Algoritmos Paralelos usando CGM/PVM/MPI: Uma Introdução

Edson Norberto Cáceres[1]
Dept. de Computação e Estatística
Universidade Federal de Mato Grosso do Sul
79070-900 Campo Grande, MS
e
Dept. de Ciência da Computação
Instituto de Matemática e Estatística
Universidade de São Paulo
05508-900 São Paulo, SP
`edson@dct.ufms.br, caceres@ime.usp.br`

Henrique Mongelli
Dept. de Computação e Estatística
Universidade Federal de Mato Grosso do Sul
79070-900 Campo Grande, MS
`mongelli@dct.ufms.br`

Siang Wun Song[2]
Dept. de Ciência da Computação
Instituto de Matemática e Estatística
Universidade de São Paulo
05508-900 São Paulo, SP
`song@ime.usp.br`

[1]Parcialmente financiado pelo PRONEX SAI e FAPESP Proc. No. 1997/10982-0

[2]Parcialmente financiado pela FAPESP Proc. No. 1998/06138-2, CNPq/NSF Collaborative Research Program Proc. No. 68.0037/99-3, e CNPq Proc. No. 52.3778/96-1 e 46.1230/00-3

Capítulo 1

Introdução

Objetivos

- Introduzir o conceito de um Sistema de Computação Paralela e Distribuída.
- Introduzir o conceito de avaliação de um Algoritmo Paralelo e Distribuído.
- Descrever os principais Modelos de Computação Paralela e Distribuída (Memória Compartilhada, Rede e Realístico).
- Apresentar exemplos de Algoritmos Paralelos para os Modelos PRAM, Rede e BSP/CGM.
- Analisar a eficiência desses Algoritmos nos respectivos Modelos.
- Efetuar a comparação entre os Modelos apresentados.
- Introduzir o PVM e o MPI, que serão utilizados para implementar os algorimos para o modelo BSP/CGM.

Neste capítulo apresentamos uma visão geral do minicurso, destacando os principais modelos de Computação Paralela e Distribuída, Algoritmos Paralelos, Complexidade, o conceito da Troca de Mensagens em um Sistema de Computação Paralelo e uma uma descrição dos aspectos funcionais principais do PVM e MPI.

1. Sistemas de Computação Paralela e Distribuída

Um **Sistema de Computação Paralela e Distribuída** consiste de uma coleção de elementos de computação (processadores), geralmente do mesmo tipo, interconectados de acordo com uma determinada topologia para permitir a coordenação de suas atividades e troca de dados. Esses processadores trabalham simultaneamente, de forma coordenada, com o objetivo de resolver um problema específico. Esses sistemas surgiram em função da necessidade de solucionar problemas onde a computação sequencial não consegue obter uma solução dentro de tempos razoáveis. Utilizaremos o termo **Sistema de Computação Paralela** sempre que nos referirmos a um sistema de computação paralela e distribuída.

Os sistemas de computação paralela podem ser classificados de acordo com diversos aspectos. A classificação original dos sistemas de computação paralela é popularmente conhecida como **taxionomia de Flynn**. Em 1966, Michael Flynn classificou os sistemas de computação paralela de acordo com o número de fluxos de instruções e o número dos fluxos de dados. A máquina clássica de von Neumann tem um único fluxo de instruções e um único fluxo de dados, e portanto é identificada como uma máquina **single-instruction single-data** (**SISD**). No extremo oposto está o sistema **multiple-instruction multiple-data** (**MIMD**), na qual uma coleção de processadores autônomos operam seus próprios fluxos de dados. Na taxionomia de Flynn, esta é a arquitetura mais abrangente. Entre os sistemas SISD e MIMD estão os sistemas **single-instruction multiple data** (**SIMD**) e o **multiple-instruction single-data** (**MISD**). Vamos apresentar alguns detalhes dos modelos SIMD e MIMD.

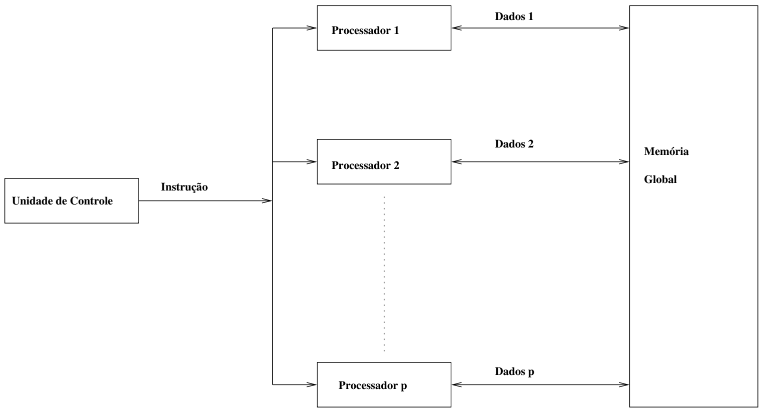


Figura 1.1: O Modelo SIMD

A Figura 1.1 ilustra o modelo SIMD, onde os p processadores executam em paralelo (e ao mesmo tempo) o mesmo programa. Existe uma única unidade de controle, que passa a instrução a ser executada para todos os processadores ativos. Em cada instante, todos os processadores ativos executam necessariamente a mesma instrução sobre dados possivelmente diferentes.

No modelo MIMD, exemplificado pela Figura 1.2, os p processadores executam em paralelo (ao mesmo tempo) instruções que podem ser distintas, estão interligados através de uma rede de interconexão. Cada processador é identificado através de um número inteiro e além disso possui associado a ele uma memória local, acessível apenas a ele e não existe nenhuma memória compartilhada. A operação em uma rede pode ser síncrona ou assíncrona.

Uma discussão mais completa da taxionomia de Flynn pode ser encontrada em [20].

Existem muitas outras formas de classificar sistemas de computação paralela, entre eles, destacamos os seguintes: **dispersão dos processadores**, **estrutura de interconexão**, e **sincronismo**.

Do ponto de vista de **dispersão**, os sistemas são classificados como **sistemas com dispersão geográfica**, constituindo as **redes de computadores** e os **sistemas distribuídos**, e como **sistemas confinados**, constituindo as **máquinas paralelas**.

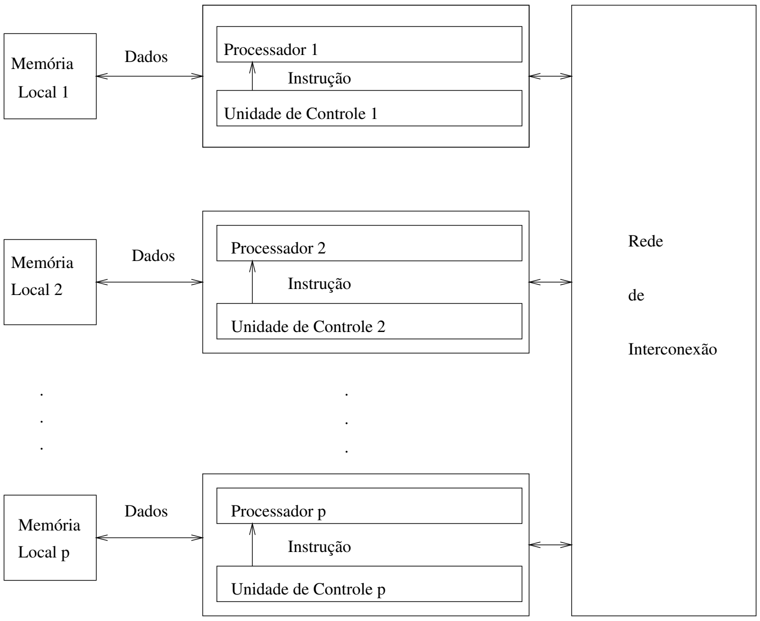


Figura 1.2: O Modelo MIMD

Uma **rede de computadores** é definida como um conjunto de computadores autônomos interligados por um subsistema de comunicação, envolvendo canais de comunicação. Um **sistema distribuído** é um sistema composto por um hardware baseado em múltiplos processadores independentes, interligados através de uma rede de interconexão, e por um software integrado, explorando o compartilhamento implícito de recursos. Uma **máquina paralela** é simplesmente uma coleção de processadores, geralmente do mesmo tipo, interconectados de maneira a permitir a coordenação de suas atividades e troca de dados.

Segundo a **estrutura de interconexão**, os sistemas de computação paralelos são classificados de acordo com a topologia utilizada.

De acordo com o **sincronismo**, podem ser **síncronos**, aqueles em onde todos os processadores ativos operam sincronizadamente sob o controle de um único relógio global comum, compartilhado pelos processadores, ou **assíncronos**, aqueles em que existe uma ausência completa de uma base de tempo comum a todos os processadores que compõem o sistema e cada processador opera sobre um relógio separado. Neste modo, é responsabilidade do programador indicar os pontos de sincronização apropriados sempre que necessário. Mais precisamente, se dados precisam ser acessados por um processador, é responsabilidade do programador garantir que os valores corretos serão obtidos, uma vez que o valor de uma variável, compartilhada ou local, muda dinamicamente durante a execução do algoritmo.

Independente da classificação, temos que, em qualquer sistema de computação paralela, os processadores estão sempre ativos ou inativos. Todos os processadores ativos estão executando alguma tarefa para colaborar para a solução de um único problema. O acesso do processador i a algum dado calculado pelo processador j, é feito através da comunicação entre esses processadores. A maneira como se dá a comunicação entre processadores depende do arquitetura da **rede de interconexão** do sistema.

Existem vários tipos de arquiteturas paralelas que implementam diferentes sistemas de computação paralela. Para cada arquitetura paralela, temos modelos distintos de desenvolvimento de algoritmos paralelos. Estes modelos, nos quais baseamos o desenvolvimento de algoritmos, são denominados **modelos de computação paralela e distribuída**.

Sistemas de computação paralela exigem naturalmente novos algoritmos e programas, também paralelos. Diferentemente do modelo sequencial, a computação paralela não possui um modelo algorítmico que seja amplamente aceito. A eficiência dos algoritmos paralelos dependem de um conjunto de fatores que são diretamente ligados à arquitetura da máquina. Entre estes fatores podemos destacar:

- Concorrência computacional
- Alocação e escalonamento de processos
- Comunicação
- Sincronização

Abordaremos três modelos de computação paralela: **memória compartilhada**, **rede** e **realístico**. Os algoritmos estudados serão projetados para o modelo realístico (**BSP/CGM**). A implementação dos algoritmos será feita num ambiente de computação paralela distribuída (rede de estações de trabalho) utilizando bibliotecas (**PVM - Parallel Virtual Machine** e a **MPI - Message-Passing Interface**) que implementam **troca de mensagens** e a linguagem de programação C/C++.

2. Algoritmos Paralelos e Complexidade

O primeiro passo no projeto de um algoritmo paralelo é a decomposição do problema em problemas menores. Dessa forma, os problemas menores são atribuídos aos processadores para trabalharem de forma simultânea. Basicamente, existem dois tipos de decomposição **decomposição do domínio** e **decomposição funcional**.

Na **decomposição do domínio**, também conhecida como *paralelismo de dados*, os dados são divididos em partes com aproximadamente o mesmo tamanho e então atribuídos aos processadores. Então, cada processador trabalha somente no conjunto de dados que lhe foi atribuído. Os processos podem ter a necessidade se comunicarem periodicamente para a troca de dados. O paralelismo de dados propicia a vantagem de manter um único fluxo de controle. Um algoritmo que usa o paralelismo dos dados consiste de uma sequencia de instruções elementares aplicadas aos dados: uma instrução só é iniciada se a instrução anterior terminou a sua execução. O modelo **Single-Program Multiple-Data** (**SPMD**) segue essa idéia, onde o código a ser executado é igual para todos os processadores.

Frequentemente, a estratégia de decomposição do domínio não leva ao algoritmo mais eficiente para o programa paralelo. Este é o caso quando os conjuntos de dados atribuídos aos diferentes processadores requerem tempos de execução muito diferentes entre si. O desempenho do algoritmo é então limitado pela velocidade do processo mais lento. Muitos processadores podem ficar ociosos durante um grande intervalo de tempo. Neste caso, a **decomposição funcional** ou *paralelismo de tarefas* faz mais sentido do que a decomposição do domínio. No paralelismo de tarefas, o problema é decomposto em um grande número de tarefas menores e então, as tarefas são atribuídas aos processadores na medida em que eles ficam disponíveis. Os processadores que rapidamente terminam suas tarefas recebem mais trabalho. O paralelismo de tarefas é implementado com o paradigma **cliente-servidor**, onde as tarefas são alocadas a um grupo de processos escravos através de um processo mestre que também pode executar algumas tarefas.

Nos algoritmos sequenciais, o modelo **RAM** (**Random Access Machine**) é bastante próximo da forma de descrever os algoritmos e da arquitetura de Von Neumam. Logo, a avaliação da execução de um algoritmo sequencial está intimamente ligado à análise de sua complexidade. Como vimos na Seção 1., um sistema de computação paralela depende de um grande número de parâmetros, tais como o número de processadores, as capacidades de memórias locais, esquema de comunicação e os protocolos de sincronização, fazendo com que o projeto, avaliação e análise de algoritmos paralelos seja bem mais complexa do que no modelo sequencial. Para iniciar, introduziremos duas maneiras gerais comumente usadas para avaliar a execução de um algoritmo paralelo.

Seja P um dado problema computacional e seja n o tamanho da entrada. Denotemos a complexidade sequencial de P por $T^*(n)$. Existe um algoritmo sequencial que resolve P com esse tempo mínimo, e ainda mais, podemos provar que nenhum algoritmo sequencial pode resolver P de forma mais rápida. Seja A um algoritmo paralelo que resolve P num tempo $T_p(n)$ em um computador paralelo que com p processadores.

Definimos o **speedup** alcançado por A como

$$S_p(n) = \frac{T^*(n)}{T_p(n)}$$

O valor $S_p(n)$ mede o speedup obtido por um algoritmo A quando p processadores estão disponíveis para utilização. É fácil verificar que o melhor speedup possível de ser alcançado é $S_p(n) \leq p$ O objetivo é projetar algoritmos paralelos que alcancem $S_p(n) \simeq p$.

Na realidade, existem vários fatores que introduzem ineficiência nos algoritmos paralelos. Dentre esse fatores destacamos os atrasos que são introduzidos pela comunicação, a sobrecarga ocorrida na sincronização das atividades dos vários processadores e no controle do sistema.

Note que $T_1(n)$, é o tempo do algoritmo paralelo A quando o número p de processadores é igual a 1, não é necessariamente o mesmo que $T^*(n)$; portanto, o speedup é medido relativamente de acordo com o melhor algoritmo sequencial possível. É comum considerar o $T^*(n)$ como o tempo do melhor algoritmo sequencial conhecido, sempre que a complexidade do problema não é conhecida.

Outra medida de execução do algoritmo paralelo A é a **eficiência**, definida por:

$$E_p(n) = \frac{T_1(n)}{pT_p(n)}$$

Esta medida proporciona uma indicação da utilização efetiva de p processadores relativa ao algoritmo dado. Um valor de $E_p(n)$ aproximadamente igual a 1, para algum p, indica que o algoritmo A executa aproximadamente p vezes mais rápido usando p processadores do que o faria usando apenas 1 processador. Segue que cada um dos processadores está trabalhando durante cada passo de tempo relativo ao total do trabalho requerido pelo algoritmo A.

Existe um limite no tempo de execução denotado por $T_\infty(n)$, através do qual o algoritmo não pode executar nada mais rápido, independente do número de processadores. Portanto, $T_p(n) \leq T_\infty(n)$, para qualquer valor de p, e a eficiência $E_p(n)$ satisfaz $E_p(n) \leq T_1(n)/pT_\infty(n)$. Entretanto a eficiência de um algoritmo degrada rapidamente enquanto p cresce acima de $T_1(n)/T_\infty(n)$.

Para descrever os algoritmos paralelos utilizaremos as instuções de um *Algol like* (programação estruturada) com uma construção adicional:

para $x \in X$ **em paralelo faça**

```
instrução 1;
⋮
instrução k;
```

Onde X é um conjunto e cada elemento $x \in X$ é associado a um processador. No total são utilizados $|X|$ processadores. Cada processador executada as instruções 1 até k, sequencialmente, em paralelo de forma sincronizada com os demais processadores, para cada x, ou seja os processadores só iniciam a execução da instrução i após todos os processadores terem terminado de processar a instrução $i - 1$. Essa construção é um comando de alto nível, ou seja, cria processos e aloca-os nos processadores.

Vimos, nesta seção, algumas formas de avaliar a execução de algoritmos paralelos. Para analisar a complexidade de algoritmos paralelos, necessitamos definir um modelo que incorpore os diferentes aspectos presentes num sistema de computação paralela.

3. Modelos de Computação Paralela e Distribuída

O principal objetivo do projeto de um algoritmo paralelo é o de obter um desempenho superior com relação à versão sequencial. Com isto em mente, existem diversos fatores que necessitamos considerar para projetar nossos algoritmos paralelos com o intuito de obter o melhor desempenho possível dentro das restrições do problema sendo solucionado. Esses fatores são: **balanceamento de carga**, **minimização da comunicação** e **sobreposição de comunicação e computação**.

Balanceamento de carga é a tarefa de dividir equitativamente o trabalho entre os processadores disponíveis.

O tempo de execução total do algoritmo é a principal preocupação da programação paralela pelo fato que de ser um componente essencial na comparação e melhoramento dos programas. O tempo de execução de um programa é composto de três componentes: **tempo de computação**; **tempo ocioso** e **tempo de comunicação**.

O **tempo de computação** é o tempo gasto na execução das computações sobre os dados do problema. Como vimos no seção 2., o ideal é que se tivermos p processadores trabalhando na execução de um problema, a execução do algoritmo será finalizada em $1/p$-ésimas unidades de tempo do algoritmo sequencial. Nesse caso, todo o tempo dos processadores seria gasto na computação.

Denominamos **tempo ocioso** o tempo que um processador gasta aguardando dados de outros processadores. Durante esse tempo, os processadores não fazem nenhum trabalho útil.

Finalmente, o **tempo de comunicação** é o tempo utilizado pelos processos para enviar e receber mensagens. O **custo do tempo de comunicação** pode ser medido em termos da **latência** e da **largura da banda**. A **latência** é o tempo utilizado na preparação do pacote para comunicação, e **largura da banda** é a velocidade real de transmissão, ou *bits* por unidade de tempo. Programas sequenciais não utilizam comunicação entre os processos, portanto na **minimização da comunicação** necessitamos diminuir o tempo de comunicação. Se compararmos o tempo gasto de comunicação com o tempo total de execução algoritmo, temos que a redução dos custos de comunicação é uma das melhores formas de se obter aumento do desempenho dos algoritmos.

Existem diversas formas de minimizar o tempo ocioso dentro dos processos e uma delas é a **sobreposição de comunicação e computação**, que consiste na atribuição de uma ou mais novas tarefas para manter ocupado um processo enquanto esse processo aguarda a finalização de uma determinada comunicação. Isso é bastante difícil de ser obtido na prática.

Na concepção de um modelo para algoritmos paralelos, temos que levar em conta todos os aspectos descritos acima, para isto precisamos definir modelos sob os quais os algoritmos paralelos serão desenvolvidos. No caso sequencial, o modelo **RAM** (**Random Access Memory**), utilizado para o projeto e análise de algoritmos sequenciais, consiste de uma unidade central de processamento com uma memória de acesso aleatório anexada a ela. O conjunto típico de instruções para este modelo inclui a leitura e escrita na memória, e operações aritméticas e lógicas básicas. O acesso a qualquer posição da memória pode ser efetuado em tempo constante. No modelo para a computação paralela, a situação é mais complexa em virtude da existência de um conjunto de processadores interconectados e memórias locais anexadas a esses processadores. A garantia de que o acesso a qualquer posição da memória pode ser feito em tempo constante fica condicionada à existência de uma memória global e a eficiência da rede de interconexão que providencia a comunicação entre os processadores e a memória global. No caso da não existência dessa memória global o tempo de acesso dependerá da latência e largura de banda da rede interconexão. Além dos aspectos abordados até aqui, um modelo algorítmico para descrever e analisar algoritmos paralelos também deve satisfazer os seguintes requisitos.

Simplicidade: o modelo deve ser simples o suficiente para descrever algoritmos paralelos facilmente, para analisar matematicamente medidas de execução importantes como velocidade, comunicação e utilização da memória. E ainda, o modelo não deve ser vinculado a nenhuma classe particular de arquiteturas, e portanto deve ser independente de hardware.

Implementabilidade: o algoritmo paralelo desenvolvido para o modelo deve ser facilmente implementável nos computadores paralelos existentes.

Uma grande variedade de modelos foi proposta para a computação paralela. Não há um modelo que seja amplamente utilizado (da mesma forma que o modelo sequencial RAM) para o projeto e análise de algoritmos paralelos. Vamos apresentar três modelos que são bastante utilizados no desenvolvimento e análise de algoritmos paralelos. Posteriormente vamos discutir seus méritos e deficiências. Os modelos são:

1. Modelo de Memória Compartilhada.
2. Modelo de Rede
3. Modelo Realístico

Vamos apresentar diversos algoritmos para esses modelos, analisar as complexidades e fazer uma breve comparação entre esses modelos.

