1

Modelos para a computação paralela

Alfredo Goldman[1] *(DCC-IME-USP, gold@ime.usp.br)*

Resumo:

O objetivo principal deste curso é a apresentação de como podem ser desenvolvidas aplicações paralelas. Para isto são necessários diversos tipos de modelos, tanto para representar máquinas paralelas, como para representar as aplicações.

Se considerarmos todos os detalhes de uma máquina paralela teremos ao menos dois problemas. Por um lado, certos detalhes são menos importantes do que outros, e quando todos são considerados, o problema de alocação das tarefas da aplicação pode ficar muito complexo. Além disto, se resolvermos um problema considerando todos os detalhes de uma máquina específica quando passarmos a outra máquina boa parte do trabalho seria perdido. Por outro lado, quando poucos detalhes são considerados, pode ser mais fácil resolver o problema de escalonamento, mas, não será possível prever com exatidão o tempo de execução do problema, e as caracteríscas da máquina não serão aproveitadas. Existe um compromisso claro entre o nível de detalhe, e a aproximação da realidade dos modelos.

Nas seções seguintes veremos diversos modelos conhecidos para a computação paralela, para ilustrar o funcionamento de cada um deles, também serão apresentados algoritmos para os mesmos. É interessante ressaltar que a maioria dos algoritmos depende fortemente do modelo.

[1]Professor Doutor do Departamento de Ciência da Computação da Universidade de São Paulo desde 1993. Mestre pelo mesmo departamento, e doutor pelo Instituto Nacional Politécnico de Grenoble, França. Suas principais áreas de interesse são Computação Paralela e Distribuída, Computação Móvel e Programação Extrema. Ocupa desde 2000, o cargo de diretor do Centro de Ensino da Computação do IME-USP.

Departamento de Ciência da Computação do IME - USP, rua do Matão 1010, 05508-900, Cidade Universitária, São Paulo, Brasil.

1.1. Modelos

Na computação tradicional, com um processador, existem dois modelos que são muito bem aceitos na literatura: o modelo de arquitetura de *Von Neuman* e o modelo de complexidade de *Turing* (modelo teórico). O primeiro apresenta uma visão simplificada de um computador, onde estão separados os seguintes componentes: CPU, unidade aritmética e memória. Veja uma representação na figura 1.1.

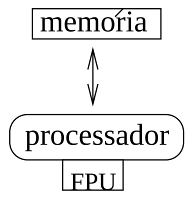

Figura 1.1: Arquitetura de Von Neuman.

Em uma máquina de *Von Neuman*, as instruções são executadas seqüencialmente em ciclos compostos dos passos abaixo:

1. Busca e decodificação das instruções;
2. Cálculo dos endereços dos operandos;
3. Busca dos operandos na memória;
4. Cálculo com os operandos;
5. Armazenamento dos resultados na memória.

É interessante ressaltar que nem todas as operações possuem todos os passos vistos. Apesar deste modelo não considerar vários detalhes de máquinas existentes, como por exemplo a existência de vários níveis de memória *cache* que oferecem diferentes velocidades de acesso a memória. Ele captura a essencial do funcionamento de um computador seqüencial, onde a CPU busca dados na memória e faz operações entre eles usando a unidade aritmética (*Floating Point Unit*).

Infelizmente não existe um modelo universal para computação paralela similar ao modelo de *Von Neuman* para a computação seqüencial. Um bom modelo deve ter as seguintes qualidades: ser ao mesmo tempo simples e realista, sendo estas duas qualidades antagônicas. Quanto mais um modelo é simples, logo abstrato, menos ele aproxima o comportamento de uma máquina real. Por outro lado, um modelo muito realista geralmente é de difícil utilização [2].

Uma possibilidade de divisão dos diversos modelos para computação paralela é a seguinte:

Modelos para a aplicação Servem para representar o paralelismo potencial de um algoritmo;

[2] Em uma conversa com um colega de trabalho há alguns anos constatei que boa parte dos "detalhes" que eu geralmente ignoro em uma máquina paralela eram parte importante do seu trabalho.

Modelos de máquina Descrevem as principais características da máquina paralela, como o fluxo de instruções e a comunicação entre os processadores;

Modelo de execução Detalham as formas de programação e seus paradigmas.

Apesar de esta classificação não ser um padrão na literatura especializada, ela separa em três blocos lógicos os diversos modelos para a computação paralela. Sendo estes separados em: algoritmo, máquina e suporte a execução. Além disto a fronteira entre os modelos de máquina e de execução não é trivial. Por exemplo, para o modelo sistólico, esta separação não é clara. Mas, modelos de mais alto nível como BSP, ou tarefas maleáveis, são claramente modelos de execução.

