorithms. *Proceedings of SWAT 92, in Lectures Notes in Computer Science*, 621:1–18, 1993. Também como Techinical Report TR-10-92 (Extended Version) - Harvard University - Center for Research in Computing Technology.

[Hay88] John P. Hayes. *Computer Architecture and Organization.* McGraw-Hill Book Company, 2a. edição, 1988.

[HR92] Todd Heywood e Sanjay Ranka. A practical hierarchical model of parallel computation - The model. *Journal of Parallel and Distributed Computing*, 16:212–232, 1992.

[HRD93a] Tsan-Sheng Hsu, Vijaya Ramachandran, e Nathaniel Deam. Implementation of parallel graph algorithms on the MasPar. Technical Report TR-93-38, University of Texas at Austin, Julho de 1993.

[HRD93b] Tsan-Sheng Hsu, Vijaya Ramachandran, e Nathaniel Deam. Implementation of parallel graph algorithms on a massively parallel SIMD computer with virtual processing. Technical Report TR-93-14, University of Texas at Austin, Julho de 1993.

[Jáj92] Joseph Jájá. *An Introduction to Parallel Algorithms.* Addison-Wesley Publishing Company, 1992.

[Kar91] Richard Karp. Parallel combinatorial computing. Technical Report TR-91-006, International Computer Science Institute - ICSI, Janeiro de 1991.

[KFW94] R. Kent Koeninger, Mark Furtney, e Martin Walker. A shared memory MPP from cray research. *Digital Technical Journal*, 6(2):8–21, Spring 1994 de 1994.

[KLH93] Richard M. Karp, Michael Luby, e Friedhelm Meyer Auf Der Heide. Efficient PRAM simulation on a distributed memory machine. Technical Report TR-93-040, International Computer Science Institute - ICSI, Agosto de 1993.

[Knu81] Donald E. Knuth. *Seminumerical Algorithms.* Series in Computer Science and Information Processing. Addison-Wesley Publishing Company, 2a. ed. edição, 1981.

[KR90] Richard Karp e Vijaya Ramachandran. *Parallel Algorithms for Shared-Memory Machines. In Handbook of Theoretical Computer Science*, volume A, capítulo 17, pp. 869–941. MIT Press/Elsevier, 1990.

[KSSS93] Richard Karp, Abhijit Sahay, Eunice Santos, e Klaus Erik Schauser. Optimal broadcast and summation in the LogP model. In *Proc. of Symp. on Parallel Algorithms and Architectures*, pp. 142–153, Junho de 1993.

[LM93] Thomas J. LeBlanc e Evangelos P. Markatos. Shared memory vs. message passing in shared memory multiprocessors. Technical report, Computer Science Department - University of Rochester, 1993.

[Mac92] N. B. MacDonald. An overview of SIMD parallel systems. Technical Report EPCC-TR92-18, University of Edinburgh - Parallel Computing Centre, 1992.

[McC93] W. F. McColl. *General Purpose Parallel Computing. In Lectures in Parallel Computation. Gibbons, P. (ed.)*, capítulo 13, pp. 337–391. Cambridge University Press, 1993.

[McC94a] W. F. McColl. BSP programming. In G. Blelloch, M. Chandy, e S. Jagannathan, editores, *Proc. DIMACS Workshop on Specification of Parallel Algorithms, Princeton, May 9-11, 1994*. American Mathematical Society, 1994. To Appear.

[McC94b] W. F. McColl. Scalable parallel computing: A grant unifield theory and its practical development. In B. Pehrson e I. Simon, editores, *Proc. 13th IFIP World Computer Congress. Volume 1 (Invited Paper)*. Elsevier, 1994. To appear.

[MV84] Kurt Mehlhorn e Uzi Vishkin. Randomized and deterministic simulations of PRAMs by parallel machines with restricted granularity of parallel memories. *Acta Informatica*, 21:339–374, 1984.

[Pat93] David Paterson. Observations on massively parallel processors. In Gibbons *et. al* [GKLP93]. DIMACS Technical Report 93-87.

[PS85] Franco P. Preparata e Michael I. Shamos. *Computational Geometry - An Introduction*. Texts and Monographs in Computer Science. Springer-Verlay, 1985.

[Ran87] A. Ranade. How to emulate shared memory. In *Proc. 28th IEEE Symp. on Foundation of Computer Science*, pp. 185–194, 1987.

[RO93] Gowri Ramanathan e Joel Oren. Survey of comercial parallel machines, 1993. Não publicado.

[Sah93] Abhijit Sahay. Hiding communication costs in bandwidth-limited parallel FFT computation. Technical report, University of California, Berkeley, Janeiro de 1993.

[SBM62] D. L. Slotnick, W. C. Borck, e R. C. McReynolds. The SOLOMON computer. In *AFIPS Conference Proceedings*, volume 22, pp. 97–107, 1962.

[Sed88] Robert Sedgewick. *Algorithms*. Addison-Wesley Publishing Company, 1988.

[Val90] Leslie G. Valiant. A bridging model for parallel computation. *Communications of ACM*, 33(8):103–111, Agosto de 1990.

[Val92] Leslie G. Valiant. A combining mechanism for parallel computers. Technical Report TR-24-92, Harvard University - Center for Research in Computing Technology, Novembro de 1992.

[Val93] Leslie G. Valiant. Why BSP computers ? *IEEE Computer Society - Press Reprint*, Abril de 1993.

[Vis92] Uzi Vishkin. A case for the PRAM as a standard programmer's model. Technical Report UMIACS-TR-92-130, University of Maryland, Novembro de 1992.

[Vis93] Uzi Vishkin. On the case for having the PRAM as a standard programmer's model. In Gibbons et alli [GKLP93]. DIMACS Technical Report 93-87.

[Yap94] Chee K. Yap. Fundamental problems in algorithmic algebra. 1994.