3.1. O Modelo de Memória Compartilhada

O **modelo de memória compartilhada** consiste de um conjunto de processadores, cada qual com uma memória local própria, podendo executar seus próprios programas locais, e a comunicação entre os processadores é feita através de uma unidade de memória compartilhada. Cada processador é unicamente identificado por um índice (número inteiro), o qual é utilizado localmente. Cada posição da memória compartilhada também chamada de memória global possui uma identificação (endereço) e pode ser acessada por qualquer processador. A cada instante os processadores podem estar em dois estados: ativos ou inativos. Cada processador possui uma unidade de controle, que passa a instrução a ser executada no caso do processador estar ativo. Em cada instante, todos os processadores ativos executam instruções (diferentes ou não), sobre dados possivelmente diferentes. Os dados calculados pelo processador i e que serão utilizados por outros processadores serão escritos pelo processador i na memória compartilhada (utilizando variáveis globais/compartilhadas). Esse modelo oferece uma estrutura bastante atrativa para o desenvolvimento de técnicas algorítmicas para computação paralela. A Figura 1.3 ilustra o modelo de memória compartilhada.

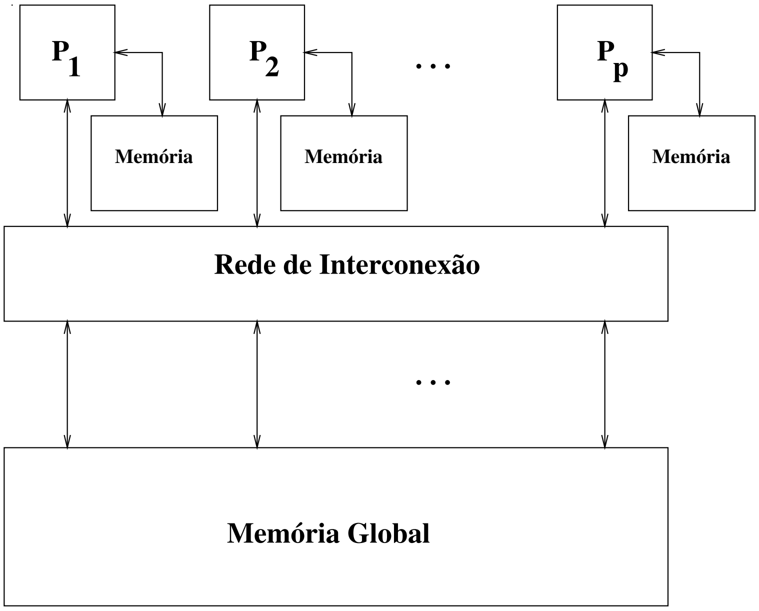


Figura 1.3: O modelo de memória compartilhada

Existem dois modos básicos de operação em memória compartilhada: síncrono e assíncrono. Um modelo de máquina síncrona padrão é a **Parallel Random Access Machine** (**PRAM**). Desde que cada processador pode executar uma instrução ou operar dados diferentes daqueles executados ou operados por algum outro processador durante uma unidade de tempo, nosso modelo de memória compartilhada é do tipo MIMD. Apesar disso, os algoritmos desenvolvidos para o modelo de memória compartilhada, mais especificamente para o modelo PRAM têm sido do tipo SIMD.

Antes de apresentarmos o próximo exemplo, vamos aumentar nossa linguagem algorítmica as duas construções seguintes:

1. `global read(X,Y)`
2. `global write(U,V)`

O efeito da instrução `global read` é mover o bloco de dados X, armazenado na memória compartilhada para a variável local Y. Similarmente, o efeito da instrução `global write` é escrever o dado local U na variável compartilhada V.

Exemplo 1 Soma na PRAM. Dado um vetor A de $n = 2^k$ números, e uma PRAM com n processadores $P_1, P_2, \ldots, P_n$, desejamos computar a soma $S = A(1)+A(2)+\cdots+A(n)$. Cada processador executa o mesmo algoritmo, descrito aqui para o processador P_i. O Algoritmo 1 implementa esse exemplo.

Algoritmo 1 Soma na PRAM
Entrada: Um vetor A de ordem $n = 2^k$ armazenados na memória compartilhada de uma PRAM. As variáveis locais inicializadas são n e o número do processador i.
Saída: A soma das entradas de A armazenados na variável compartilhada S. O vetor A mantém seu valor inicial.

(1) `global read(A(i),a)`

(2) `global write(a,B(i))`

(3) **para** $h = 1$ **até** $\log n$ **faça**
  (3.1) **se** $i \leq n/2^h$ **então**
    (3.1.1) `global read(B(2i-1),x);`
    (3.1.2) `global read(B(2i),y);`
    (3.1.3) `z` $\leftarrow$ `x+y;`
    (3.1.4) `global write(z,B(i));`

(4) **se** $i = 1$ **então** `global write(z,S);`

— Fim do Algoritmo —

Durante os passos 1 e 2, uma cópia B de A é criada e é armazenada na memória compartilhada (podem ser acessadas por qualquer processador). O esquema de computação (passo 3) é baseado em uma árvore balanceada binária cujas folhas correspondem aos elementos de A. A Figura 1.4 ilustra o algoritmo para o caso onde $n = 8$. O processador responsável pela execução da operação é indicado abaixo do nó representando a operação. O algoritimo é síncrono, ou seja, em uma unidade de tempo cada processador tem a permissão de executar uma instrução ou permanecer inativo. Note que P_1, que é responsável pela alteração do valor de $B(1)$ e por escrever a soma em S, está sempre ativo durante a execução do algoritmo, enquanto P_5, P_6, P_7 e P_8 estão ativos apenas durante os passos 1 e 2. $\square$

Definimos a **complexidade de tempo paralelo** de um algoritmo PRAM como sendo o tempo total consumido pelo algoritmo (número de passos do algoritmo), onde

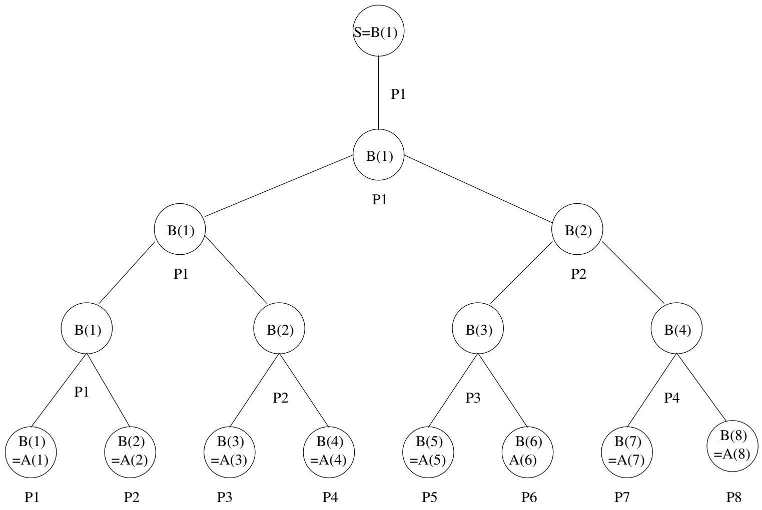


Figura 1.4: Soma na PRAM

cada computação efetuada em paralelo contribui com uma unidade para o tempo total, independente do número de processadores envolvidos e a **complexidade de processadores** como sendo o maior número de processadores envolvidos em qualquer passo do algoritmo. A **complexidade de espaço** é definida da mesma forma que no caso sequencial. Dado um algoritmo paralelo PRAM, medimos seu desempenho em termos da análise do pior caso. Seja Q um problema para o qual temos um algoritmo PRAM que é executado em tempo $T(n)$ (complexidade de tempo paralelo) usando $P(n)$ (complexidade de processadores) processadores, para uma instância de tamanho n. O produto tempo-processador $C(n) = T(n) \times P(n)$ (número de operações realizadas) representa o **custo** do algoritmo paralelo. É fácil verificar que um algoritmo paralelo PRAM pode ser convertido em um algoritmo sequencial que é executado em tempo $O(C(n))$. Para isso, simplesmente temos que simular os $P(n)$ processadores em $O(P(n))$ tempo, para cada $T(n)$ passos paralelos. Um algoritmo paralelo para o modelo PRAM é **eficiente** se é executado em tempo paralelo polilogaritmico ($O(\log^p n)$) usando um número polinomial de processadores (n^k), onde $p, k \in N$ e n é o tamanho da entrada do algoritmo. Um algoritmo paralelo PRAM é **ótimo** se o seu custo for igual ao tempo do melhor algoritmo sequencial para o problema. Um algoritmo ótimo pode deixar de se-lo, se for obtido um algoritmo sequencial mais eficiente. Um algoritmo é ótimo absoluto se seu custo for igual ao tempo do melhor algoritmo sequencial, e este tempo for igual ao limite inferior do problema.

Algoritmos Eficientes	*Tempo*	*Processadores*
Sequenciais Eficientes	polinomial	1
Paralelos Eficientes	polilogaritmo	polinomial
	$O(\log_2 n)$	n
	$O(\log_2^3 n)$	n^2
Paralelos Não Eficientes	$O(n)$	$O(\log_2 n)$
	$O(\log_2 n)$	2^n

Para simplificar a apresentação de algoritmos PRAM, vamos omitir os detalhes relativos às operações de acesso à memória. Uma instrução do tipo $A \leftarrow B + C$,onde A, B e C são variáveis compartilhadas, devem ser interpretadas como a seguinte sequencia de instruções.

```
global read(B,x);
global read(C,y);
z← x + y;
global write(z,A);
```

Exemplo 2 Vamos considerar novamente o problema de computar a soma S de $n = 2^k$ números armazenados em um vetor A utilizando a notação simplificada. O algoritmo apresentado anteriormente especificava o programa para ser executado por cada processador P_i, onde $1 \leq i \leq n$.

Algoritmo 2 Soma de n números
Eentrada: Um vetor de $n = 2^k$ números, armazenados em $A[1] \ldots A[n]$
Saída: A Soma $S = \sum_{i=1}^{n} A(1)$

(1) **para** $1 \leq i \leq n$ **em paralelo faça**
  (1.1) `B[i] ← A[i]`

(2) **para** $h \leftarrow 1$ **até** $\log n$ **faça**
  (2.1) **para** $1 \leq i \leq n/2^h$ **em paralelo faça**
    (2.1.1) `B[i] ← B[2i-1] + B[2i]`

(3) `S ← B[1]`

— Fim do Algoritmo —

Vamos agora analisar a complexidade desse algoritmo. O Passo 1 é executado em tempo $O(1)$. O Passo 2 é executado $\log n$ vezes, em cada execução gasta tempo paralelo $O(1)$. O Passo 3 é executado em tempo $O(1)$. Portanto, a complexidade desse algoritmo é tempo paralelo $O(\log n)$ com n processadores. O custo do algoritmo é $O(n \log n)$. Temos que o algoritmo é eficiente mas não é ótimo. □

Uma importante questão relacionada ao modelo PRAM diz respeito ao que acontece quando mais de um processador tenta, ao mesmo tempo, acessar a mesma célula da memória global (leitura e/ou escrita). As diferentes formas de resolver os conflitos de leitura ou escrita definem os modelos e submodelos do modelo PRAM. O modelo PRAM **Exclusive Read Exclusive Write** (**EREW**) não permite qualquer tipo de acesso simultâneo à uma posição da memória compartilhada. O modelo PRAM **Concurrent Read Exclusive Write** (**CREW**) permite que acessos simultâneos a uma posição da memória sejam feitos exclusivamente para leitura. Acessos simultâneos a uma posição da memória para leitura e escrita é permitido no modelo **Concurrent Read Concurrent Write** (**CRCW**). Existem vários submodelos do CRCW PRAM, eles diferem na forma como a escrita simultânea é tratada. Pode ser demonstrado de que um algoritmo eficiente para um dos modelos PRAM também é eficiente para qualquer outro modelo. Isso pode ser feito através da simulação entre os modelos. Já o conceito de algoritmo ótimo depende do modelo e submodelo considerado. Um algoritmo pode ser implementado em dois submodelos com complexidades diferentes. Para maiores detalhes com relação a modelos PRAM e simulação entre esses modelos, sugerimos uma leitura do trabalho de Karp e Ramachandran [16].

Uma pergunta que podemos fazer é o que aconteceria se o número de processadores disponíveis p fosse menor do que $P(n)$? Neste caso podemos aplicar o teorema de Brent que é utilizado para reduzir a complexidade de processadores de um algoritmo (aumentando a complexidade de tempo).

Teorema 1 (Brent) *Suponha que um problema possa ser resolvido através de um algoritmo paralelo, com complexidade de tempo paralelo $O(t)$, complexidade de custo $O(m)$ operações, e complexidade de processadores maior que $O(p)$. Então este algoritmo pode ser implementado com uma complexidade de tempo paralelo $O(m/p+t)$ e com complexidade de processadores $O(p)$.*

Demonstração: Suponha que no passo i, o algoritmo realize m_i operações. Logo temos que: $m = m_1 + m_2 + m_3 + \cdots + m_t$. Se utilizarmos apenas p processadores no passo i, o tempo total gasto neste passo será $\lceil m_i/p \rceil$ passos. Então, temos que o tempo total do algoritmo será:

$$\sum_{i=1}^{t} \left\lceil \frac{m_i}{p} \right\rceil \leq \sum_{i=1}^{t} \frac{m_i}{p} + 1 = \frac{m_1 + m_2 + \cdots + mt}{p} + t = \frac{m}{p} + t$$

Complexidade de tempo: $O(m/p + t)$. $\square$

Vamos exemplificar a utilização do Teorema de Brent para melhorar a complexidade do Algoritmo 2. Como vimos anteriormente, seu custo é $O(n \log n)$, pior do que a complexidade do algoritmo sequencial que é $O(n)$. Podemos observar que no laço do passo 2, na primeira iteração são executadas $n/2$ adições, $n/4$ na segunda iteração, $n/8$ na terceira, e assim sucessivamente, executando no total $n - 1$ adições ao final das $\log n$ iterações. Desde que ambos os algoritmos sequencial e paralelo executam $n-1$ adições, o custo ótimo do algoritmo paralelo da soma existe. Reduzindo o número de processadores para $p = n/\log n$, conseguimos obter a mesma complexidade do Algoritmo 2 e um custo ótimo.

Exemplo 3 Soma ótima Assumimos que nossa PRAM tem $p = 2^q \leq n = 2^k$ processadores $P_1, P_2, \ldots, P_p$ e seja $l = \frac{n}{p} = 2^{k-p}$. O vetor A de entrada é dividido em p subvetores tal que cada processador P_s é responsável pelo processamento do s-ésimo subvetor $A(l(s-1)+1), A(l(s-1)+2), \ldots, A(ls)$. Até a altura h da árvore binária, a geração dos $B(i)$'s é dividida de maneira análoga entre os p processadores. O número de operações concorrentes possíveis no nível h é $n/2^h = 2^{k-h}$. Se $2^{k-h} \geq p = 2^q$ (equivalentemente, $k-h-q \geq 0$, como no Algoritmo 3, dado a seguir), então essas operações são divididas igualmente entre os p processaores (passo 2.1 no Algoritmo 3). Caso contrário ($k-h-q < 0$), os 2^{k-h} processadores indexados com os menores índices executam essas operações (passo 2.2 do Algoritmo 3). O algoritmo executado pelo s-ésimo processador é dado a seguir.

Algoritmo 3 Algoritmo ótimo da Soma de n números para o Processador P_s
Entrada: Um vetor de $n = 2^k$ números, armazenados na memória compartilhada. As variáveis locais inicializadas são: (1) a ordem n; (2) o número p de processadores, onde $p = 2^q \leq n$, e (3) o número do processador s.
Saída: A Soma dos elementos de A armazenada na variável compartilhada S. O vetor A mantém seu valor original.

(1) **para** $1 \leq j \leq l(= n/p)$ **faça**
  (1.1) `B[l(s-1)+j]` $\leftarrow$ `A[l(s-1)+j]`

(2) **para** $h \leftarrow 1$ **até** $\log n$ **faça**
  (2.1) **se** $k - h - q \geq 0$ **então**

para $j = 2^{k-h-q}(s-1)+1$ **até** $j = 2^{k-h-q}s$ **faça**
`B[j]` $\leftarrow$ `B[2i-1] + B[2i]`
(2.2) **caso contrário se** $s \leq 2^{k-h}$ **então**
`B[s]` $\leftarrow$ `B[2s-1] + B[2s]`

(3) **se** $s = 1$ **então** `S` $\leftarrow$ `B[1]`

— Fim do Algoritmo —

O tempo de execução do algoritmo acima pode ser estimado como segue. O passo 1 utiliza $O(n/p)$ tempo, visto que cada processador executa n/p operações. A h-ésima iteração do passo 2 utiliza $O(\frac{n}{2^h p})$ tempo, visto que um processador tem que executar no máximo $\lceil \frac{n}{2^h p} \rceil$ operações. O passo 3 utiliza $O(1)$ tempo. Portanto, o tempo de execução total do algoritmo é dado por

$$T_p(n) = O\left(\frac{n}{p} + \sum_{h=1}^{\log n} \left\lceil \frac{n}{2^h p} \right\rceil\right) = O(\frac{n}{p} + \log n)$$

de acordo com o esperado pelo teorema de Brent.

Temos que durante as primeiras iterações do algoritmo, cada processador simula o trabalho que seria executado por um conjunto de processadores. Se o número de processadores for igual a $n/\log n$, podemos observar que o tempo gasto nessas iterações não altera o tempo de execução total do algoritmo, que é $O(\log n)$. Depois das primeiras iterações, quando não mais que $n/\log n$ processadores são necessários, o algoritmo é idêntico ao algoritmo PRAM original. O custo do algoritmo (produto da complexidade do tempo paralelo com o número de processadores usados), é $O(n)$. Portanto o algoritmo é ótimo. O algoritmo não efetua leituras nem escritas concorrentes, logo usa o modelo PRAM EREW. □

A quantidade dos dados transferidos entre a memória compartilhada e as memórias locais dos diferentes processadores representa a quantidade de comunicação necessária pelo algoritmo. No modelo PRAM esse aspecto não é levado em conta, uma vez que assume-se que o tempo de acesso a memória por qualquer processador é constante. Esse fato faz com que a complexidade de um algoritmo PRAM seja muito diferente dos resultados de obtidos em implementações desse algoritmo em sistemas de computação paralela existentes. Em função disso, apesar do modelo PRAM ser uma excelente ferramenta para projetar algoritmos paralelos, necessitamos de um modelo que capte os detalhes referentes a comunicação do algoritmo.

3.2. O Modelo de Rede

O **modelo de rede**, também chamado de **modelo de memória distribuída**, leva em consideração a comunicação através da incorporação da topologia da **rede de interconexão** dos processadores no próprio modelo. Uma **rede de interconexão** é um conjunto de canais de comunicação que ligam os processadores uns aos outros. Nesse modelo, um processador não possui acesso à uma memória local associada a outro processador (memória distribuída). Cada posição de cada memória local possui uma identificação (endereço) e pode ser acessada apenas por seu processador associado. A cada instante os processadores podem estar em dois estados: ativos ou inativos. Cada processador possui sua unidade de controle, que passa a instrução a ser executada para ele. Em cada instante, cada processador ativo executa uma instrução, possivelmente diferente da dos demais, sobre dados possivelmente diferentes. O modelo de rede possui arquitetura MIMD e pode operar no modo síncrono ou assíncrono.

Uma das formas dos processadores se comunicarem no modelo de rede, usando a rede de intreconexão, é através da **troca de mensagens**. Na abordagem de **troca de mensagens**, a divisão dos dados e das tarefas entre os processadores, além do gerenciamento das comunicações entre eles, é de responsabilbidade do programador. Se um processador i precisar, para efetuar alguma operação, de um dado que foi calculado pelo processador j, o processador j envia este dado para o processador i (através de uma mensagem), usando a rede de interconexão, e o processador i recebe este dado. Os tempos de transmissão das mensagens através da rede de interconexão são indeterminados (porém finitos). Note que, neste caso, um par de processadores não necessita obrigatoriamente ser adjacente; o processo de entregar cada mensagem da sua fonte ao seu destino é chamado de **roteamento**.

Para implementar os algoritmos no modelo de redes, precisamos de construções adicionais (**send** e **receive**) que permitam a comunicação entre os processadores.

1. `send(X,i)`
2. `receive(Y,j)`

Um processador P, ao executar a instrução **send**, envia uma cópia de X ao processador P_i, e passa a executar imediatamente a próxima instrução. Um processador P, ao executar uma instrução **receive**, suspende a execução de seu programa até que os dados do processador P_j sejam recebidos. Então, P armazena os dados em Y e continua a execução de seu programa. Essas comunicações são feitas da seguinte forma: o **send** realiza o envio de uma mensagem de um processador ao outro e é não-bloqueante, e o **receive** realiza o recebimento de uma mensagem e é bloqueante.

O modelo de rede pode ser visto como um grafo $G = (N, E)$, onde cada vértice $i \in N$ representa um processador, e cada aresta $(i, j) \in E$ representa um canal de comunicação bidirecional entre os processadores i e j. Existem diversos parâmetros usados para avaliar a topologia de uma rede. Seja o grafo $G = (N, E)$ representando o modelo de rede, definimos o **diâmetro** de G como sendo a distância máxima entre quaisquer par de vértices. A **conectividade** de G é representada pelo **grau máximo** de $i \in G$.

Vamos introduzir as seguintes topologias representativas de redes de interconexão: o **array linear**, o **mesh bidimensional** e o **hipercubo**. Várias outras topologias foram introduzidas e estudadas, não está dentro do escôpo deste trabalho a análise dessas topologias.

Array Linear e o Anel

O **array linear** consiste de p processadores $P_1, P_2, \ldots, P_p$ conectados em um array linear. O processador P_i está conectado ao P_{i-1} e ao P_{i+1}, sempre que eles existirem. O diâmetro de um array linear (máxima distância entre um par de vértice) é $p - 1$ e seu grau máximo é 2.

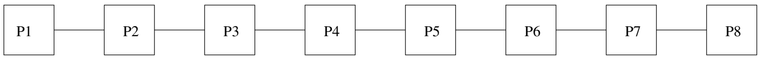


Figura 1.5: Um array linear com oito processadores

Um anel é um array linear de processadores onde os processadores P_1 e P_p estão diretamente conectados.

Exemplo 4 Soma de um vetor em um anel Seja A um vetor de ordem n, considere o problema de computar a soma $S = A(1) + \cdots + A(n)$ em um anel com p processadores, onde $p \leq n$. Seja $r = n/p$. A é particionado em p blocos, como segue: $A = (A_1, A_2, \ldots, A_p)$, onde cada A_i tem tamanho r. Podemos determinar a soma S calculando $s_i = A_i((i-1)r+1) + \cdots + A_i(ir)$, para $1 \leq i \leq p$, e então acumulando a soma $s_1 + s_2 + \cdots + s_p$.

Se o processador P_i de nossa rede tem inicialmente $B = A_i$ em sua memória local, para todo $1 \leq i \leq p$. Então, cada processador pode computar localmente a soma $B(1) + \cdots + B(r)$ e podemos acumular a soma desses valores percorrendo circularmente o anel. O valor da soma será armazenada em P_1. O algoritmo a ser executado por cada processador é dado abaixo:

Algoritmo 4 Soma de um vetor em um anel
Entrada: (1) O número do processador i; (2) O número p de processadores; (3) O i-ésimo sub-vetor $B = A((i-1)r + 1 : ir)$ de tamanho r, onde $r = n/p$.
Saída: Processador P_i calcula o valor $S = s_1 + \cdots + s_i$ e passa o resultado para a direita. Quando o algoritmo termina, P_1 terá a soma S.

(1) `z` $\leftarrow$ `B[1]` $+\cdots+$ `B[r];`

(2) **se** $i = 1$ **então** `S` $\leftarrow$ `0;`
    caso contrário `receive(S,esquerda);`

(3) `S` $\leftarrow$ `S+z;`

(4) `send(S,direita);`

(5) **se** $i = 1$ **então** `receive(S,esquerda);`

— Fim do Algoritmo —

Cada processador P_i começa calculando $A_i((i-1)r+1) + \cdots + A_i(ir)$; e armazena o vetor resultante na variável local z. No passo 2, o processador P_1 inicializa o valor de S com 0, enquanto cada um dos outros processadores suspende a execução do programa esperando receber o dado do seu vizinho esquerdo. O processador P_1 calcula $S = A_1(1) + \cdots + A_1(r)$ e envia o resultado ao seu vizinho da direita nos passos 3 e 4 respectivamente. Nesse ponto, P_2 recebe $A_1(1) + \cdots + A_1(r)$ e retoma a execução de seu programa calculando $A_1(1) + \cdots + A_1(r) + A_2(r+1) + \cdots + A_2(2r)$ no passo 3; o processador, então, envia o novo valor para a direita no passo 4. Quando a execução dos programas em todos os processadores termina, P_1 possui a soma S.

A computação executada por cada processador consiste de r operações no passo 1. Nos passos 2 e 3, os processadores P_i executam uma operação. Portanto, o tempo de execução do algoritmo é $O(n/p)$, digamos que $T_{comp} = \alpha(n/p)$, onde α é uma constante. Por outro lado, o processador P_1 tem de esperar até que a soma parcial $s_1 + \cdots + s_p$ seja calculada antes que ele possa computar o passo 5. Logo, o tempo total de comunicação T_{comm} é proporcional ao produto $p * comm(n) = p(\sigma + n\tau)$, onde σ é o tempo da transmissão, e τ é a taxa na qual a mensagem pode ser transmitida. O tempo total de execuçãoo é dado por $T = T_{comp} + T_{comm} = \alpha(n/p) + p(\sigma + n\tau)$. □

O **mesh** é uma versão bidimensional do vetor linear, que consiste de $p = m^2$ processadores arranjados em uma matriz $m \times m$, onde o processador $P_{i,j}$ está conectado aos processadores $P_{i\pm1,j}$ e $P_{i,j\pm1}$, sempre que eles existirem. O diâmetro de um mesh com $p = m^2$ processadores é $\sqrt{p}$, e o grau máximo de qualquer nó é 4. Esta topologia tem diversas características atraentes, tais como simplicidade, regularidade e escalabilidade. Além disso, ela traduz as estruturas de computação que aparecem em muitas aplicações. Contudo, em função do diâmetro do *mesh*, quase toda computação não trivial requer $\Omega(\sqrt{p})$ passos.

Hipercubo

O **hipercubo** consiste de $p = 2^d$ processadores interconectados em um cubo booleano de dimensão d que pode ser definido como segue. Seja $i_{d-1}i_{d-2}\ldots i_0$ a representação binária de i , onde $0 \leq i \leq p-1$. O processador P_i é conectado aos processadores $P_{i^{(j)}}$, onde $i^{(j)} = i_{d-1}\ldots\bar{i}_j\ldots i_0$, e $\bar{i}_j = 1 - i_j$, para $0 \leq j \leq d-1$. Em outras palavras, dois processadores são conectados se, e somente se, as representações binárias (*bits*) de seus índices diferem apenas em uma posição. Note que nossos processadores estão indexados de 0 até $p-1$.

O hipercubo tem uma estrutura recursiva. Podemos ampliar um cubo de d dimensões para um de $d+1$ dimensões conectando os processadores correspondentes dos dois cubos de d dimensões. Os endereços dos processadores de um dos cubos tem o *bit* mais significante igual a 0; o outro cubo tem os endereços dos processadores com o *bit* mais significativo igual a 1. A Figura 1.6 ilustra um hipercubo com 4 dimensões.

O diâmetro de um hipercubo d-dimensional é igual a $\log p$, visto que a distância entre os processadores P_i e P_j é igual ao número de posições (*bits*) na qual i e j diferem; portando, é menor ou igual a d, e a distância entre P_0 e P_{2^d-1} é d. Cada vértice possui grau $d = \log p$.

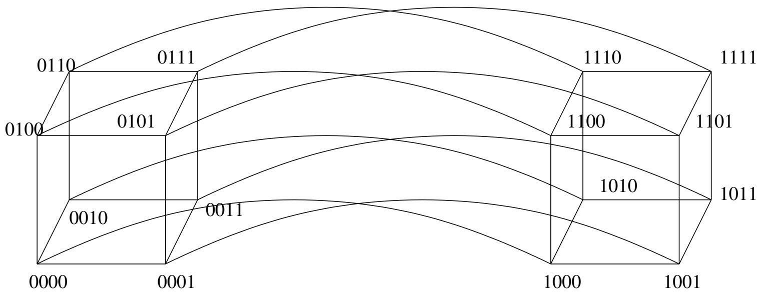


Figura 1.6: Um hipercubo tetra-dimensional

Exemplo 5 (**Soma no hipercubo**) Cada entrada $A(i)$ de um vetor A de tamanho n é inicialmente armazenado na memória local do processador P_i de um hipercubo síncrono com $n = 2^d$ processadores. O objetivo é calcular a soma $S = \sum_{i=0}^{n-1} A(i)$, e armazenar

S no processador P_0. O algoritmo para calcular S consiste de d iterações. A primeira iteração calcula somas de pares de elementos entre processadores cujos índices diferem no bit mais significativo. Essas somas são armazenadas no subcubo $(d-1)$-dimensional cujo *bit* mais significativo do endereço é igual a 0. As iterações restantes continuam de modo análogo.

No algoritmo abaixo, o hipercubo opera sincronizadamente, e $i^{(l)}$ denota o índice i cujo *bit* l foi complementado. A instrução $A(i) \leftarrow A(i) + A(i^{(l)})$ é composta de dois subpassos. No primeiro subpasso, P_i copia $A(i_{(l)})$ do processador $P_{i^{(l)}}$ através do canal conectando $P_{i^{(l)}}$ e P_i; no segundo subpasso, P_i executa a soma $A(i) + A(i^{(l)})$, armazenando o resultado em A(i).

Algoritmo 5 (**Soma no Hipercubo - Algoritmo para o processador** P_i)
Entrada: Um vetor A de $n = 2^d$ elementos tal que $A(i)$ está armazenado na memória local do processador P_i de um hipercubo síncrono com n processadores; onde, $0 \leq i \leq n$.
Saída: A soma $S = \sum_{i=0}^{n-1} A(i)$ armazenada em P_0.

(1) **para** $l \leftarrow d-1$ **até** 0 **faça**
  se $0 \leq i \leq 2^l - 1$ **então**
    `A[i]` $\leftarrow$ `A[i] + A[i`$^{(l)}$`]`

— Fim do Algoritmo —

Vamos analisar o algoritmo para $n = 8$. Então, durante a primeira iteração do laço **para**, as somas $A(0) = A(0) + A(4)$, $A(1) = A(1) + A(5)$, $A(2) = A(2) + A(6)$ e $A(3) = A(3) + A(7)$ são calculadas e armazenadas nos processadores P_0, P_1, P_2 e P_3, respectivamente. No final da segunda iteração, obtemos $A(0) = (A(0) + A(4)) + (A(2) + A(6))$ e $A(1) = (A(1) + A(5)) + (A(3) + A(7))$. A terceira iteração claramente atrubui a soma S a $A(0)$. O algoritmo termina depois de $d = \log n$ passos paralelos. □

Apesar do modelo de rede incorporar a comunicação no projeto e análise de seus algoritmos, a escalabilidade desses algoritmos é bastante reduzida. Por outro lado, a portabilidade é bastante pequena, uma vez que os algorimos incorporam a topologia da aquitetura de rede de interconexão em seus projetos. Um outro ponto é o fato de que os resultados das implementações desses algoritmos em máquinas existentes nem sempre obtêm bons speedups. Isso faz com analisemos outros modelos, onde o resultado previsto pelo modelo teórico seja próximo dos obtidos nas implementações.

3.3. Modelos Realísticos

Durante o final dos anos 80, o campo de algoritmos paralelos atravessava uma série crise. Vários problemas haviam sido estudados, e vários limites inferiores e superiores foram demonstrados para esses problemas em diferentes modelos de computação, tais como memória compartilhada, hipercubos e *meshs*. Quando esses resultados teóricos eram implementados nas máquinas existentes (usando essas topologias de interconexão), os *speedups* obtidos eram muitas vezes desapontadores. Os anos 90 troxeram uma mudança significativa para a área com o surgimento de modelos de computação paralela com "granularidade grossa" (Valiant [25]). Nesse modelo, o conceito de computação paralela é modelado com uma série de **superpassos** ao invés de passos com o envio individual de mensagens ou acessos individuais a memória compartilhada.

Vamos apresentar dois modelos que traduzem os conceitos propostos por Valiant [25], o modelo **BSP** (**Bulk Synchronous Parallel Model**) e modelo **CGM**

(**Coarse Grained Multicomputers**). Os algoritmos projetados para esses modelos, quando implementados nas máquinas existentes, têm obtido *speedups* próximos dos previstos em resultados teóricos. Esse fato faz com sejam denominados modelos realísticos.

O Modelo BSP

O modelo **BSP** foi proposto por Valiant [25], em 1990. Além de ser um dos modelos realísticos mais importantes, foi um dos primeiros a considerar os custos de comunicação e a abstrair as características de uma máquina paralela em um pequeno número de parâmetros. O objetivo principal deste modelo é servir de modelo ponte entre as necessidades de *hardware* e *software* na computação paralela, essa ponte é uma das características fundamentais do sucesso do modelo sequencial de Von Neumann.

O modelo **BSP** consiste de um conjunto de p processadores com memória local, comunicando-se através de algum meio de interconexão, gerenciados por um **roteador** e com facilidades de sincronização global. Um algoritmo BSP consiste numa sequência de **superpassos** separados por **barreiras de sincronização**, como mostra a Figura 1.7. Em um **superpasso**, a cada processador é atribuído um conjunto de operações independentes, consistindo de uma combinação de passos de computação, usando dados disponibilizados localmente no início do superpasso, e passos de comunicação, através de instruções de envio e recebimento de mensagens. Neste modelo uma ***h*-relação** em um superpasso corresponde ao envio e/ou recebimento de, no máximo, *h* mensagens em cada processador. A resposta a uma mensagem enviada em um superpasso somente será usada no próximo superpasso.

Os parâmetros do modelo BSP são os seguintes:

- n: tamanho do problema;
- p: número de processadores disponíveis, cada qual com sua memória local;
- L: o tempo mínimo entre dois passos de sincronização. Também chamado de parâmetro de periodicidade ou **latência** de um superpasso;
- g: é a capacidade computacional dividida pela capacidade de comunicação de todo o sistema, ou seja, a razão entre o número de operações de computação realizadas em uma unidade de tempo e o número de operações de envio e recebimento de mensagens. Este parâmetro descreve a taxa de eficiência de computação e comunicação do sistema.

Os dois últimos parâmetros, L e g, são utilizados para computar o **custo de comunicação** de um algoritmo BSP. O parâmetro L representa o **custo de sincronização**, de tal forma que cada operação de sincronização contribui com L unidades de tempo para o tempo total de execução. Podemos também ver L, como sendo a **latência** da comunicação, pois os dados recebidos somente podem ser acessados no próximo superpasso. A capacidade de comunicação de uma rede de computadores está relacionada ao parâmetro g. Através deste parâmetro podemos estimar o tempo gasto pela troca de dados entre os processadores. Se o número máximo de mensagens enviadas por algum processador durante uma troca simples é h, então seriam necessárias até gh unidades de tempo para a conclusão da troca. Na prática, o valor de g é determinado empiricamente, para cada máquina paralela, através da execução de *benchmarks* apropriados [17].

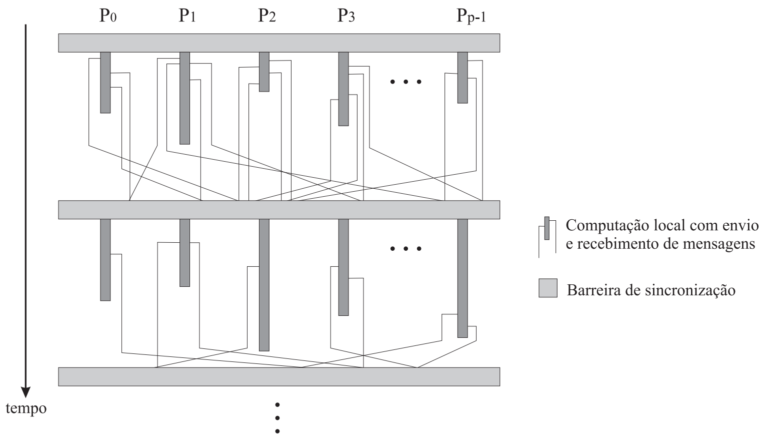


Figura 1.7: O Modelo BSP

Logo, o **tempo total de execução** de um superpasso de um algoritmo BSP é igual a $w_i + gh_i + L$, onde $w_i = \max\{L; t_1; \ldots; t_p\}$ e $H_i = \max\{L; c_1; \ldots; c_p\}$, t_j e c_j são respectivamente, o número de operações de computações executadas e o número de mensagens recebidas e/ou enviadas pelo processador j no superpasso i. O **custo total** de um algoritmo é dado pela soma dos custos de cada um dos superpassos. Considerando T o número de superpassos, seja

$$W = \sum_{i=0}^{T} w_i \;\text{ e }\; H = \sum_{i=0}^{T} h_i$$

a soma de todos os valores de w e h de cada superpasso. Então, o **custo total** de um algoritmo BSP é $W + gH + LT$. O valor de W representa o total de computações locais e o valor de H representa o volume total de comunicação.

O Modelo CGM

O modelo **CGM** foi proposto por Dehne *et al* [7]. Nesse modelo, os processadores podem estar conectados por qualquer meio de interconexão. O termo "granularidade grossa" (*coarse grained*) vem do fato de que o tamanho do problema é consideravelmente maior que o número de processadores, ou seja, $n/p \gg p$.

Um **algoritmo CGM** consiste de uma sequência de **rodadas** (*rounds*), alternando fases bem definidas de computação local e comunicação global, como mostra a Figura 1.8. Normalmente, durante uma rodada de computação é utilizado o melhor algoritmo sequencial para o processamento dos dados disponibilizados localmente.

O CGM é semelhante ao modelo BSP, no entanto é definido em apenas dois parâmetros:

1. n: tamanho do problema;
2. p: número de processadores disponíveis, cada um com uma memória local de tamanho $O(n/p)$.

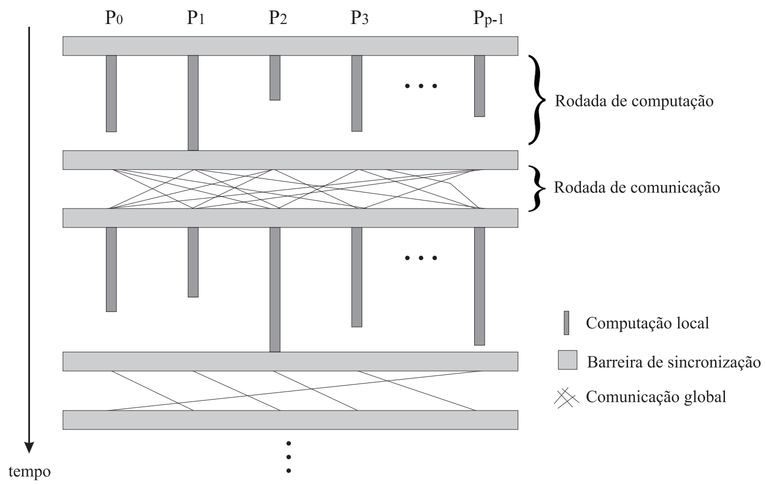


Figura 1.8: O Modelo CGM

Em uma rodada de comunicação uma h-relação (com $h = O(n/p)$) é roteada, isto é, cada processador envia $O(n/p)$ dados e recebe $O(n/p)$ dados. No modelo CGM, o custo de comunicação é modelado pelo número total de rodadas de comunicação.

O custo de um algoritmo CGM é a soma dos tempos obtidos em termos do número total de rodadas de computação local (análogo ao W do modelo BSP) e o número de superpassos (análogo ao T do modelo BSP), que equivale ao número total de rodadas de comunicação.

Um algoritmo CGM é um caso especial de um algoritmo BSP onde todas as operações de comunicação de um superpasso são feitas na h-relação. Conforme observado por Dehne [8], os algoritmos CGM, quando implementados em multiprocessadores atualmente disponíveis, se comportam bem e exibem *speedups* similares àqueles previstos em suas análises. Para estes a lgoritmos, o maior objetivo é minimizar o número de superpassos e a quantidade de computação local.

Algoritmo da Soma no Modelo BSP/CGM

Vamos agora apresentar um algoritmo BSP/CGM para o problema da soma dos elementos de um vetor.

Exemplo 6 Soma de um vetor no modelo BSP/CGM Seja A um vetor de ordem n, considere o problema de computar a soma $S = A(1) + \cdots + A(n)$ no modelo BSP/CGM com p processadores, onde $p \ll n$. Seja $r = n/p$. A é particionado em p blocos, como segue: $A = (A_1, A_2, \ldots, A_p)$, onde cada A_i tem tamanho r. Para determinar a soma S, cada processador P_i computa a i-ésima soma parcial $s_i = A_i((i-1)r+1) + \cdots + A_i(ir)$, para $1 \leq i \leq p$, e envia s_i, através de uma mensagem, para o processador P_1, que computa o total das somas parciais.

Algoritmo 6 Soma de um vetor no BSP/CGM
Entrada: (1) O número do processador i; (2) O número p de processadores; (3) O i-ésimo sub-vetor $B = A((i-1)r + 1 : ir)$ de tamanho r, onde $r = n/p$.
Saída: Processador P_i calcula o valor $S = s_1 + \cdots + s_i$ e envia o resultado para P_1. Quando o algoritmo termina, P_1 terá a soma S.

(1) `z` $\leftarrow$ `B[1] +···+ B[r];`

(2) **se** $i = 1$ **então** `S` $\leftarrow$ `z;`
caso contrário `send(z,P`$_1$`);`

(3) **se** $i = 1$ **então**
para $i = 2$ **até** p **faça**
`receive(z,P`$_i$`);`
`S` $\leftarrow$ `S+z;`

— Fim do Algoritmo —

Cada processador P_i começa calculando $A_i((i-1)r+1)+\cdots+A_i(ir)$ e armazena o vetor resultante na variável local z. No passo 2, o processador P_1 inicializa o valor de S com z, enquanto cada um dos outros processadores enviam o valor da soma parcial z para P_1 e finalizam a execução do programa. No passo 3, o processador P_1 recebe $p - 1$ somas parciais e computa a soma total S.

A computação executada por cada processador P_i, $1 \leq i \leq p$ consiste de r operações no passo 1. No passo 3, o processador P_1 executa $p - 1$ operações. Portanto, o tempo de computação do algoritmo é $O(n/p)$. Por outro lado, o processador P_1 recebe $p - 1$ mensagens, todas no mesmo superpasso (BSP) ou rodada (CGM), logo o algoritmo utiliza $O(1)$ rodadas de comunicação. $\square$

3.4. Comparação

Embora, para uma dada situação, cada um dos modelos paralelos mostrados podem ser claramente vantajosos, os modelos realísticos tem se mostrado bastante apropriados para o projeto, análise e implementação de algoritmos paralelos.

O modelo de memória compartilhada é apropriado para o projeto e análise de algoritmos paralelos, pois não temos que tratar de todos os detalhes referentes a comunicação. Muitos algoritmos foram projetados para o modelo PRAM nos anos 80 e início dos anos 90's. Poucas máquinas disponíveis no mercado implementam o conceito de memória compartilhada. Ainda assim, as máquinas existentes possuem apenas algumas dezenas de processadores, pois existem limitações tecnológicas para possibilitar a construção de máquinas com um número expressivo de processadores e que utilizem memória compartilhada.

Embora o modelo de rede pareça ser consideravelmente melhor adaptado para resolver problemas computacionais e de comunicação do que o modelo PRAM, sua comparação com o modelo de memória compartilhada é mais sutil. Apesar de que no modelo de rede ser significativamente mais difícil descrever e analisar algoritmos, e depender consideravelmente da topologia particular sob consideração, topologias diferentes podem necessitar de algoritmos completamente diferentes para resolver o mesmo problema.

Os algoritmos projetados para o modelo de rede não possuem a portabilidade e a escalabilidade desejada, pois são projetados para máquinas específicas. A escalabilidade, no caso do hipercubo por exemplo, é crítica, pois só é feita através da duplicação dos processadores.

No caso dos modelos realísticos, ambos os modelos, BSP e CGM, tentam reduzir os custos de comunição de modo muito semelhante e buscam caracterizar uma máquina paralela através de um conjunto de parâmetros, sendo que dois destes se assemelham, o número de operações de computação local e o número de superpassos (rodadas no CGM).

No entanto, existem algumas diferenças. Uma delas, segundo Götz [13], é que o modelo CGM simplifica o projeto e o desenvolvimento de algoritmos por ser um modelo mais simples e um pouco mais poderoso que o modelo BSP.

Além disso, um algoritmo CGM pode ser transferido para o modelo BSP sem mudanças. Se um algoritmo CGM executa W operações de computação local, T_{cp} superpassos de computação e T_{cm} superpassos de comunicação, então o algoritmo BSP correspondente terá um custo de $O(W + ghT_{cm} + L(T_{cp} + T_{cm}))$, detalhes podem ser vistos em Cáceres *et al* [3].

Em resumo, os modelos BSP e CGM apresentam muitas similaridades. Entretanto o modelo CGM simplifica os custos de comunicação, facilitando o projeto de algoritmos.

4. Implementação de Algoritmos Paralelos

Uma vez definido o modelo para o projeto e análise de algoritmos, vamos agora descrever alguns aspectos referentes a implementação dos algoritmos projetados para o modelo BSP/CGM.

Na implementação de algoritmos em sistemas de computação paralela utilizando o paradigma de troca de mensagens, temos que levar em consideração um grande número de fatores, dentre os quais destacamos os seguintes:

- arquitetura;
- formato dos dados;
- velocidade computacional;
- carga da máquina, e
- carga da rede.

Com o objetivo de tornar transparente ao usuário todos esses detalhes, foram desenvolvidas várias bibliotecas que implementam o paradigma da troca de mensagens. Vamos utilizar duas que ao longo dos anos 90 praticamente tornaram-se padrões nessa área, o **Parallel Virtual Machine** (**PVM**) e o **Message Passing Interface** (**MPI**).

Neste capítulo descrevemos alguns aspectos dessas bibliotecas através da implementação do Algoritmo da Soma BSP/CGM utilizando o PVM e o MPI. Nos Capítulos 2 e 3 introduziremos outras características dessas bibliotecas. Uma descrição mais detalhada de algumas das funcionalidades dessas bibliotecas será apresentada no Apêndice.

4.1. PVM

O *software* **PVM (Parallel Virtual Machine)** providencia uma estrutura unificada dentro da qual programas paralelos podem ser desenvolvidos de uma maneira direta e eficiente usando equipamentos de *hardware* existentes. O PVM possibilita que uma coleção heterogêna de estações de trabalho possam ser vistas como uma única máquina paralela virtual. O PVM trata de forma transparente todo o roteamento das mensagens, a conversão de dados e o escalonamento das tarefas numa rede de computadores com arquiteturas possivelmente incompatíveis.

O modelo computacional do PVM é simples, além de bastante genérico, e acomoda uma grande variedade de estruturas de programas de aplicação. A interface de programação é direta, permitindo que estruturas de programas simples possam ser implementadas de maneira intuitiva. O usuário escreve sua aplicação como uma coleção de tarefas cooperativas. As tarefas acessam os recursos do PVM através de uma biblioteca de rotinas de interface padrão. Essas rotinas permitem a inicialização e o término das tarefas na rede além da comunicação e sincronização entre as tarefas. As primitivas de troca de mensagens do PVM são orientadas para operações heterogêneas, envolvendo construções de tipos de dados para *buffering* e transmissão. Construções para a comunicação incluem as de envio e recebimento de estruturas de dados além de primitivas de alto-nível tais como *broadcast*, barreiras de sincronização e soma global.

As tarefas do PVM podem ter estruturas de dependência e controle arbitrárias. Em outras palavras, em qualquer ponto da execução de uma aplicação concorrente, qualquer tarefa existente pode iniciar ou finalizar outras tarefas, ou adicionar ou remover computadores da máquina virtual. Qualquer processo pode se comunicar-se e/ou sincronizar com qualquer outro. Qualquer controle específico e estrutura de dependência pode ser implementado em um sistema PVM através do uso apropriado de contruções do PVM e instruções de controle de fluxo da linguagem utilizada.

Devido a sua natureza onipresente (especificamente, o conceito de máquina virtual) e também pela sua simples mas completa interface de programação, o sistema PVM tem tido uma grande aceitação na comunidade de computação científica de alto desempenho.

Os programas `somappvm.c++` e `somafpvm.c++` descrevem uma implementação do Algoritmo da Soma BSP/CGM no modelo mestre escravo.

```
// Programa:  somappvm.c++ -- versão simplificada
// Programador:  jai2001
// Data:  21/05/2001
// O Diálogo:  Este programa inicializa um vetor de TAMMAX
// elementos e um conjunto de p tarefas (filhos).  Envia
// para cada filho um subvetor de tamanho TAMMAX/p.  Cada
// filho efetua a soma de seu subvetor e envia a resposta
// para o pai.  O pai recebe as somas parciais dos filhos e
// efetua a soma total.
// Declaração das bibliotecas utilizadas
#include<pvm3.h>
#include<iostream.h> // cout, endl
#include<stdlib.h> // exit
#include<iomanip.h> // hex
// Declaração das constantes globais
const unsigned int MSGTAG = 11; // valor arb.  para a TAG
const unsigned int TAMSTR = 30; // tamanho máximo da string
const unsigned int NUMHOSTS = 4; // hosts maq.  virtual
const unsigned int TAMMAX = 16; // tamanho do vetor

// Declaração dos tipos
```

```
typedef int Vetor[TAMMAX];
typedef char string[TAMSTR];
typedef string VetorS[NUMHOSTS];

// início da função principal
int main(void) {
// declaracao das variáveis locais
     int Soma = 0, SomaP, mtid, tam;
     Vetor Tid, SubVetor;
     VetorS Hosts;
     int i, j;
     int VetorDados[16]=
               {3,-2,5,2,3,7,1,0,4,-3,1,9,6,8,-1,2};

// Passo 1.  Inicilização
     strcpy(Hosts[0],"knuth.localdomain"); // maq.  virtual
     strcpy(Hosts[1],"knuth.localdomain"); // maq.  virtual
     strcpy(Hosts[2],"knuth.localdomain"); // maq.  virtual
     strcpy(Hosts[3],"knuth.localdomain"); // maq.  virtual
     tam = (int) TAMMAX/NUMHOSTS; // tamanho subvetor
// Passo 2.  Determine a id da tarefa
     mtid = pvm_mytid();
// Passo 3.  Inicialize as tarefa a serem executadas
     for (i = 0; i < NUMHOSTS; i++)
          pvm_spawn("somafpvm", (char**)0, 1, Hosts[i], 1,
&Tid[i]);
// Passo 4.  Envie os dados as tarefas filhos
     for (i = 0; i < NUMHOSTS; i++) {
          for (j = 0; j < tam; j++) // Compute o vetor
               SubVetor[j]=VetorDados[tam*i + j];
          pvm_initsend(PvmDataDefault); // buffer
          pvm_pkint(&tam, 1, 1); // tamanho do subvetor
          pvm_pkint(SubVetor, tam, 1); // subvetor
          pvm_send(Tid[i], MSGTAG); // envie para o filho i
     } // fim for
// Passo 5.  Receba as Somas Parciais
     for (i = 0; i < NUMHOSTS; i++) {
          pvm_recv(-1, MSGTAG); // receba de algum filho i
          pvm_upkint(&SomaP, 1, 1); // valor recebido
          Soma = Soma + SomaP; // i-ésima soma parcial
     } // fim for
// Passo 6.  Imprima o valor da Soma
     cout << endl << "Soma:  ≪< Soma << endl;
// Passo 7.  Finalize o PVM
     pvm_exit();
     exit(0);
} // fim função main
```

```
// Programa:  somafpvm.c++
// Programador:  jai2001
// Data:  21/05/2001
// O Diálogo:  Este programa recebe um subvetor da tarefa
// pai e efetua a soma dos elementos do subvetor e envia
// o resultado da soma parcial a tarefa pai.
// Declaração das bibliotecas utilizadas
#include<pvm3.h> // pvm_parent(), pvm_recv, pvm_upkint,
               //pvm_send, pvm_exit
#include<stdlib.h> // exit
// Declaração das constantes
const unsigned int MSGTAG = 11;
const unsigned int TAMMAX = 16;
// Declaração de tipos
typedef int Vetor[TAMMAX];

// início da funcao principal
int main(void) {
// declaração das variáveis locais
     int mtid, ptid, info, tam, Soma=0;
     Vetor SubVetor;
     unsigned int i;

// Passo 1.  Determine os ids das tarefas
     mtid = pvm_mytid(); // tarefa
     ptid = pvm_parent(); // tarefa pai
// Passo 2.  Recebe uma MSG da tarefa pai
     pvm_recv(ptid, MSGTAG);
// Passo 3.  Desempacote a MSG
     pvm_upkint(&tam, 1, 1); // tamanho do subvetor
     pvm_upkint(SubVetor, tam, 1); // subvetor
// Passo 4.  Calcule a Soma
     for (i = 0; i < tam; i++)
          Soma = Soma + SubVetor[i];
// Passo 5.  Envie a Soma a tarefa pai
     pvm_initsend(PvmDataDefault); // inicialize buffer
     pvm_pkint(&Soma, 1, 1); // empacote o valor da soma
     pvm_send(ptid, MSGTAG); // envie o conteudo do buffer
// Passo 6.  Finalize o PVM
     pvm_exit();
     exit(0);
} // fim da função main
```

4.2. MPI

O padrão **MPI (Message Passing Interface)**, cuja especificação foi completada em Abril de 1994, é o resultado de um esforço da comunidade de tentar definir a sintaxe e a semântica da parte essencial de uma biblioteca de rotinas de troca de mensagens que seria útil a uma grande gama de usuários e que pudesse ser implementada numa grande variedade de máquinas paralelas. A principal vantagem de estabecer um padrão de troca de mensagens é a portabilbidade. Um dos objetivos do desenvolvimento do MPI é o de fornecer aos contrutores de máquinas paralelas com um conjunto base claramente definido de rotinas que eles podem implementar eficientemente ou, em alguns casos, prover o hardware necessário para essas rotinas, portanto aumentando a escalabilidade.

O MPI não tem o propósito de ser uma infraestrutura de software completo e auto-contido que possa ser utilizado em computação distribuída. O MPI-1 não inclui funções necessárias ao gerenciamento de processos (a capacidade de iniciar tarefas), configuração da máquina virtual, e suporte de entrada e saída.

Abaixo são listados algumas razões que fazem com que o MPI venha sendo cada vez mais utilizado.

1. O MPI tem mais que uma implementação de boa qualidade disponível gratuitamente.
2. O MPI tem total comunicação assíncrona.
3. Os grupos MPI são sólidos, eficientes e determinísticos.
4. O MPI gerencia eficientemente os *buffers* de mensagens.
5. O MPI pode ser utilizado eficientemente para programar computadores parelelos e *clusters* de estações de trabalho.
6. O MPI é totalmente portável.
7. O MPI é formamente especificado.
8. O MPI é um padrão.

O programa `somampi.c` descreve uma implementação do Algoritmo da Soma BSP/CGM no modelo SPMD.

```
// Programa:  somampi.c -- versão simplificada
// Programador:  jai2001
// Data:  21/05/2001
// O Diálogo:  Este programa inicializa um vetor de TAMMAX
// elementos e um conjunto de p tarefas (filhos).  Envia
// para cada filho um subvetor de tamanho TAMMAX/p.Cada
// filho efetua a soma de seu subvetor e envia a resposta
// para o pai.  O pai recebe as somas parciais dos filhos e
// efetua a soma total.
// Declaração das bibliotecas utilizadas
#include<mpi.h>
#include<stdio.h> // printf
// Declaração das constantes globais
const unsigned int MSGTAG = 11; // valor arb.  para a TAG
const unsigned int TAMMAX = 16; // tamanho do vetor
```

```
// Declaração dos tipos
//typedef int Vetor[TAMMAX];

// início da função principal
int main(int argc, char *argv[]) {
// declaração das variáveis locais
     int Soma = 0, SomaP = 0, rank, size, tam;
     int SubVetor[10];
     int i, j;
     int VetorDados[16]=
               {3,-2,5,2,3,7,1,0,4,-3,1,9,6,8,-1,2};
     MPI_Status status;

// Passo 1.  Inicilização
     MPI_Init(&argc, &argv);
     MPI_Comm_size(MPI_COMM_WORLD, &size); // n.  tarefas
     MPI_Comm_rank(MPI_COMM_WORLD, &rank); // id.  da tarefa
     tam = (int) TAMMAX/size; // tamanho subvetor
// Passo 4.  Envie os dados as tarefas filhos
     if (rank == 0) {
          for (i = 1; i < size; i++) {
               for (j = 0; j < tam; j++) // subvetor
                    SubVetor[j]=VetorDados[tam*i + j];
               MPI_Send(SubVetor, tam, MPI_INT, i, MSGTAG,
MPI_COMM_WORLD);
          } // fim for
          for (j = 0; j < tam; j++) // subvetor P. 0
               SubVetor[j]=VetorDados[j];
     } else {
// Passo 5.  Recebe uma MSG da tarefa 0
          MPI_Recv(SubVetor, tam, MPI_INT, 0, MSGTAG,
MPI_COMM_WORLD, &status);
     } // fim if
// Passo 4.  Calcule a Soma
     for (i = 0; i < tam; i++)
          SomaP = SomaP + SubVetor[i];
// Passo 5.  Receba as Somas Parciais
     MPI_Reduce(&SomaP, &Soma, 1, MPI_INT, MPI_SUM, 0,
MPI_COMM_WORLD);
// Passo 6.  Imprima o valor da Soma (Processador 0)
     if (rank == 0)
          printf("Soma:  // Passo 7.  Finalize o MPI
     MPI_Finalize();
     return 0;
} // fim funcao main
```

Capítulo 2

Técnicas Básicas

Objetivos

- Descrever algumas das principais técnicas para o desenvolvimento de algoritmos paralelos.
- Descrever um algoritmo paralelo para ordenação.
- Introduzir problemas básicos em processamento de listas.
- Resolver de forma ótima o problema do *list ranking* paralelo.
- Analisar a eficiência dos algoritmos apresentados.

1. Soma de Prefixos

Considere a sequencia de n elementos $\{x_1, x_2, \ldots, x_n\}$ pertencentes a um conjunto S com uma operação binária associativa, denotada por $*$. As **somas de prefixos** desta sequencia são as n somas parciais (ou produtos) definidos por

$$s_i = x_1 * x_2 * \ldots * x_i, 1 \leq i \leq n$$

.

Esse problema é resolvido no modelo sequencial por um algoritmo bastante simples. Basta utilizarmos um vetor s, onde $s[0] \leftarrow 0$ e $s[i] \leftarrow s[i-1] + x_i$. Após n iterações, obtemos o resultado. Esse algoritmo é executado em tempo $O(n)$ e é inerentemente sequencial.

A solução do problema de Somas de Prefixos no modelo BSP/CGM é semelhante ao da Soma de n números. A idéia é o de dividir a entrada em p (número de processadores) subconjuntos, cada um com n/p elementos e distribuir esses subconjuntos entre os p processadores (um subconjunto para cada processador). Mais precisamente, seja A um vetor de ordem n, desejamos computar as Somas de Prefixos $S[i] = A[1] + \cdots + A[i], 1 \leq i \leq n$ com p processadores, onde $p \ll n/p$. Seja

$r = n/p$. A é particionado em p blocos, como segue: $A = (A_1, A_2, \ldots, A_p)$, onde cada A_i tem tamanho r. Para determinar as Somas Parciais $S[i]$, cada processador P_i computa a i-ésima soma $s_i = A_i[(i-1)r+1] + \cdots + A_i[ir]$, para $1 \leq i \leq p$, e efetua um *broadcast* de s_i, através de uma mensagem, para todos os processadores P_i. Cada processador P_i computa as suas respectivas somas parciais $S[(i-1)*r+1], \ldots, S[i*r]$, onde $S[(i-1)*r+j] \leftarrow \sum_{l=1}^{i-1} s_l + \sum_{j=1}^{r} A[(i-1)*r+j]$.

Algoritmo 7 Somas de Prefixos de um vetor no modelo BSP/CGM
Entrada: (1) O número do processador i; (2) O número p de processadores; (3) O i-ésimo sub-vetor $B = A((i-1)r+1 : ir)$ de tamanho r, onde $r = n/p$.
Saída: Em cada Processador P_i as somas de prefixo $S[(i-1)*r+j]$, $1 \leq j \leq n/p$.

(1) $s_i \leftarrow$ `B[1] +` $\cdots$ `+ B[r];`

(2) `broadcast(`$s_i, p_j \neq i$`);`

(3) `S[(i-1)*r]` $\leftarrow s_1 + \cdots s_{i-1}$

(4) **para** $k = 1$ **até** r **faça**
```
    S[(i-1)*r+k] ← S[(i-1)*r+k-1] + B[k]
```

— Fim do Algoritmo —

Teorema 2 *O Algoritmo 7 (Somas de Prefixo) computa as somas de prefixo de n elementos em tempo $O(n/p)$, com $O(1)$ rodadas de comunicação.*

Demonstração: Cada processador P_i começa calculando $A_i[(i-1)r+1] + \cdots + A_i[ir]$; e armazena o valor resultante na variável local s_i. Isso é feito sequencialmente sem a necessidade de nenhuma comunicação. No passo 2, os processadores fazem um *broadcast* de s_i para os demais processadores. Essa comunicação pode ser feita em uma única rodada de comunicação. No passo 3, cada processador calcula, sequencialmente, o valor da soma dos $A(1 : (i-1)r)$ elementos do vetor e com esse valor, no passo 4 são computadas as somas de prefixo dos $A((i-1)r+1 : ir)$ elementos do vetor A (com relação aos $(i-1)r+j, 1 \leq j \leq r$ elementos de A). Esses passos podem ser feitos sem necessidade de comunicação entre os processadores.

A computação local executada por cada processador P_i, $1 \leq i \leq p$ consiste de r operações nos Passos 1 e 4. No passo 3, cada processador P_i executa $i-1, 1 \leq i \leq p$ operações. Portanto, o tempo de computação do algoritmo é $O(n/p)$. Por outro lado, cada processador P_i tem que receber $p-1$ mensagens, todas no mesmo superpasso (BSP) ou rodada (CGM), logo o algoritmo termina com $O(1)$ rodadas de comunicação. $\square$

2. Ordenação

Outro procedimento básico e importante é o de ondenar um conjunto de $O(n)$ itens que estão armazenados em p processadores cada um com $O(n/p)$ itens.

O algoritmo de Cole [4] efetua a ordenação de n elementos em tempo paralelo $O(\log n)$ usando n processadores no modelo PRAM CREW. Esse algoritmo foi adaptado para o modelo CGM por Goodrich [12]. No algoritmo proposto por Goodrich [12], os $O(n)$ elementos estão uniformemente distribuídos entre os p processadores e a ordenação dos $O(n)$ elementos é efetuada com tempo de computação local $O(n \log n/p)$ usando

$O(\log n/\log(h+1))$ rodadas de comunicação, onde $p < n^{1-\frac{1}{c}}(c \geq 1)$, e $h = \Theta(\frac{n}{p})$, ou seja, com computação local ótima e $O(1)$ rodadas de comunicação quando $\frac{n}{p} \geq p$.

Embora esses algoritmos sejam ótimos, do ponto de vista teórico, suas implementações utilizam uma grande quantidade de detalhes. Vamos agora descrever um algoritmo de ordenação mais simplesmais, o CGM *split-sort*. Esse algoritmo é um pouco menos escalável, mas ainda é ótimo e com muito poucas rodadas de comunicação, para $\frac{n}{p} \geq p$.

Algoritmo 8 CGM Split Sort
Entrada: (1) Um vetor A com n elementos. (2) p processadores $p_0, p_1, \ldots, p_{p-1}$. (3) Os elementos do vetor A são distribuídos entre os p processadores (n/p elementos por processador).
Saída: Todos elementos ordenados dentro de cada processador e por processador, ou seja, se $i < j$, temos que os elementos em p_i são menores que os elementos pertencentes a p_j.

(1) `Compute um conjunto divisor` $S = \{s_1, s_2, \ldots, s_{p-1}\}$`;`

(2) `broadcast(S,`p_0`);`

(3) `Particionar os elementos de` p_i `em buckets` B_j^i `de acordo com` S`;`

(4) `send(`B_j^i`,`p_j`);`

(5) `Ordene` $B_i^k = B_i^0 \cup B_i^1 \ldots B_i^{p-1}$`;`

— Fim do Algoritmo —

É fácil verificar que o Algoritmo 8 ordena qualquer entrada, visto que não foi efetuado nenhuma restrição ao tamanho dos *buckets*. Como no modelo CGM temos que cada processador tem $O(n/p)$ memória local, devemos escolher cuidadosamente o conjunto S, pois isso influenciará no tamanho dos *buckets*.

Vamos apresentar um algoritmo CGM para computar o conjunto S (conjunto *splitter*), que utiliza apenas $O(p)$ espaço de memória por processador. O método é baseado no particionamento da entrada em p subconjuntos do mesmo tamanho como segue.

Definição 1 *A* **mediana** *de um conjunto ordenado de n números é o $(n+1)/2$-ésimo elemento de n para n ímpar ou a média do $n/2$-ésimo com $(n+1)/2$-ésimo elemento.*

Definição 2 *Os p-***quartis** *de um conjunto ordenado A de tamanho n são os $p-1$ elementos, de índices $\frac{n}{p}, \frac{2n}{p}, \ldots, \frac{(p-1)n}{p}$, que dividem A em p partes de igual tamanho.*

Os p-quartis podem ser facilmente computados de forma sequencial usando um algoritmo recursivo em tempo $O(n \log p)$.

Algoritmo 9 p-quartis Sequencial
Entrada: (1) Um vetor A com n elementos. (2) p processadores $p_0, p_1, \ldots, p_{p-1}$ o número de quartis.
Saída: O conjunto A dividido em p-quartis.

(1) `Compute a mediana de` A`;`

(2) `Usando a mediana, divida` A `em dois conjuntos` A_1 `e` A_2`;`

(3) `Aplique o algoritmo recursivamente, até que` $p - 1$ `splitters sejam encontrados;`

— Fim do Algoritmo —

Usamos o Algoritmo 9 no algoritmo CGM para computar S.

Algoritmo 10 p**-quartis CGM**
Entrada: (1) Um vetor A com n elementos. (2) p processadores $p_0, p_1, \ldots, p_{p-1}$. (3) Os elementos do vetor A são distribuídos entre os p processadores (n/p elementos por processador).
Saída: O conjunto A dividido em p-quartis.

(1) $Q_i \leftarrow p$`-quartis(`A_i`);`

(2) `send(`Q_i`,`p_0`);`

(3) **se** $i = 0$ **então** `S` $\leftarrow$ `Ordena(`$Q_0 \cup Q_1 \ldots Q_{p-1}$`;`

(4) `broadcast(S,`p_i`);`

— Fim do Algoritmo —

Teorema 3 *O algoritmo para a determinação dos p-quartis de um conjunto de n elementos é executado no modelo BSP/CGM com p processadores em $O(1)$ rodadas de comunicação e tempo de computação local de $O(\frac{n \log p}{p})$, onde $\frac{n}{p} = \Omega(p^2)$.*

Demonstração: □

O teorema acima pode sem melhorado de tal forma que $\frac{n}{p} = \Omega(p)$.

Corolário 1 *A ordenação de um conjunto de n elementos pode ser executada no modelo BSP/CGM com p processadores em $O(1)$ rodadas de comunicação e tempo de computação local de $O(\frac{n \log p}{p})$, onde $\frac{n}{p} = \Omega(p)$.*

3. List Ranking

Seja L uma lista representada por um vetor s tal que $s[i]$ é o nó sucessor de i na lista L, para u, o último elemento da lista L, $s[u] = u$. A **distância** entre i e j, $d_L(i,j)$, é o número de nós entre i e j mais 1. O problema do **list ranking** consiste de computar para cada $i \in L$ a distância entre i e j, denotado $rank_L(i) = d_L(i,j)$.

Diferentemente dos problemas da soma e somas de prefixos de n números, onde as tarefas (somas parciais) foram feitas de forma independente por cada um dos p processadores, no problema do *list ranking* não é possível aplicar a idéia do Teorema de Brent. O número de nós da lista cujos sucessores não estão armazenados no mesmo processador pode variar de 0 a n/p. Mesmo no caso em que todos os sucessores estejam no mesmo processador no passo inicial, nada garante que com a aplicação da duplicação recursiva (*pointer jumping*) isso continuará ocorrendo no passo seguinte.

A Figura 2.1 ilustra uma lista ligada com os ($n = 24$) elementos armazenados nos ($p = 4$) processadores. Cada processador armazena $n/p = 6$ elementos.

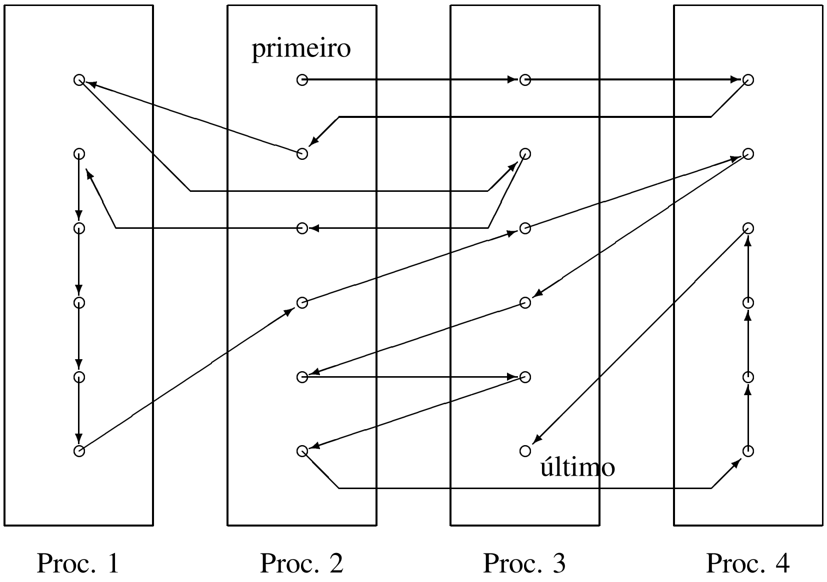


Figura 2.1: Uma lista ligada armazenada nos processadores

Pode ocorrer que a cada iteração, pelo menos um dos processadores sempre necessite efetuar uma comunicação para obter o sucessor de um de seus elementos. Logo, o número de rodadas de comunicação para obter o *list ranking* pode chegar a $O(\log n)$. Ou seja a simples aplicação da duplicação recursiva na lista L não leva a um algoritmo CGM eficiente. Como vimos anteriormente, o número de rodadas de de comunicação de um algoritmo CGM eficiente deve ser da ordem de ($O(\log^k p), k \in N$).

Para diminuir o número de rodadas de comunicação, a idéia é a de selecionar um conjunto de elementos $i^* \in L$, bem distribuídos em L, de tal forma que a distância de qualquer $i \in L$ a i^* possa ser computada com $O(\log^k p)$ aplicações de *pointer jumping*. Um conjunto com essas características é definido a seguir.

Um **r-ruling set** de L é um subconjunto de elementos selecionados de lista L com as seguintes propriedades: (1) Dois vizinhos nunca são selecionados. (2) A distância entre qualquer elemento não selecionado ao próximo elemento não selecionado é no máximo r.

Logo, a estratégia para computar o algoritmo do list ranking é a seguinte:

1. Computar um $O(p^2)$-ruling set R com $|R| = O(n/p)$ e *broadcast* R para todos os processadores. O subconjunto R é representado por uma lista ligada onde para cada elemento i é atribuído um ponteiro para o próximo elemento j de R com respeito à ordem induzida por L.
2. Todo processador efetua sequencialmente o *list ranking* de R, computando para cada $i \in R$ sua distância ao último elemento da lista.
3. A distância dos demais elementos da lista é obtida através da duplicação recursiva *pointer jumping* até que um elemento de R seja alcançado. Todos os outros elementos da lista têm no máximo a distância $O(p^2)$ do próximo elemento de R na lista.

Todos os passos, exceto a computação do $O(p^2)$-ruling set R, podem ser facilmente implementados em $O(\log p)$ rodadas de comunicação.

Vamos introduzir uma nova técnica, denominada **compressão determinística de listas**, que possibilita que um $O(p^2)$-ruling set R possa ser computado em $O(\log^k p)$ rodadas de comunicação. Uma **compressão determinística de listas** é composta de uma sequencia alternada de **fases de compressão e de concatenação**.

Na **fase de compressão**, utilizando um esquema de rotulação (*deterministic coin tossing* de Cole e Vishkin [5] adaptado ao modelo BSP/CGM) selecionamos um subconjunto de elementos da lista L. A **fase de concatenação** consiste da construção de uma lista ligada, através da duplicação recursiva, com os elementos selecionados na fase de compressão.

O rótulo usado, $l(i) \forall i \in L$, na fase de compressão é o número do processador p que armazena o nó i. O primeiro elemento da lista da Figura 2.1 está armazenado no processador 2, portanto será rotulado com o número 2, da mesma forma, o último elemento é rotulado com o número 3. A Figura 2.2 mostra os rótulos de cada elemento da lista.

Temos que esse esquema rotulação faz com que os elementos de L possuam no máximo p rótulos distintos. Seja M um subconjunto de elementos $i, i+1, \ldots, i+k \in L$, tal que $l(i) \neq l(s[i]), \forall i \in M$, definimos $s[i]$ como sendo um **máximo local** se $l(i) < l(s[i]) > l(s[s[i]])$.

Selecionando apenas os máximos locais, não podemos garantir que a distância entre eles seja menor $O(p)$, pois pode haver um subconjunto L' de elementos consecutivos $j, j+1, \ldots j+k, j+l$ de L onde $l(suc[j]) = l(j), \forall j \in L'$ e $k > p$. Para contornar esse problema, sempre que tivermos um subconjunto com essas características, selecionamos todos os segundos elementos.

Algoritmo 11 p**-ruling set**
Entrada: (1) Uma lista ligada L representada pelo vetor s onde $s[i]$ é o sucessor de i na lista L. (2) p processadores $p_0, p_1, \ldots, p_{p-1}$.
Saída: Um subconjunto $R \subset L$ de nós selecionados.

(1) **para** $i = p_j * (n/p) + 1$ **até** $(p_j + 1) * (n/p)$ **faça**
  (1.1) `sel[i]` $\leftarrow$ `não selecionado;`

(2) **para** $i = p_j * (n/p) + 1$ **até** $(p_j + 1) * (n/p)$ **faça**
  (2.1) **se** $l(i) < l(s[i]) > l(s[s[i]])$ **então** `sel(s[i])` $\leftarrow$ `selecionado;`
  (2.2) **se** $l(i) = l(s[i])$ **então**
    se `l(s[i]) = l(s[s[i]])` **então**
      `sel[s[i]]` $\leftarrow$ `selecionado;`
    caso contrário se $l(s[i]) > l(s[s[i]])$ **então** `sel[i]` $\leftarrow$ `selecionado;`

— Fim do Algoritmo —

O Algoritmo 11 computa um $O(p)$-ruling set R, mas $|R|$ pode ser igual a $O(n)$, pois se os n/p elementos da lista L em cada processador p_j tiverem rótulo igual a j, serão selecionados $n/2$ elementos. Para obter R com $O(n/p)$ elementos necessitamos executar o Algoritmo 11 $\log p$ vezes.

A Figura 2.2 ilustra uma aplicação do Algoritmo 11. Os elementos selecionados são mostrados com um círculo preenchido de preto.

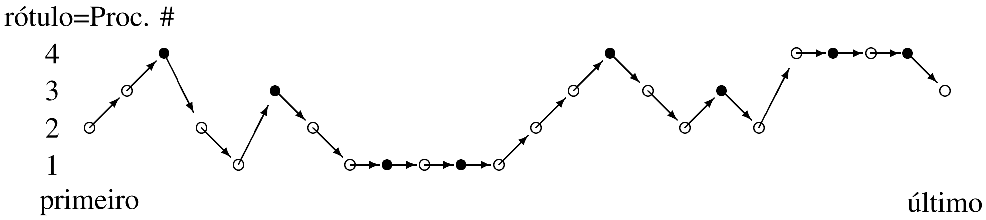


Figura 2.2: Elementos rotulados pelos números dos processadores

Para obter um p^2-*ruling set* R de elementos selecionados com $O(n/p)$ elementos, necessitamos desmarcar elementos que foram selecionados. Para isso, contruímos uma nova lista ligada com os elementos selecionados e aplicamos novamente o precedimento de marcação (máximos locais e segundos elementos de sublistas no mesmo processador). A Algoritmo 12 computa um subconjunto $R \subset L$ com essas características.

Algoritmo 12 p^2**-ruling set**
Entrada: (1) Uma lista ligada L representada pelo vetor s onde $s[i]$ é o sucessor de i na lista L. (2) p processadores $p_0, p_1, \ldots, p_{p-1}$ e LC uma cópia de L
Saída: Um subconjuno $R \subset L$ de nós selecionados e $|R| = O(n/p)$.

(1) **para** $k = 1$ **até** $\log p$ **faça**
  (1.1) `R` $\leftarrow p$`-Ruling_Set(LC);`
  (1.2) `LC` $\leftarrow$ `R;`

— Fim do Algoritmo —

Teorema 4 *O Algoritmo 12 computa um* p^2*-ruling set* R*, onde* $|R| = O(\frac{n}{p})$ *usando* $O(\log^2 p)$ *rodadas de comunicação e* $O(\frac{n}{p})$ *computação local por rodada.*

Demonstração: exercício.

Para obter um algoritmo que compute um p^2-*ruling set* R com $O(\frac{n}{p})$, temos que arrumar uma estratégia diferente para diminuir o número de elementos selecionados no Algoritmo 11.

Temos que se dois elementos selecionados estão a uma distância $\Theta(p)$ a um dados momento, então não é necessário aplicar novamente a compressão para reduzir o número de elementos selecionados. A abordagem básica do algoritmo a seguir é a de intercalar duplicação recursiva (concatenação) com compressão. Mais precisamente, somente aplicaremos um passo de duplicação recursiva entre passos consecutivos compressão, e essa duplicação recursiva não será aplicada aos elementos da lista que estão apontando para elementos selecionados.

Algoritmo 13 p^2**-ruling set**
Entrada: (1) Uma lista ligada L representada pelo vetor s onde $s[i]$ é o sucessor de i na lista L. (2) p processadores $p_0, p_1, \ldots, p_{p-1}$ e LC uma cópia de L
Saída: Um subconjuno $R \subset L$ de nós selecionados e $|R| = O(n/p)$.

(1) `R` $\leftarrow p$`-Ruling_Set(LC);`

(2) **para** $k = 1$ **até** $\log p$ **faça**

(2.1) **para todo** $i \in L$ **em paralelo faça**

  se `s[i] = não selecionado` **então** `s[i]` $\leftarrow$ `s[s[i]];`

(2.2) **para todo** $i \in L$ **em paralelo faça**

  se `(i, s[i] e s[s[i]]) são selecionados` **e** `NOT ((l(i)<l(s[i])>l(s[s[i]])) AND (l(i)` $\neq$ `l(s[i])) AND (l(s[i])` $\neq$ `l(s[s[i]])))` **então**

   `s[i]` $\leftarrow$ `não selecionado;`

(2.3) `Sequencialmente, cada processador, processa as sublistas de elementos subsequentes que estão armazenadas no mesmo processador. Para cada sublista, marque todo segundo elementos como não selecionado. Se uma das sublistas possui apenas dois elementos, marque ambos elementos como não selecionados;`

(3) `Selecione o último elemento.`

— Fim do Algoritmo —

Inicialmente vamos demonstrar que o conjunto de elementos selecioandos ao final do Algoritmo 13 é do tamanho $O(\frac{n}{p})$.

Lema 1 *Após a k-ésima iteração no Passo 2, não existem mais que dois elementos selecionados entre quaisquer 2^k elementos subsequentes na lista original.*

Demonstração:

Vamos agora mostrar que elementos subsequentes selecionados ao final do Algoritmo 13 tem, no máximo, distância $O(p^2)$. Esse resultado é dados pelos Lemas 2, 3 e 4.

Lema 2 *Após cada execução do Passo 2.3, a distância de dois elementos subsequentes selecionados, com respeito aos ponteiros atuais (representado pelo vetor s), é no máximo $O(p)$.*

Demonstração:

Lema 3 *Após a k-ésima iteração do Passo 2.3, dois elementos subsequentes com respeito aos ponteiros atuais (representado pelo vetor s) tem distância $O(2^k)$ com repseito a lista original L.*

Demonstração: Temos que somente k duplicações recursivas foram efetuadas no Passo 2.1. □

Lema 4 *Quaisquer dois elementos subsequentes selecionados não distam um do outro mais que $O(p^2)$ com respeito a lista original L.*

Demonstração: Lema 2 e Lema 3. □

Teorema 5 *O problema do list ranking para uma lista L com n vértices pode ser resolvida no modelo CGM com p processadores e $O(\frac{n}{p})$ memória local por processador usando $O(\log p)$ rodadas de comunicação e $O(\frac{n}{p})$ computação local por rodada.*

Capítulo 3

Alguns Algoritmos para Problemas em Grafos

Objetivos

- Introduzir diversos problemas em computação de funções em árvores.
- Analisar a eficiência dos algoritmos apresentados.

1. Euler Tour em Árvores

Vários algoritmos para problemas em árvores e grafos incluem soluções para problemas estruturais básicos (problemas de enumeração e contagem) como subrotinas. Exemplos de tais subproblemas para árvores são:

1. Encontrar o pai de cada vértice.
2. Calcular a numereção pré e pós ordem.
3. Computar o nível de cada vértice.
4. Determinar o número de descendentes de cada vértice.

Todos esses problemas possuem soluções sequenciais em tempo linear que usam basicamente a busca em profundidade. A técnica do **Euler Tour**, proposta por Tarjan e Vishkin [24], é uma aplicação do *list ranking* que pode ser utilizado para construir algoritmos paralelos eficientes para esses problemas.

Seja $T = (V, E)$ uma árvore com $V = \{1, \ldots, n\}$ e $v_i \in V$ designado como sua **raiz**. Seja $T_D = (V, E')$ um grafo dirigido obtido a partir de T onde cada aresta $(u, v) \in E$ é substituída por dois arcos (arestas dirigidas) (v, w) e (w, v). Visto que o grau de entrada de cada vértice de T' é igual ao grau de saída, T' é um **grafo Euleriano**; isto é, possui um caminho fechado (circuito) $\{v_0, v_1, \ldots, v_n, v_0\}$ que percorre cada aresta uma única vez, sendo que os vértices podem ser visitados um número qualquer de vezes. Esse circuito é denominado **Euler Tour** de T'.

Assumimos que a árvore T está representada pelas listas de suas arestas. Isto é, para cada vértice v, temos uma lista de arestas que conectam v a seus vizinhos u_i. Cada aresta (v, w) aparece duas vezes na estrutura, uma vez na lista de v e outra na lista de w. Cada aresta (v, w) possui um ponteiro para a **próxima** aresta da lista v, (esse ponteiro na última aresta é **nil**) e um ponteiro para sua cópia na lista de w. Para o grafo T_D simplesmente interpretamos uma aresta como sendo dirigida de v para w quando ela aparece na lista de v, e de w para v quando ela aparece na lista de w. A aresta dirigida $e_R = (w, v)$ é denominada **reversa** da aresta $e = (v, w)$.

O vetor $ListaAdj$ tem n entradas; $ListaAdj[v]$ é um ponteiro para a primeira aresta da lista de arestas incidentes com v.

Uma das formas de organizar a estrutura de dados é a de armazenar todas as arestas em um vetor $EDGE$ de tal forma que que todas as k arestas originando no vértice 1 estejam armazenadas nas primeiras k posições de E, seguidas das arestas originando no vértice 2 e assim sucessivamente. Essa estrutura de dados pode ser facilmente construída através de ordenações do conjunto de arestas.

Dado as listas de adjacências, usando um algorimto de ordenação, é possível construir a estrutura de dados necessária para o Algoritmo 14 que computa um *Euler Tour* para T_D.

Algoritmo 14 Euler Tour de uma árvore
Entrada: (1) O número de processador i; (2) O número p de processadores; (3) O i-ésimo sub-vetor $BEDGE = EDGE((i-1)r+1 : ir)$ de tamanho r, onde o $EDGE$ armazena as arestas de T_D. Cada elemento do vetor $EDGE$ é composto de três campos: (i) índice da aresta; (ii) ponteiro para a aresta reversa e (iii) ponteiro para a próxima aresta; (4) Um vetor $ListaAdj$ onde cada elemento contém um apontador para o início do bloco referente a cada vértice no vetor $EDGE$.
Saída: Um vetor com $2n - 2$ entradas representando um *Euler Tour* de T_D como uma lista ligada. Ou seja, $ETourLink[e]$ é o índice da aresta sucessora de e no *tour*.

(1) **para** $k = 1$ **até** r **faça**
  se `BEDGE[BEDGE[e].reverso].próximo` $\neq$ **nil estão**
    `ETourLink[e]` $\leftarrow$ `BEDGE[BEDGE[e].reverso].próximo;`
    caso contrário `ETourLink[e]` $\leftarrow$ `ListaAdj[e.y];`

— Fim do Algoritmo —

Teorema 6 *O Euler Tour de uma árvore T com n vértices pode ser computado no modelo CGM com p processadores e $O(\frac{n}{p})$ memória local por processador usando $O(\log p)$ rodadas de comunicação e com $O(\frac{n}{p})$ computação local por rodada.*

Demonstração:

O Algoritmo 14 não faz uso da raiz da árvore. O algoritmo canstrói um *Euler tour* para uma árvore não enraizada. Se designarmos uma aresta saindo da *raiz*, digamos $(raiz, v)$ como sendo a primeira aresta do *tour*, enão o *tour* percorre a árvore enraizada na ordem de uma busca em profundidade.

O Algoritmo 14 tem como saída um *Euler tour* representado através de uma lista ligada. Para algumas aplicações, necessitamos saber a posição de cada aresta no *tour*. Isso pode ser feito com a utilização do algoritmo de *list ranking*. Os $ETourLink$s são os ponteiros para os sucessores no problema do *list ranking*. Os ponteiros dos predecessores de um nó numa lista podem ser obtidos em tempo contante. O predecessor de $(raiz, v)$ é a última aresta do *tour*,e podemos alterar o ponteiro de seu sucessor para apontar para a própria raiz, com isso o *tour* tem a a mesma estrutura da entrada do problema do *list ranking*.

Algoritmo 15 Euler Tour de uma árvore
Entrada: (1) O número de processador i; (2) O número p de processadores; (3) O i-ésimo sub-vetor $BEDGE = EDGE((i-1)r+1 : ir)$ de tamanho r, onde o $EDGE$ armazena as arestas de T_D. Cada elemento do vetor $EDGE$ é composto de três campos: (i) índice da aresta; (ii) ponteiro para a aresta reversa e (iii) ponteiro para a próxima aresta; (4) Um vetor $ListaAdj$ onde cada elemento contém um apontador para o início do bloco referente a cada vértice no vetor $EDGE$.
Saída: Um vetor com $2n-2$ entradas representando um *Euler Tour* de T_D como uma lista ligada. Ou seja, $ETourLink[e]$ é o índice da aresta sucessora de e no *tour* e um vetor $posn$ onde $posn[e]$ é a posição de e no *tour*..

(1) `Aplicar o Algoritmo 14;`

(2) `O processador` p_i `que contém a aresta` $e = (raiz, v)$ `atribui ETourLink[e]` $\leftarrow$ `e;`

(3) `Aplicar o algoritmo de` *list ranking* `na lista` $ETourLink$ `e armazenar o resultado em` $rank$`;`

`(Os` *ranks* `variam de zero para a última aresta a 2n-3 para a primeira)`

(4) `Cada processador` p_i `computa posn[e]` $\leftarrow$ `2n-2-rank[e] para suas arestas;`

— Fim do Algoritmo —

2. Ancestral Comum mais Baixo

O **ancestral comum mais baixo** de dois vértices u e v de uma árvore com raiz é o vértice w que é um ancestral de u e v e que está mais distante da raiz. $w = \text{LCA}(u, v)$.

Não estamos interessados somente no problema de dado um par de vértices u e v determinar $w = \text{LCA}(u, v)$. O que queremos é um caso mais geral e aplicável em uma série de problemas como descrito em Reif [21]. Estamos interessados em, dada uma árvore $T = (V, E)$ com raiz, pré-processar T de forma que uma consulta LCA(u, v), para um par qualquer de vértices u e v de T, possa ser respondida rapidamente - em tempo sequencial $O(1)$.

Para este problema, conhecido como Problema do Ancestral Mais Baixo (LCA – *Lowest Common Ancestor*), temos dois casos especiais:

1. T é simplesmente um caminho. Neste caso, basta determinarmos a distância de cada vértice à raiz, o que nos permite responder a LCA(u, v) em tempo constante, comparando-se as distâncias de u e v à raiz.
2. T é uma árvore binária completa. Neste caso, dada uma árvore binária completa T com raiz, inicialmente determina-se o número *in-ordem* dos vértices de T. O

número *in-ordem* dos vértices de uma árvore corresponde à ordem dos vértices obtida em um percurso *in-ordem* na árvore a partir da raiz. A partir disto, cada vértice será identificado através do seu número *in-ordem* e convenientemente dado em sua representação binária. Indexamos os bits na representação binária do bit menos significativo, que tem índice 0, para o mais significativo, que tem índice l. Dados dois vértices x e y da árvore T, seja i a posição do bit mais à esquerda em que x e y diferem. LCA(x, y) consiste dos $l - i$ bits mais à esquerda, seguidos por um "1"e i "0"s. Assim, após a determinação dos números *in-ordem*, podemos responder a consultas LCA(x, y) em tempo $O(1)$.

Quando a árvore é arbitrária, a solução para o problema não é imediata. Existem diversos algoritmos, sequenciais e paralelos, que pré-processam a árvore de entrada para consultas LCA. Na seção seguinte descreveremos uma das possíveis soluções, utilizando o modelo CGM. Os algoritmos CGM obtidos para este problema são baseados em algoritmos desenvolvidos para o modelo PRAM. Uma estratégia básica descrita em [21], para o modelo PRAM e que também é sequencialmente eficiente, consiste no mapeamento da árvore de entrada T em uma árvore binária de altura logarítmica. Entretanto, vamos nos basear nas idéias desenvolvidas em [15] e que utiliza a técnica *Euler Tour* e o problema de mínimos intervalares.

2.1. O Algoritmo para uma Árvore Arbitrária

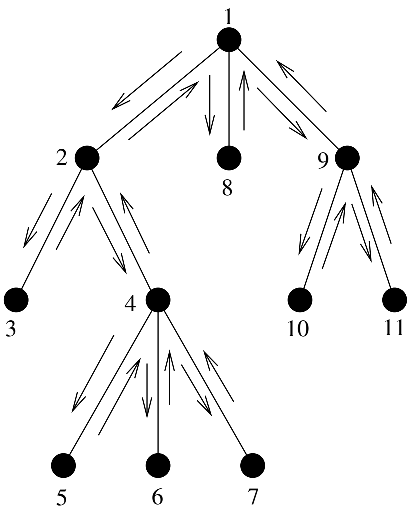
Consideraremos uma árvore com raiz $T = (V, E)$, onde $|V| = n$. A árvore encontra-se inicialmente distribuída entre os processadores de forma que seus nós estão aleatoriamente armazenados nas memórias locais dos p processadores com apontadores para os respectivos pais de cada nó na árvore. Se o pai de determinado nó não encontra-se armazenado no mesmo processador, o índice do processador que o armazena é conhecido. O pai da raiz é ela própria.

O primeiro passo do algoritmo consiste em fazer um *Euler Tour* na árvore de forma a obter, para cada vértice, o seu número em pré-ordem. Os vértices do percurso que foram obtidos estarão armazenados em um vetor A, indexado pela posição em que o vértice aparece no percurso. O vetor A estará distribuído de forma contígua entre os processadores. Como vimos na seção anterior, a obtenção de um *Euler Tour* leva tempo $O(n/p)$, usando $O(\log p)$ rodadas de comunicação.

O passo seguinte consiste em determinar $nivel(v)$, para cada vértice v da árvore, contendo o nível do vértice na árvore. Determinamos também $esq(v)$ e $dir(v)$, os índices das ocorrências mais à esquerda e mais à direita de v em A, respectivamente. Os níveis dos vértices podem ser obtidos, através de somas prefixas, em tempo $O(n/p)$, usando $O(1)$ rodadas de comunicação. Na Figura 3.1, temos o vetor A calculado para a árvore dada.

Considerando o vetor A, $a_i = v$ é a ocorrência de v em A mais à esquerda se e somente se $nivel(a_{i-1}) = nivel(v) - 1$. De forma semelhante, $a_i = v$ é a ocorrência de v em A mais à direita se e somente se $nivel(A_{i+1}) = nivel(v) - 1$. Por conseguinte, esq e dir podem ser obtidos em tempo $O(n/p)$ e, possivelmente, uma rodada de comunicação[1],

[1]Se o vetor A, distribuído contiguamente entre os processadores, contiver também as posições anterior e posterior ao subvetor localmente armazenado, não será necessária nenhuma rodada de comunicação para determinar os vetores esq e dir.



A={1,2,3,2,4,5,4,6,4,7,4,2,1,8,1,9,10,9,11,9,1}

B={0,1,2,1,2,3,2,3,2,3,2,1,0,1,0,1,2,1,2,1,0}

nivel={0,1,2,2,3,3,3,1,1,2,2}

esq={1,2,3,5,6,8,10,14,16,17,19}

dir={21,12,3,11,6,8,10,14,20,17,19}

Figura 3.1: Uma árvore dada e os vetores A, B, $nivel$, esq e dir obtidos para a árvore.

uma vez que o vetor de níveis encontrar-se-á distribuído contiguamente entre os processadores.

O problema LCA, então, será reduzido ao processamento do vetor $B = nivel(A)$. Este vetor contém, em cada posição, o valor do nível na árvore do respectivo vértice e é obtido em tempo $O(n/p)$, usando 1 rodada de comunicação. A justificativa para esta redução é dada pelo lema a seguir.

Lema 5 *Dada uma árvore com raiz $T = (V, E)$. Sejam A, $nivel(v)$, $esq(v)$, $dir(v)$, para cada $v \in V$, como definido. Sejam u e v dois vértices quaisquer de T, $u \neq v$. Então as seguintes afirmações são válidas.*

1. *u é ancestral de v se e somente se $esq(u) < esq(v) < dir(u)$.*
2. *u e v não estão relacionados, isto é, u não é ancestral de v e v não é ancestral de u, se e somente se ou $dir(u) < esq(v)$ ou $dir(v) < esq(u)$.*
3. *Se $dir(u) < esq(v)$, então $LCA(u, v)$ é o vértice com nível mínimo dentre aqueles no intervalo $[dir(u), esq(v)]$.*

Demonstração: Afirmação 1. Vamos supor que u seja ancestral de v. Como o *Euler Tour* em T corresponde a uma busca em profundidade na árvore a partir da raiz, u é visitado antes de v e, além disso, a subárvore com raiz em v é totalmente visitada antes da última ocorrência de v. Daí, temos que $esq(u) < esq(v) < dir(u)$. Reciprocamente, vamos supor que $esq(u) < esq(v) < dir(u)$ e que u não seja ancestral de v. Como $esq(u) < esq(v)$, a subárvore com raiz em u é completamente visitada antes de v ser visitado pela primeira vez. Mas então $dir(u) < esq(v)$ o que contradiz a hipótese. Portanto, u deve ser ancestral de v.

A Afirmação 2 tem demonstração similar.

Afirmação 3. Vamos supor que $dir(u) < esq(v)$. Pode-se ver facilmente que todos os vértices cujos níveis aparecem no intervalo $[dir(u), esq(v)]$ ou fazem parte do caminho entre u e v, ou são seus descendentes, o que inclui também o $LCA(u, v)$ pois este estará no caminho entre u e v. Portanto, o vértice de nível mínimo deve ser o LCA de u e v. $\square$

Dessa forma, vemos que, para determinar o $LCA(u, v)$, precisamos pré-processar o vetor B de forma que possamos responder, em tempo constante, a consultas de mínimo intervalar. A solução para este problema será discutida na próxima seção e, como veremos, o pré-processamento levará tempo $O(n/p)$, usando $O(1)$ rodadas de comunicação.

Com a descrição apresentada, podemos observar que o tempo do algoritmo para o problema LCA é $O(n/p)$, usando $O(\log p)$ rodadas de comunicação.

Na Figura 3.1 se quisermos determinar $LCA(3, 7)$ procedemos da forma descrita a seguir. Como estes vértices não estão relacionados, o LCA será o vértice de menor nível no intervalo $[dir(3), esq(7)]$. Temos que, $dir(3) = 3$ e $esq(7) = 10$. O mínimo intervalar entre as posições 3 e 10, no vetor B, vale 1 e corresponde ao vértice 2 em A, que é justamente o valor do LCA.

3. Minimo Intervalar

Dado um vetor $A = (a_0, a_1, \ldots, a_{n-1})$ de números reais, definimos $MIN(i, j) = \min\{a_i, \ldots a_j\}$. O **problema de mínimo intervalar (*range minima*)** consiste em pré-processar o vetor A de forma que consultas $MIN(i, j)$ possam ser respondidas em tempo constante, para todo $0 \leq i, j \leq n - 1$.

O algoritmo para o problema de mínimo intervalar no modelo CGM, descrito em [19] usa os algoritmos seqüenciais de Gabow *et al* [10] e Alon e Schieber [2] que são executados usando os dados locais em cada processador.

3.1. Algoritmo Sequencial de Gabow *et al*

Este algoritmo usa a estrutura de dados árvore Cartesiana descrita a seguir e nela é aplicado um algoritmo para o problema do LCA.

Dado um vetor $A_{0,n-1} = (a_0, a_1, \ldots, a_{n-1})$ de n nú-meros reais distintos, a árvore Cartesiana de $A_{0,n-1}$ é uma árvore binária com nós rotulados com os elementos do vetor A. A raiz da árvore tem rótulo $a_m = \min\{a_0, a_1, \ldots, a_{n-1}\}$. Sua subárvore esquerda é a árvore Cartesiana de $A_{0,m-1} = (a_0, a_1, \ldots, a_{m-1})$, e sua árvore direita é a árvore Cartesiana de $A_{m+1,n-1} = (a_{m+1}, \ldots, a_{n-1})$. A árvore Cartesiana de um vetor vazio é a árvore vazia.

O algoritmo é dado a seguir.

Algoritmo 16 Mínimo_Intervalar(Gabow et al)
Entrada: o vetor $A = (a_0, a_1, \ldots, a_{n-1})$ de n números reais.
Saída: uma estrutura de dados que responde a consultas $MIN(i, j)$ em tempo constante.

(1) Construir a árvore Cartesiana de A.

(2) Usar um algoritmo sequencial linear para o problema LCA, usando a árvore Cartesiana.

— Fim do Algoritmo —

A construção da árvore Cartesiana leva tempo linear, bem como o algoritmo para o problema do LCA. O algoritmo leva, então, tempo linear para obter a estrutura que responde a consultas de Mínimo Intervalar em tempo constante. Qualquer consulta $MIN(i, j)$ pode ser respondida da seguinte forma: da definição recursiva de árvore Cartesiana temos que o valor de $MIN(i, j)$ é o valor do LCA de a_i e a_j. Assim, cada consulta de mínimo intervalar pode ser respondida em tempo constante através de uma consulta de LCA em uma árvore Cartesiana. Logo, o problema de mínimo intervalar é solucionado em tempo constante.

3.2. Algoritmo Seqüencial de Alon e Schieber

O algoritmo de Alon e Schieber tem complexidade de tempo $O(n \log n)$. Apesar desta complexidade não-linear, este algoritmo é crucial na descrição do algoritmo CGM, como será visto na seção 3.3.. Sem perda de generalidade, vamos considerar n como sendo uma potência de 2.

Na descrição deste algoritmo, utilizamos o conceito de vetores mínimo prefixo e mínimo sufixo, vistos na seção **??**.

Algoritmo 17 Mínimo_Intervalar(Alon e Schieber)
Entrada: o vetor $A = (a_0, a_1, \ldots, a_{n-1})$ de n números reais.
Saída: uma estrutura de dados que responde a consultas $MIN(i, j)$ em tempo constante.

(1) Construir uma árvore binária completa T com n folhas.

(2) Associar os elementos de A às folhas de T.

(3) Para cada nó v de T, calcular P_v e S_v, os vetores de mínimo prefixo e mínimo sufixo, respectivamente, das folhas da subárvore com raiz v.

— Fim do Algoritmo —

A árvore T construída pelo algoritmo *Mínimo_Intervalar(Alon e Schieber)* será denominada árvore-PS. Na Figura 3.2 está ilustrada uma árvore-PS gerada por este algoritmo.

Vamos introduzir a seguinte notação: Considere uma árvore binária completa e um nó v da árvore. Denotemos $a_r, a_{r+1}, \ldots, a_s$ as folhas abaixo de v que são descendentes de v. Para cada j tal que $r \leq j \leq s$, definimos $Pos(a_j) = j - r$.

O processamento das consultas $MIN(i, j)$ com $0 \leq i, j \leq n - 1$ é feito como se segue. Determinamos $w = \text{LCA}(a_i, a_j)$ em T. Sejam v e u os filhos esquerdo e direito de w, respectivamente. Então o valor de $MIN(i, j)$ é o mínimo entre o valor de $S_v[Pos(a_i)]$ e o valor de $P_u[Pos(a_j)]$.

A árvore-PS é uma árvore binária completa e, como vimos, consultas LCA em árvores binárias completas podem ser feitas em tempo constante.

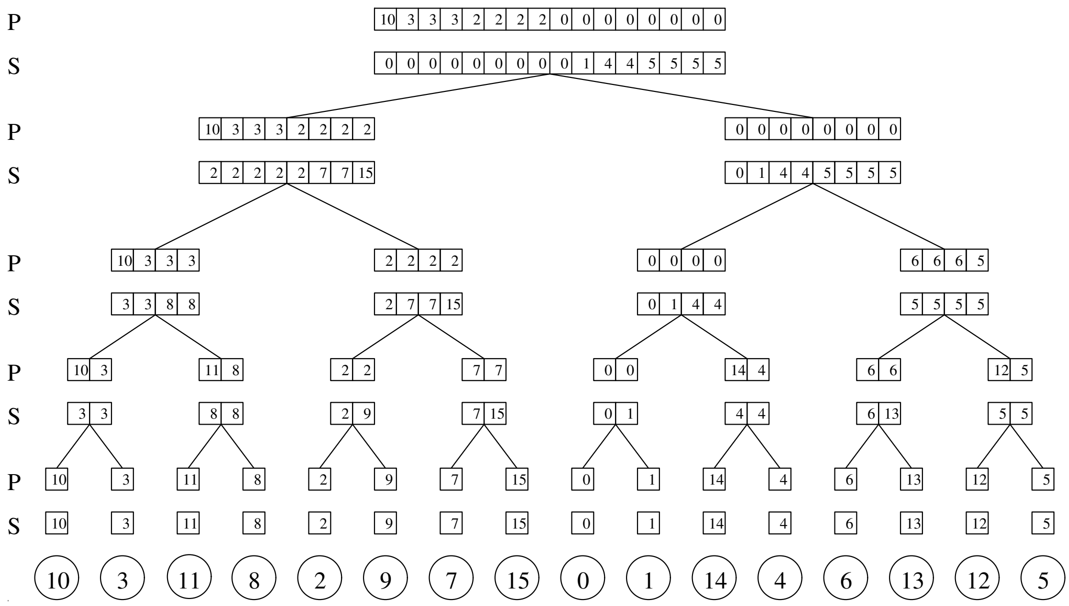


Figura 3.2: árvore-PS gerada pelo Algoritmo *Minimo_Intervalar(Alon e Schieber)* para um vetor particular.

3.3. O Algoritmo CGM

Em um algoritmo CGM, estamos interessados em reduzir o número de rodadas de comunicação e concentrar na computação de dados locais nos processadores.

Baseado nos algoritmos sequenciais vistos nas seções anteriores, podemos construir um algoritmo CGM para o problema de mínimo intervalar.

O algoritmo CGM de Mongelli e Song [19] é executado em tempo $O(n/p)$ e usa $O(1)$ rodadas de comunicação e utiliza os algoritmos sequenciais vistos. A maior dificuldade é no armazenamento dos dados necessários pelos processadores de forma que as consultas possam ser feitas em tempo constante sem violar o limite de $O(n/p)$ de memória fixada pelo modelo CGM.

O modelo $CGM(n, p)$ precisa de um mínimo de $O(n/p)$ de memória em cada processador. Infelizmente, não há o que fazer se esta quantidade de memória disponível for menor que este mínimo. Primeiro, precisamos de memória suficiente para armazenar os dados de entrada. Também precisamos de memória para construir as estruturas de dados em cada processador de forma a obter tempo constante nas consultas do mínimo intervalar.

Na descrição do algoritmo, consideraremos a seguinte notação: Dado um vetor $A = (a_0, a_1, \ldots, a_{n-1})$, escrevemos $A[i] = a_i, 0 \leq i, \leq n-1$, e $A[i \ldots j] = (a_i, a_{i+1}, \ldots, a_{j-1}, a_j)$, $0 \leq i \leq j \leq n-1$.

A idéia do algoritmo é baseada em como as consultas $MIN(i, j)$ podem ser respondidas. Consideraremos p processadores denotados por $0, 1, \ldots, p-1$. Sem perda de generalidade, assumimos p como sendo um potência de 2. Cada processador armazena n/p posições contíguas do vetor de entrada. Assim, dado $A = (a_0, a_1, \ldots a_{n-1})$, o processador i armazena o subvetor $A_i = (a_{in/p}, \ldots, a_{(i+1)(n/p)-1})$, para $0 \leq i \leq p-1$.

$MIN(i,j)$ é respondido dependendo da localização de a_i e a_j nos processadores. Temos os seguintes casos:

1. Se a_i e a_j estão em um mesmo processador, o domínio do problema se reduz ao subvetor armazenado localmente naquele processador. Assim, precisamos de uma estrutura de dados para responder a este tipo de consulta em tempo constante. Esta estrutura de dados é fornecida em cada processador pelo Algoritmo *Mínimo_Intervalar(Gabow et al)*.
2. Se a_i e a_j estão em processadores distintos $\bar{i}$ e $\bar{j}$ (sem perda de generalidade, $\bar{i} < \bar{j}$), respectivamente, temos dois subcasos:
   (a) Se $\bar{i} = \bar{j} - 1$, isto é, a_i e a_j estão em processadores vizinhos, $MIN(i,j)$ então corresponde ao mínimo entre o mínimo de a_i até o final do subvetor $A_{\bar{i}}$ e o mínimo do começo do subvetor $A_{\bar{j}}$ até a_j. Estes mínimos podem ser novamente determinados pela estrutura de dados obtida pelo Algoritmo *Mínimo_Intervalar(Gabow et al)*. Para determinar o mínimo destes mínimos, precisamos de uma rodada de comunicação.

   (b) Se $\bar{i} < \bar{j} - 1$, $MIN(i,j)$ corresponde ao mínimo do subvetor $A_{\bar{i}}[i \ldots (\bar{i}+1)(n/p) - 1]$, o mínimo do subvetor $A_{\bar{j}}[\bar{j}(n/p) \ldots j]$ e os mínimos dos subvetores $A_{\bar{i}+1}, \ldots, A_{\bar{j}-1}$. Os primeiros dois mínimos são determinados como no subcaso anterior. Os mínimos dos subvetores $A_{\bar{i}+1}, \ldots A_{\bar{j}-1}$ são facilmente determinados usando a árvore Cartesiana do vetor de mínimos de todos os subvetores A_k, $0 \leq k \leq p-1$. Isto se reduz a um problema de mínimo intervalar restrito a um vetor de p valores. Assim, precisamos de uma estrutura de dados para responder a estas consultas em tempo constante. Como o vetor de mínimos contém apenas p valores, esta estrutura de dados pode ser obtida pelo Algoritmo *Mínimo_Intervalar(Alon e Schieber)*.

A dificuldade do Caso 2(b) é que não podemos construir a árvore-PS explicitamente em cada processador como descrito na seção 3.2., pois para isto seria necessário uma memória de tamanho $O(p \log p)$, maior do que o fixado pelo modelo CGM, que é $O(n/p)$, com $n/p \geq p$. Para contornar esta dificuldade, armazenamos apenas informações parciais na árvore-PS em cada processador. Construímos vetores P' e S' de $\log p + 1$ posições em cada processador, como descrito a seguir.

Vamos descrever esta construção para um processador i, com $0 \leq i \leq p-1$. Seja b_i o valor do mínimo do subvetor A_i. Seja v um nó qualquer da árvore T tal que a subárvore com raiz v tem b_i como folha, e seja d_v a **profundidade** de v em T, isto é, o comprimento do caminho da raiz até v, como definido em [1]; e seja l_v o *nível* de v, que é a altura da árvore menos a profundidade de v, como definido em [1] ($l_v = \log p - d_v$, pois a árvore tem altura $\log p$). O vetor P' (respectivamente, S') contém na posição l_v o valor do vetor P_u (respectivamente, S_v), do nível l_v de T, na posição correspondente à folha b_i. Em outras palavras, temos $P'[l_v] = P_v[i \bmod 2^{l_v}]$.

A Figura 3.3 ilustra a correspondência entre os vetores P' e S' armazenados em cada processador e a árvore-PS construída pelo algoritmo *Mínimo_Intervalar(Alon e Schieber)*. Na Figura 3.3(b), as posições em destaque nos vetores S em cada nível da árvore correspondem ao mínimo sufixo de $b_0 = 3$ em cada nível. Desta forma, obtemos o vetor S' no processador 0 (Figura 3.3(c)).

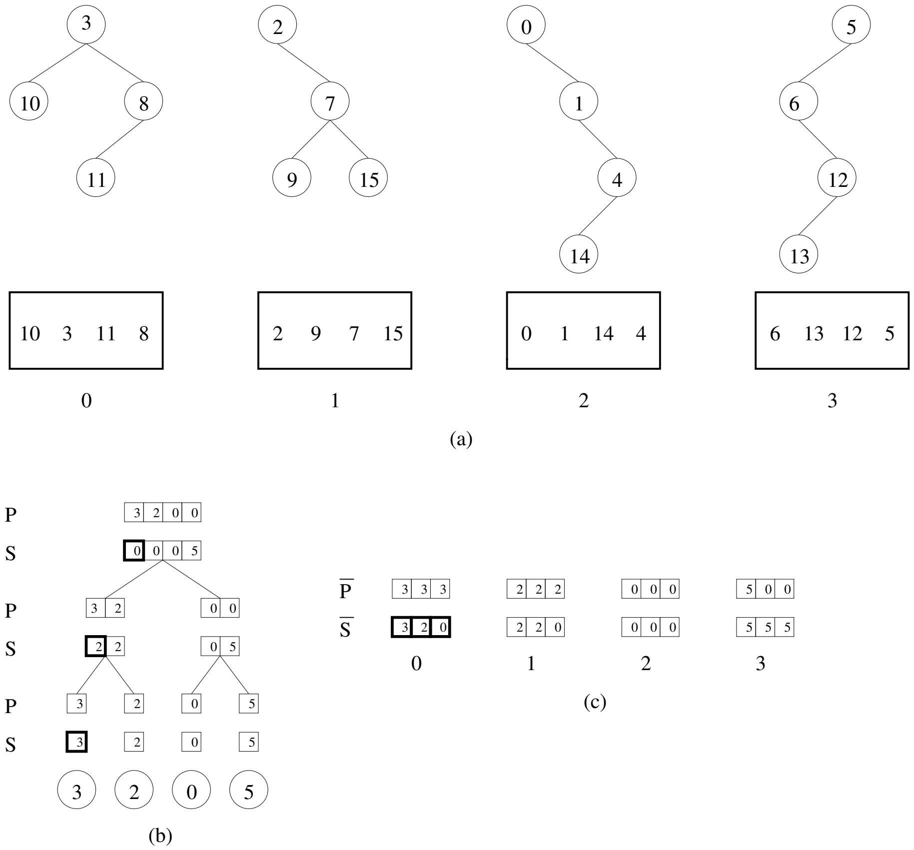


Figura 3.3: Execução do Algoritmo *Mínimo_Intervalar(CGM)* usando o vetor (10, 3, 11, 8, 2, 9, 7, 15, 0, 1, 14, 4, 6, 13, 12, 5). (a) Os dados distribuídos nos processadores e as árvores Cartesianas correspondentes. (b) Árvores-PS construídas pelo Algoritmo *Mínimo_Intervalar(Alon e Schieber)* para o vetor $(3, 2, 0, 5)$ de mínimos dos processadores. (c) Vetores P' e S' construídos pelo passo 3 do Algoritmo *Mínimo_Intervalar(CGM)* correspondente aos vetores P e S de T.

A seguir, damos uma descrição em alto nível do algoritmo. Cada processador recebe n/p posições contíguas do vetor A, dividido em subvetores $A_0, A_1, \ldots, A_{p-1}$.

Algoritmo 18 Mínimo Intervalar(CGM)
Entrada: o vetor $A = (a_0, a_1, \ldots, a_{n-1})$ com n números reais.
Saída: uma estrutura de dados que responde a consultas $MIN(i, j)$ em tempo constante.

(1) Cada processador i executa seqüencialmente o Algoritmo *Mínimo_Intervalar(Gabow et al).*

(2) Comentário: Cada processador constrói um vetor $B = (b_i)$ de tamanho p, contendo os mínimos dos subvetores armazenados em cada processador.
(2.1) Cada processador i calcula $b_i = \min A_i = \min\{a_{i(n/p)}, \ldots, a_{(i+1)(n/p)-1}\}$.
(2.2) Cada processador i envia b_i aos outros processadores.
(2.3) Cada processador i coloca em b_k o valor recebido do processador k, $k \in \{0, \ldots, p-1\} \setminus \{i\}$.

(3) Cada processador i executa os algoritmos *Constrói_P'* and *Constrói_S'* (ver a seguir).

— Fim do Algoritmo —

Dado um vetor B, o seguinte algoritmo constrói o vetor P' no processador i, para $0 \leq i \leq p-1$. Este algoritmo constrói P' em tempo $O(p)$ $(= O(n/p))$ usando somente dados locais. A construção do vetor S' é feita de maneira simétrica, considerando B na ordem inversa.

Algoritmo 19 Constrói_P'
Entrada: o vetor $B = (b_0, b_1, \ldots, b_{p-1})$ com p números reais.
Saída: o vetor P' of $\log p + 1$ posições.

(1) $P'[0] \leftarrow b_i$

(2) $aponta \leftarrow i$

(3) $inordem \leftarrow 2 * i + 1$

(4) **para** $k \leftarrow 1$ **até** $\log p$ **faça**
  (4.1) $P'[k] \leftarrow P'[k-1]$
  (4.2) **se** $\lfloor inordem/2^k \rfloor$ é ímpar **então**
    (4.2.1) **para** $l \leftarrow 1$ **até** 2^{k-1} **faça**
      (4.2.1.1) $aponta \leftarrow aponta - 1$
      (4.2.1.2) **se** $P'[k] > B[aponta]$ **então**
        (4.2.1.2.1) $P'[k] \leftarrow B[aponta]$

— Fim do Algoritmo —

Para simplificar a prova de corretude deste procedimento, consideramos p como sendo uma potência de 2 e que, em cada processador i, o vetor B contém as folhas de uma árvore binária completa, como na descrição do Algoritmo *Mínimo_Intervalar(Alon e Schieber)*. Não armazenamos inteiramente o vetor em cada nó interno desta árvore, mas somente as $\log p + 1$ posições do vetor de mínimo prefixo em cada nível correspondendo à posição i do vetor B.

O valor da variável $inordem$ do passo 3 é o valor do número *in-ordem* da folha contendo b_i, obtido em uma travessia *in-ordem* nos nós da árvore. Pode ser visto facilmente que estes valores, da esquerda para a direita, são números ímpares no intervalo $[1, 2p-1]$.

Teorema 7 *O Procedimento Constrói_P' corretamente calcula o vetor P', para cada processador i, $0 \leq i \leq p-1$.*

Prova. Para um processador i, $0 \leq i \leq p-1$, provamos o seguinte invariante em cada iteração do laço do passo 4:

Em cada iteração k, seja T_k a subárvore que contém b_i e tem raiz no nível k. $P'[k]$ armazena a posição i do vetor de mínimo prefixo do subvetor de B correspondente às folhas da subárvore T_k e a variável $aponta$ contém o índice, em B, da folha de T_k mais à esquerda.

Inicialmente, provamos que o invariante vale antes da primeira iteração. Neste caso, $k = 0$, $P'[k] = b_i$ e $aponta = i$ são válidos, pois a subárvore T_k é formada apenas pela folha que contém b_i e está na posição i de B.

Vamos supor que o invariante seja válido imediatamente antes do início de uma iteração k e vamos mostrar que permanece válido no fim desta iteração.

Na iteração k, seja C_k o subvetor de B cujos elementos são as folhas da subárvore T_k com raiz v. T_k contém b_i e v está no nível k.

Analisamos dois casos:

1. $\lfloor inordem/2^k \rfloor$ é par.
   Como estamos considerando árvores binárias completas, o valor par de $\lfloor inordem/2^k \rfloor$ significa que a folha correspondente a b_i está na subárvore esquerda de v.
   Neste caso, os elementos à esquerda de b_i em C_k são os mesmos elementos à esquerda de b_i em C_{k-1}. Assim, $P'[k] = P'[k-1]$. Além disso, a folha mais à esquerda de T_k é também a folha mais à esquerda da árvore T_{k-1} que contém b_i. Assim, o valor da variável $aponta$ não muda.
   No Procedimento *Constrói_P'*, o passo 4.1 faz a atualização $P'[k] \leftarrow P'[k-1]$, e este valor de $P'[k]$ não é alterado até o final da iteração. O valor da variável $aponta$ também não muda.
   Assim, neste caso, o invariante permanece válido no final da iteração k.
2. $\lfloor inordem/2^k \rfloor$ é ímpar.
   Neste caso, a folha correspondente a b_i está na subárvore direita de raiz v.
   O valor armazenado em $P'[k-1]$ corresponde ao mínimo prefixo do subvetor C_{k-1} do vetor B. A variável $aponta$ armazena o índice do vetor B, correspondendo à folha mais à esquerda da subárvore cujas folhas são os elementos de C_{k-1}. Isto é, esta variável aponta para a primeira posição de C_{k-1}.
   O subvetor C_k, considerado na iteração k, corresponde às folhas da subárvore T_k. Assim, este subvetor é formado pelas folhas das subárvore esquerda e direita de v. (As folhas da subárvore direita correspondem aos elementos do subvetor C_{k-1}.)
   Como este novo subvetor C_k apresenta valores à esquerda de b_i que não estavam envolvidos na determinação de $P'[k-1]$, o valor de $P'[k]$ será o mínimo entre $P'[k-1]$ e os valores correspondentes às folhas da subárvore esquerda de v.
   O laço **para** do passo 4.2 examina estas posições, comparando os valores com $P'[k]$ (inicialmente igual a $P'[k-1]$) e atualizando-o quando necessário. Isto é feito usando a variável $aponta$, que varre estas posições, e, ao final, aponta para a primeira posição do vetor C_k.
   Assim, ao final desta iteração, $P'[k]$ conterá o valor correto da posição i do vetor de mínimo prefixo do vetor B, e a variável $aponta$ conterá o índice, em B, do primeiro elemento de C_k que corresponde à folha mais à esquerda de T_k.

Na última iteração, $C_k = B$ e $P'[\log p]$ contém o valor da posição i do vetor mínimo prefixo de B, e a variável $aponta$ contém o índice do primeiro elemento de B.

Assim, o Procedimento *Constrói_P'* corretamente calcula o vetor P'.

□

Para a determinação de S', temos um teorema e prova similares.

Lema 6 *A execução do Procedimento Constrói_P' em cada processador leva tempo seqüencial $O(n/p)$.*

Prova. O número máximo de iterações é $\log p$. Em cada iteração, o valor de $aponta$ é decrementado ou permanece o mesmo. O pior caso é quando $i = p - 1$. Neste caso, em cada iteração, o valor da variável $aponta$ é atualizado. Como o número de elementos de B é p, o algoritmo atualiza o valor de $aponta$ no máximo p vezes. Assim, o Procedimento *Constrói_P'* é executado em tempo seqüencial $O(p) = O(n/p)$.

□

O teorema a seguir resume os resultados obtidos.

Teorema 8 *O algoritmo Mínimo_Intervalar(CGM) resolve o problema de mínimo intervalar em tempo $O(n/p)$ usando $O(1)$ rodadas de comunicação e memória $O(n/p)$.*

Prova. O passo 1 é executado em tempo seqüencial $O(n/p)$, não necessita de comunicação e utiliza memória $O(n/p)$. O passo 2 é executado em tempo seqüencial $O(n/p)$, utiliza espaço $O(p)$ e necessita de uma rodada de comunicação onde é enviado um valor e são recebidos $p - 1$ dados. Pelo teorema 6, o passo 3 é executado em tempo seqüencial $O(n/p)$, não precisa de comunicação e utiliza memória $O(p)$. Portanto, o Algoritmo *Mínimo_Intervalar(CGM)* resolve o problema de mínimo intervalar em tempo seqüencial $O(n/p)$, usando $O(1)$ rodadas de comunicação, onde são trocados no máximo $O(p)$ dados e utilizando memória local $O(n/p)$.

□

3.4. Processamento de Consultas

Nesta seção, mostraremos como usar a saída do Algoritmo *Mínimo_Intervalar(CGM)* para responder a consultas $O(n/p)$ em tempo constante. Uma consulta $MIN(i, j)$ é determinada como se segue. Assumimos que i e j são conhecidos por todos os processadores e o resultado será dado pelo processador 0. Se a_i e a_j estão no mesmo processador, então $MIN(i, j)$ pode ser determinado pelo passo 1 do Algoritmo *Mínimo_Intervalar(CGM)*. Caso contrário, suponha que a_i e a_j estão em processadores distintos $\bar{i}$ e $\bar{j}$, respectivamente, com $\bar{i} < \bar{j}$. Seja $direita(\bar{i})$, o índice em A do elemento mais à direita no vetor $A_{\bar{i}}$, e $esquerda(\bar{j})$ o elemento mais à esquerda no vetor $A_{\bar{j}}$. Calculamos $MIN(i, direita(\bar{i}))$ e $MIN(esquerda(\bar{j}), j)$, usando o passo 1. Temos, então, dois casos:

1. Se $\bar{j} = \bar{i}+1$ então $MIN(i, j) = \min\{MIN(i, direita(\bar{i}), MIN(esquerda(\bar{j}), j)\}$.
2. Se $\bar{i} + 1 < \bar{j}$, então calculamos $MIN(direita(\bar{i}) + 1, esquerda(\bar{j}) - 1)$, usando o passo 3 do algoritmo. Notemos que $MIN(direita(\bar{i}) + 1, esquerda(\bar{j}) - 1)$ corresponde a $\min\{b_{\bar{i}+1}, \ldots, b_{\bar{j}-1}\}$. Assim, $MIN(i, j) = \min\{MIN(i, direita(\bar{i})), MIN(direita(\bar{i}) + 1, esquerda(\bar{j}) - 1), MIN(esquerda(\bar{j}), j)\}$.

O valor de $MIN(direita(\bar{i}) + 1, esquerda(\bar{j}) - 1)$ é obtido usando o passo 3, como descrito a seguir. Cada processador calcula $w = \text{LCA}(b_{\bar{i}+1}, b_{\bar{j}-1})$ em tempo constante, então determina o nível l_w e determina v e u, os filhos esquerdo e direito de w, respectivamente. O processador $\bar{i} + 1$ calcula $S'[l_w - 1]$ e envia este valor para o processador 0. O processador $\bar{j} - 1$ calcula $P'[l_w - 1]$ e envia este valor para o processador 0. O processador 0, finalmente, calcula o mínimo entre os mínimos recebidos. Em ambos os casos, o processador 0 recebe os mínimos dos processadores $\bar{i}$ e $\bar{j}$.

Notemos, finalmente, que a corretude do algoritmo *Mínimo_Intervalar(CGM)* resulta das observações apresentadas nesta seção.

Apêndice A

MPI

1. Introdução

O MPI não é uma nova linguagem de programação, é uma coleção de funções ou macros, ou seja uma *biblioteca* que pode ser usada em programas C, C++ ou Fortran.

O LAM é um *deamon* baseado na implementação MPI. Inicialmente o programa *lamboot* distribui os *deamons* da LAM baseado numa lista de máquinas fornecida pelo usuário. Esses *deamons* permanecem inativos na lista de máquinas remotas até que eles recebam uma mensagem para carrregar o executável do MPI para iniciar a execução. Só podemos executar programas MPI enquanto os *deamons* estiverem atuando em *background*.

2. Programando com MPI

Os programas que utilizaremos neste texto serão do modelo SPMD. Neste modelo, cada processo *roda* o mesmo programa executável. Contudo, os processos executam diferentes instruções na medida em que seguem diferentes ramificações do programa: as ramificações são determinadas pelo número do processo.

Todo programa MPI deve incluir a diretiva de pré-processador

```
#include <mpi.h>
```

Isto inclui as declarações e definições necessárias para compilar um programa MPI. O MPI utiliza um esquema consistente para os identificadores definidos para o MPI. Todos identificadores iniciam com os caracteres `MPI_`. Os caracteres restantes da maior parte das constantes MPI são letras maiúsculas. O primeiro caracter do restante do nome de cada função MPI é uma letra maiúscula seguida por letras minúsculas (p.e., `MPI_Init`).

2.1. O Mundo do MPI

As tarefas são representadas por um *rank* (inteiro) e os *ranks* são numerados de $0, 1, 2, \ldots, N - 1$. O comunicador `MPI_COMM_WORLD` significa todas as tarefas na aplicação MPI e providencia a informação necessária para efetuar troca de mensagens.

2.2. Iniciando e Terminando o MPI

Antes que qualquer outra função do MPI seja utilizada, nosso programa deve ter uma chamada para

`MPI_Init(&argc, &argv);`.

A finalização de um programa MPI possui uma chamada para a função

`MPI_Finalize();`.

2.3. Identificação das Tarefas

Tipicamente, uma tarefa em uma aplicação paralela necessita saber quem ele é (seu *rank*) e quantas outras tarefas existem. Uma tarefa encontra seu próprio *rank* chamando a função `MPI_Comm_rank()`:

```
int meurank;
MPI_Comm_rank(MPI_COMM_WORLD, &meurank);
```

O número total de tarefas é retornado pela função `MPI_Comm_size()`:

```
int nprocs;
MPI_Comm_size(MPI_COMM_WORLD, &nprocs);
```

2.4. Enviando Mensagens

Uma mensagem é um vetor de elementos de um determinado tipo de dados. O MPI trabalha com todos os tipos de dados básicos e permite a construção de tipos de dados mais elaborados.

Uma mensagem é enviada para uma tarefa específica e é marcada com uma *tag* (valor inteiro) especificado pelo usuário. As *Tags* são utilisadas para distinguir entre diferentes tipos de mensagens que uma tarefa pode enviar ou receber.

```
MPI_Send(buffer, count, datatype, destination, tag,
MPI_COMM_WORLD);
```

2.5. Recebendo Mensagens

Uma tarefa recebendo uma mensagem especifica a *tag* e o *rank* do processo que está enviando. As *tags* genéricas `MPI_ANY_TAG` e `MPI_ANY_SOURCE` podem ser utilizadas opcionalmente para receber uma mensagem de qualquer *tag* e de qualquer tarefa que esteja enviando.

```
MPI_Recv(buffer, maxcount, datatype, source, tag,
MPI_COMM_WORLD, &status);
```

Informação a respeito da mensagem recebida é fornecida pela variável `status`.

3. Observações

Devido a limitação de espaço, exemplos de programas MPI e detalhes de instalação e utilização do MPI serão fornecidas durante o curso.

Mais detalhes podem ser encontrados em Group et al [14], Snir et al [22] e Pacheco [20]

Apêndice B

PVM (Parallel Virtual Machine)

1. Introdução

O PVM é um conjunto integrado de ferramentas de *software* e bibliotecas que permite que uma rede de computadores heterogêneos possa ser utilizada para a realização de computação paralela ou concorrente. Essas máquinas formarão uma única **máquina virtual**. Uma **máquina virtual** pode ser composta de estações de trabalho, (ou servidores) de vários tipos e em lugares físicos diferentes.

O PVM é independente de uma linguagem particular. Ele foi escrito em C e as aplicações para o PVM podem ser compostas por um número qualquer de processos distintos, escritos em uma mistura de C, C++ e Fortran. Bibliotecas para essas linguagens são incluídas na distribuição. O PVM também permite o uso de outras linguagens, como o Lisp. O modelo de computação do PVM é baseado na idéia de que uma aplicação consiste de várias tarefas. Cada tarefa é responsável por uma parte do problema. O PVM suporta tanto o paralelismo funcional quanto o paralelismo de dados.

O PVM versão 3.4 é instalado no diretório `$PVM_ROOT` do sistema. Quando a rede faz uso de um sistema **nfs**, temmos que `PVM_ROOT=/usr/local/pvm3`. Após instalar o PVM, o programador deve providenciar que o compilador encontre os arquivos que estão em `$PVM_ROOT/include` e em `$PVM_ROOT/lib` tais como `pvm3.h` e `libpvm3.a`. Para se executar um programa que utilize o PVM, o usuário necessita "construir"a máquina paralela virtual sobre a qual este programa vai executar.

O sistema tem portabilidade para várias arquiteturas, incluindo estações de trabalho, multiprocessadore e supercomputadores. A plataforma mais utilizada para o PVM é o UNIX.

Podemos criar uma máquina paralela interativamente via console. Por exemplo, para obter uma *console* PVM, faz-se:

```
$ pvm
pvm>
```

O comando `pvm` inicia um processo de controle PVM em *background* (o *daemon* PVM) na estação de trabalho onde a *console* foi inicializada. Uma **máquina paralela virtual** é composta de processadores executando o *daemon* `pvmd` do PVM. É este *daemon* que gerencia toda comunicação entre os processos, bem como a criação e finalização destes processos.

Pela *console*, o programador poderá criar, modificar ou desativar uma máquina paralela virtual, bem como executar programas PVM e finaliza-los, por exemplo, em caso seja necessário. A *console* PVM é uma espécie de *shell* que aceita um conjunto definido de comandos.

Ao utilizar o PVM, os processadores são identificados por um nome ou por um número. O comando `add` da *console* inclui uma estações de trabalho na máquina virtual paralela. Para adicionar uma estações de trabalho, a partir da console, temos o comonado:

```
pvm> add nome_da_estação_de_trabalho
```

Pode-se adicionar um número indefinido de estações de trabalhos, desde que a mesma tenha PVM instalado (ou que o **nfs** compratilhe) e que o programador tenha a devida autorização para utilizar a máquina a ser adicionada. Para verificar a configuração da máquina virtual paralela corrente, utilizamos o comando `conf`:

```
pvm> conf
```

A remoção de estações de trabalhos da máquina virtual é feita através do comando `delete`.

```
pvm> delete nome_da_estação_de_trabalho
```

Para finalizar a *console*, dois comandos podem ser utilizados: `halt` que encerra a máquina virtual paralela corrente e o `quit` que finaliza a *console* e mantém a máquina virtual: os *daemons* ativos continuam sendo executados e programas que utilizam o PVM podem ser executado. Se, por acaso, um programa que utilize o PVM ficar sem ação, o comando `reset` pode ser utilizado para reinicializar a máquina virtual paralela, desativando todos os processos criados.

O PVM possui as seguintes característicasDentro de suas características:

- Configuração das estações de trabalho pelo usuário: as tarefas de aplicação computacional executam em um conjunto de máquinas que são selecionadas pelo usuário para um dado programa PVM. Máquinas com apenas um processador ou com vários processadores poderão fazer parte desse conjunto.
- Acesso transparente com relação ao *hardware*: os programas aplicativos podem ter uma visão do ambiente do *hardware* como coleções de elementos sem atributos, ou podem explorar as capacidades específicas de uma máquina no conjunto de estações de trabalhos, simplesmente selecionando quais tarefas serão executadas em determinada estação de trabalho.
- Computação baseada em processos: a unidade de paralelismo em PVM é chamada de **tarefa** (*task*), uma sequência independente de **threads** de controle, que alternam entre computação e comunicação. Uma **thread** é uma unidade básica de utilização da CPU, composta de um valor para os registradores da CPU e uma área de memória para pilha. Uma *thread* compartilha com suas *threads* "irmãs"a seção de código e de dados na memória e, recursos do sistema operacional. O conjunto das *threads* "irmãs" formam uma **tarefa**.
- Modelo explícito de troca de mensagens: coleções de tarefas computacionais, cada uma executando uma parte da aplicação usando decomposição de dados, funcional ou híbrida, cooperando através do envio e recebimento de mensagens ex-

plicitamente.

- Suporte à heterogeneidade: o PVM suporta heterogeneidade em termos de máquinas, redes e aplicações. Como considerações acerca de troca de mensagens, o PVM permite que mensagens contenham mais que um tipo de dado para ser trocado entre máquinas que possuem diferentes representações de dados.

2. O arquivo Makefile.aimk

O PVM provê um `make` independente, chamado `aimk`, que automaticamente direciona os arquivos executáveis para o diretório `$PVM_ARCH` e adiciona as bibliotecas necessárias à aplicação. O utilitário `make` determina quais peças de um projeto precisam ser recompiladas.

Basicamente um `makefile` simples consiste de regras com a seguinte forma:

```
target...:  dependencias...
comando
...
...
```

`target` é usualmente o nome de um arquivo que é gerado pelo programa, como por exemplo, o arquivo executável ou arquivos objeto.

`dependencia` é o arquivo que é usado para criação do `target`. Exemplo: `arquivo.cpp`.

`comando` é uma ação que o `make` executa. Uma **regra** pode ter mais que um `comando`, cada um em uma linha.

Uma **regra** explica como e quando será compilado o arquivo. `Make` executa os comandos sobre as dependências, para criar ou atualizar o `target`.

3. Troca de Mensagens no PVM

O envio de uma mensagem no PVM é composta de três passos:

1. Um *buffer* de envio deve ser inicializado com uma chamada à função `pvm_initsend(int encoding)` ou `pvm_mkbuf(int encoding)`.
2. A mensagem deve ser empacotada dentro desse *buffer*, usando qualquer número e combinação das funções `pvm_pk*(...)`.
3. A mensagem completa é enviada para outro processo pela chamada da função `pvm_send(int tid, int msgtag)` ou usando o envio múltiplo através da função `pvm_mcast(int *tids, int ntask, int msgtag)`.

Uma mensagem pode ser recebida pela chamada de funções de recebimento bloqueantes ou não bloqueantes e, então, é efetuado um desempacotamento de cada um dos itens do *buffer* de recebimento. As funções de recebimento podem ser direcionadas para aceitar qualquer mensagem; ou qualquer mensagem de uma fonte específica; ou qualquer mensagem com um *tag* de mensagem específica; ou somente mensagens com um dado *tag* de mensagem de uma dada origem (`pvm_recv(int tid, int msgtag)`, `pvm_nrecv(int tid, int msgtag)`, `pvm_trecv(int tid, int msgtag, struct timeval *tmount)`.

```
# Makefile.aimk para programas PVM

SDIR = $(HOME)/programs
XDIR = $(SDIR)/$(PVM_ARCH)/bin

INCLUDE_DIR= /usr/include
LIB_DIR= /usr/lib
cc = gcc
CC = g++
OPTIONS = -g -bnoquiet
PVMIDIR = ($PVM_ROOT)/include
PVMLIB = -lpvm3
CFLAGS = $(OPTIONS) -I$(INCLUDE_DIR) -I$(PVMIDIR)
$(ARCHCFLAGS)

LIBS = $(PVMLIB) $(ARCHLIB)
GLIBS = -lgpvm3
LFLAGS = -L$(LIB_DIR) -L$(PVM_ROOT)/lib/$(PVM_ARCH)

hello:  $(SDIR)/hello.c $(XDIR)
$(cc) $(CFLAGS) -o hello $(SDIR)/hello.c $(LFLAGS) $(LIBS)
$(GLIBS)
mv hello $(XDIR)
hello_1:  $(SDIR)/hello_1.c $(XDIR)
$(CC) $(CFLAGS) -o hello_1 $(SDIR)/hello_1.c $(LFLAGS)
$(LIBS) $(GLIBS)
mv hello_1 $(XDIR)
```

Figura B.1: Exemplo do arquivo Makefile.aimk

3.1. *Buffers* de mensagens

Se o usuário está usando somente um único *buffer* de envio (e esse é o caso mais típico), então `pvm_initsend(...)` é a única função necessária. Ela é chamada antes do empacotamento de uma nova mensagem dentro do *buffer*. A fuinção `pvm_initsend(...)` esvazia o *buffer* de envio e cria um novo para empacotar a nova mensagem.

Outras funções somente são necessárias, se o usuário deseja gerenciar múltiplos *buffers* de mensagens na sua aplicação. No PVM há sempre um *buffer* de envio ativo e um *buffer* de recebimento ativo por processo, em qualquer momento. O desenvolvedor pode criar um número qualquer de *buffers* de mensagens e escolher entre eles para o empacotamento e o envio de dados. O empacotamento, envio, recebimento e desempacotamento afetam apenas o *buffer* de mensagens ativo.

3.2. Empacotando dados

As várias rotinas disponíveis no PVM empacotam um ponteiro para um tipo de dados dentro do `buffer` de envio ativo. Elas podem ser chamadas várias vezes para empacotar dados dentro de uma mensagem simples. Assim, uma mensagem pode conter vários ponteiros, cada um com um tipo de dado diferente. Estruturas em C podem ser passadas pelo empacotamento individual de elementos. Não há limites para a complexidade dos pacotes de mensagens, mas uma aplicação deverá desempacotar as mensagens exatamente como foram empacotadas.

Exemplo de algumas funções: `pvm_pkbyte(char *cp, int cnt, int std)`,
`pvm_pkdouble(double *dp, int cnt, int std)`,
`pvm_pkfloat(float *fp, int cnt, int std)`,
`pvm_pkint(int *np, int cnt, int std)`, `pvm_pkstr(char *cp)`.

3.3. Enviando e recebendo dados

As funções `pvm_send(int tid, int msgtag)` e `pvm_mcast(int *tids, int ntask, int msgtag)`, usadas para o envio de mensagens, possuem como parâmetros a identificação do processo que vai receber a mensagem - `tid` - e um identificador - `msgtag` - para a mensagem a ser enviada. A função `pvm_send(...)` nomeia a mensagem com o identificador `msgtag` e a envia imediatamente para o processo de identificação `tid`. A função `pvm_mcast(...)` nomeia a mensagem com o identificador inteiro `msgtag` e envia a mensagem para todas as tarefas especificadas no vetor de inteiros `tids`, exceto para ela mesma.

Como vimos anteriormente, o PVM contém vários métodos para recebimento de mensagens de uma tarefa. A função mais usada é `pvm_recv(int tid, int msgtag)`, que espera (é bloqueante) enquanto uma mensagem com o nome `msgtag` venha do processo identificado por `tid`. Então, coloca a mensagem no *buffer* de recebimento ativo que foi criado. O PVM também suporta uma versão de recebimento com *timeout*. Considere o caso onde a mensagem nunca chegará (por causa de um erro ou falha); a rotina `pvm_recv(...)` ficará bloqueada para sempre. Para tratar esses casos, o usuário pode projetar o seu programa de tal forma que após esperar por determinado período de tempo, a recebimento da mensagem seja suspenso. A função `pvm_trecv(int tid, int msgtag, struct timeval *tmout)` permite ao usuário especificar um período de tempo. Se o período de tempo for muito grande, `pvm_trecv(...)` funcionará como `pvm_recv(...)`. Se o período de tempo for igual a zero, `pvm_trecv(...)` funcionará como o `pvm_nrecv(int tid, int msgtag)`, que é a função de recebimento não bloqueante.

3.4. Desempacotando dados

As funções para desempacotamento de dados presentes no PVM agem sobre os dados do *buffer* de recebimento ativo. Na aplicação do usuário essas funções devem corresponder às rotinas usadas no empacotamento.

Exemplo de algumas funções: `pvm_upkbyte(char *cp, int cnt, int std)`,
`pvm_upkdouble(double *dp, int cnt, int std)`,
`pvm_upkfloat(float *fp, int cnt, int std)`,
`pvm_upkint(int *np, int cnt, int std)`, `pvm_pkstr(char *cp)`.

3.5. Grupos Dinâmicos de Processos

Um **grupo dinâmico de processos** pode ser definido como uma estrutura, dentro da qual processos são inseridos ou removidos. Um processo PVM pode pertencer a múltiplos grupos e grupos podem ser modificados dinamicamente em qualquer instante de execução. As funções de grupos de processos dinâmicos são construídas sobre o núcleo das rotinas PVM. O PVM não executa as funções de grupos. Essa tarefa é realizada por um servidor de grupo que é automaticamente iniciado quando a primeira função de grupo é chamada.

Mantendo a filosofia PVM, as funções de grupo são designadas para serem muito gerais e transparentes para o usuário, mesmo em custo e eficiência. Qualquer tarefa PVM pode abandonar ou entrar em qualquer grupo, em qualquer tempo, sem ter que informar às outras tarefas do mesmo grupo. Tarefas podem enviar mensagens por difusão para grupos dos quais a tarefa emissora não faça parte. Qualquer processo, pode chamar a qualquer instante uma rotina de manipulação de grupos, exceto as funções `pvm_lvgroup(char *group)` e `pvm_barrier(char *group, int count)` que só fazem sentido para processos pertencentes a um determinado grupo.

Situações anômalas podem ocorrer quando uma tarefa realiza o envio de uma mensagem a todas as tarefas do grupo e ao mesmo tempo uma nova tarefa entra no grupo. Esta nova tarefa pode não receber a mensagem. Por outro lado, uma tarefa pode abandonar um grupo durante um envio de mensagem. Uma cópia desta mensagem pode ser despachada sem ter uma tarefa para recebê-la.

Exemplos de funções de grupo: `pvm_joingroup(char *group)`, `pvm_gettid(int *group, int inum)`, `pvm_lvgroup(char *group)`, `pvm_gsize(char *group)`.

4. Principais Funções do PVM

Apresentamos a seguir, algumas funções do PVM bastante utilizadas, junto com uma breve descrição de cada uma:

`pvm_exit(void)`: informa ao `pvmd` (*daemon*) local que o processo está deixando o PVM. Essa função não finaliza o processo e deve ser chamada pôr todas as tarefas, principalmente pelas que não foram iniciadas com o `pvm_spawn(char *task, char **argv, int flag, char *where, int ntask, int *tids)`.

`pvm_initsend(int encoding)`: limpa o *buffer* de envio de mensagens *default* e especifica a codificação para as mensagens.

`pvm_mytid(void)`: retorna a identificação de uma tarefa.

`pvm_pkint(int *np, int cnt, int std)`: empacota e armazena no *buffer* de mensagem ativo um vetor de inteiros.

`pvm_recv(int tid, int msgtag)`: recebe uma mensagem (dados) através do *buffer*. É bloqueante.

`pvm_send(int tid, int msgtag)`: envia dados pelo buffer de mensagens ativo.

`pvm_spawn(char *task, char **argv, int flag, char *where, int ntask,int *tids)`: inicia uma ou mais tarefas no PVM, e o número de cópias de uma mesma tarefa pode ser maior que um para uma mesma estação de trabalho.

`pvm_upkint(int *np, int cnt, int std):`
desmpacota e armazena no *buffer* de recebimento de mensagens ativo um vetor de inteiros.

5. Como o PVM Trabalha

Os objetivos mais importantes do PVM são tolerância a falhas, escalabilidade, heterogeneidade e portabilidade. O PVM é capaz de detectar falhas na rede. Ele não recupera automaticamente uma aplicação depois de uma falha e interrupção de uma estação de trabalho, mas provê primitivas de notificação que permitem às aplicações de tolerância a falhas serem reintegradas. A máquina virtual é reconfigurada automaticamente.

Essa propriedade trabalha simultaneamente com tolerância a falhas: uma aplicação pode precisar adquirir mais recursos para poder continuar sendo executada, se uma determinada estação de trabalho falhar.

O PVM pode conectar computadores de diferentes tipos em uma única sessão. Ele executa com modificações mínimas em qualquer implementação do Unix ou um sistema operacional com facilidades comparáveis (*multicasting*, *networkable*).

5.1. Classes de arquitetura

O PVM atribui um nome para cada tipo de architetura para as máquinas nas quais está sendo executado. Isso é feito para distinguir entre máquinas que possuem executáveis diferentes, pôr causa de diferenças de *hadware* ou de sistemas operacionais.

Algumas vezes, máquinas com executáveis incompatíveis usam a mesma representação de dados binários.

5.2. Modelos de Mensagens

O *deamons* do PVM podem compor e enviar mensagens de tamanhos arbitrários, contendo diferentes tipos de dados. Os dados podem ser convertidos usando `XDR`, quando trocados entre estações de trabalhos com formatos de dados incompatíveis.

Mensagens são rotuladas (`tags`) e enviadas com o código definido pelo usuário e podem ser selecionadas para receber por endereços fonte ou `tags`. Quem envia a mensagem não espera por uma confirmação do receptor, mas continua assim que a mensagem for entregue pela rede e o *buffer* de mensagem possa ser seguramente apagado ou reutilizado. A ordem das mensagens é preservada; se várias mensagens forem enviadas, elas serão recebidas na mesma ordem. Primitivas de recebimento bloqueantes e não bloqueantes são fornecidas, então uma tarefa pode esperar por uma mensagem sem, necessariamente, consumir tempo de processador.

6. Observações

Devido a limitação de espaço, exemplos de programas PVM e detalhes de instalação e utilização do PVM serão fornecidas durante o curso.

Mais detalhes podem ser encontrados em Geist et al [11].

Referências

[1] A. V. Aho, J. E. Hopcroft, and J. D. Ullman. *The Design and Analysis of Computer Algorithms*. Addison-Wesley Publishing Company, 1975.

[2] N. Alon and Schieber. Optimal preprocessing for answering on-line product queries. Technical Report TR 71/87, The Moise and Frida Eskenasy Inst. of Computer Science, Tel Aviv University, 1987.

[3] E. N. Cáceres, F. Dehne, A. Ferreira, P. Flocchini, I. Rieping, A. Roncato, N. Santoro, and S. W. Song. Efficient parallel graph algorithms for coarse grained multicomputers and bsp. In P. Degano, R. Gorrieri, and A. Marchetti-Spaccamela, editors, *Proceedings ICALP'97 - 24th International Colloquium on Automata, Languages, and Programming*, volume 1256 of *Lecture Notes in Computer Science*, pages 390–400. Springer Verlag, 1997.

[4] R. Cole. Parallel merge sort. *SIAM J. Computing*, 17(4):770–785, 1988.

[5] R. Cole and U. Vishkin. Approximate parallel scheduling, part i: The basic technique with applications to optimal parallel list ranking in logaritmic time. *SIAM J. of Computing*, 17(1):128–142, 1988.

[6] F. Dehne, A. Fabri, and C. Kenyon. Scalable and architecture independent parallel geometric algorithms with high probability optimal time. In *6th IEEE Symposium on Parallel and Distributed Processing*, pages 586–593, 1994.

[7] F. Dehne, A. Fabri, and A. Rau-Chaplin. Scalable parallel geometric algorithms for coarse grained multicomputers. In *Proc. ACM 9th Annual Computational Geometry*, pages 298–307, 1993.

[8] F. Dehne (Ed.). Coarse grained parallel algorithms. *Special Issue of Algorithmica*, 24(3/4):173–176, 1999.

[9] A. Ferreira. Discrete computing with pc clusters: an algorithmc approach. Tutorial, International School on Advanced Algorithmic Techniques for Parallel Computationwith applications, September 1999.

[10] H. N. Gabow, J. Bentley, and R. E. Tarjan. Scaling and related techniques for geometry problems. In *Proc. Sixteenth Annual ACM Symposium on Theory of Computing*, pages 135–143. ACM Press, 1984.

[11] A. Geist, A. Beguelin, J. Dongarra, W. Jiang, R. Manchek, and V. Sunderman. *PVM Parallel Virtual Machine - A Users' Guide and Tutorial for Networked Parallel Computing*. MIT Press, 1994.

[12] M.T. Goodrich. Communication efficient parallel sorting. In *ACM Symposium on Theory of Computing (STOC)*, pages 0–0, 1996.

[13] S. Gotz. *Communication-Efficient Parallel Algoritms for Minimum Spanning-Tree Computation*. PhD thesis, Universitat-Gesamthochschule Parderborn, 1998.

[14] W. Gropp, E. Lusk, and A. Skjellum. *Using MPI - Portable Parallel Programming with Message-Passing Interface*. The MIT Press, second edition, 1999.

[15] J. JáJá. *An Introductionto Parallel Algorithms*. Addison Wesley, 1992.

[16] R. M. Karp and V. Ramachandran. *Handbook of Theoretical Computer Science*, volume A, chapter 17, pages 869–941. Elsevier/MIT Press, 1990.

[17] I. G. Lassous and J. Gustedt. List ranking on a coarse grained multiprocessor. Technical Report 3640, Institut National de Recherche en Informatique et en Automatique, 1999.

[18] I. G. Lassous and J. Gustedt. List ranking on pc clusters. Technical Report 3869, Institut National de Recherche en Informatique et en Automatique, 2000.

[19] H. Mongelli and S. W. Song. A range minima paralle algorithm for coarse grained multicomputers. In Jos Rolin at al., editor, *IPPS99/Irregular - Sixth International Workshop on Solving Irregularly Structured Problems in Parallel*, volume 1586 of *Lecture Notes in Computer Science*, pages 1075–1084, San Juan, Puerto Rico, April 1999. Springer Verlag.

[20] P. S. Pacheco. *Parallel Programming with MPI*. Morgan Kaufmann Publishers, Inc, 1997.

[21] J. H. Reif, editor. *Synthesis of parallel algorithms*. Morgan Kaufmann Publishers, 1993.

[22] M. Snir, S. Otto, S. Huss-Lederman, D. Walker, and J. Dongarra. *MPI - The Complete Reference Volume 1, The MPI Core*. The MIT Press, second edition, 1998.

[23] S. Song. Communication-efficient parallel algoritms for minimum spanning-tree computation. Tutorial, International School on Advanced Algorithmic Techniques for Parallel Computation with Applications, September 1999.

[24] R. E. Tarjan and U. Vishkin. Finding biconnected components and computing tree functions in logarithmic parallel time. *SIAM J. Computing*, 14(4):862–874, 1985.

[25] L. Valiant. A bridging model for parallel computation. *Communication of the ACM*, 33(8):103–111, 1990.