Desta forma, o desenvolvimento de uma aplicação paralela em uma máquina dada pode ser visto através das etapas seguintes:

1. Escolha de um modelo que representa a máquina;

2. Representação do paralelismo potencial de um algoritmo e a escolha de um modelo de execução coerente com o modelo da máquina;

3. Procurar um escalonamento.

Basicamente queremos reduzir o desenvolvimento de uma aplicação paralela a um problema de escalonamento[3] (*scheduling*). Em um problema de escalonamento existem dois tipos de entidades, os fornecedores e os usuários recursos. No nosso contexto, os fornecedores serão as máquinas, ou processadores de uma máquina paralela, e os usuários, as diversas tarefas que compõem a aplicação paralela. Resolver um problema de escalonamento consiste em atribuir para cada tarefa, uma máquina e um tempo de execução. Veremos mais detalhes em seguida.

Claramente, as etapas 2 e 3 são as principais. Na etapa 2, a escolha do modelo de execução depende da máquina dada, e da representação do paralelismo do algoritmo. Da mesma forma a escolha do nível de paralelismo está ligada ao modelo de execução. Após a escolha dos modelos, a procura por um escalonamento pode começar.

Neste texto nós apresentaremos as diferentes formas de representar uma aplicação, em seguida os modelos para computação paralela e os modelos de execução. Para exemplificar cada modelo, serão apresentados também alguns algoritmos.

1.2. Grafo de precedência

A representação de um algoritmo, ou aplicação, é feita principalmente de duas formas análogas, cada uma utiliza um grafo dirigido como estrutura. Uma aplicação pode ser representada por um grafo de fluxo de dados (*data-flow graph*), por um grafo de precedência (*precedence graph*), ou simplesmente por um grafo de dependência.

Em uma representação por um grafo de precedência, o algoritmo é dividido em partes, estas denominadas de tarefas. Cada tarefa consiste de um grupo de

[3]Este termo deve ser familiar a quase todos, no sentido de escalonamento de processos, ou tarefas, em um único processador

instruções. As tarefas são representadas como os vértices do grafo. A dependência entre as tarefas é dada pelas arestas. Um arco (u, v) de uma tarefa u a uma tarefa v implica que a tarefa v não pode ser executada sem os dados produzidos pela tarefa u.

Geralmente, a cada arco fica associado um peso correspondente à quantidade de dados a transmitir, e a cada vértice o tempo de cálculo da tarefa. No decorrer do texto, nós faremos referência a estes valores como os pesos dos arcos e vértices. Caso não sejam associados pesos aos arcos, então temos apenas um grafo de dependência.

Os grafos de fluxo de dados são construídos a partir da evolução dos dados. As relações de precedência são derivadas pelos nós onde passam os dados. Geralmente, os vértices correspondem ao cálculo, e os arcos aos acessos em leitura, ou escrita.

A partir da representação de um programa por grafo de fluxo de dados é possível efetuar a transformação para grafo de precedência, pois o primeiro fornece mais informações. Neste texto, nós veremos apenas grafos de precedência.

A priori, a duração das tarefas pode ser desconhecida, da mesma forma, o grafo de precedência pode também não ser conhecido como um todo ao início da execução de um programa. Este é o caso de escalonamento dinâmico.

Neste curso estamos interessados especialmente em escalonamento estático, quando o grafo de precedência é conhecido antes do início da execução.

Dado um grafo de precedência G, nós adotaremos a seguinte terminologia:

Sucessores: O conjunto de sucessores de um vértice v é formado por todos os nós u tais que existe um caminho dirigido de v a u em G;

Sucessores Diretos: O conjunto de sucessores de um vértice v é formado por todos os nós u tais que existe um arco de v a u em G;

Predecessores: O conjunto de predecessores de um vértice v é formado por todos os nós u tais que existe um caminho dirigido de u a v em G;

Predecessores Diretos: O conjunto de predecessores de um vértice v é formado por todos os nós u tais que existe um arco de u a v em G;

Largura: A cardinalidade do maior conjunto de vértices tais que nenhum par deles pertence ao mesmo caminho dirigido;

Caminho Crítico: O maior caminho dirigido do grafo, considerando os pesos dos vértices;

Granularidade: Relação entre os pesos dos vértices e dos arcos. A definição formal é a seguinte: A granularidade ρ de um grafo dirigido G é a razão entre o menor peso de um vértice e o maior peso de um arco de G. Se $\rho \leq 1$ o grafo é chamado de grão fino, caso contrário é chamado de grão grande. Quanto maior ρ é maior a relação entre o cálculo e a comunicação.

Para mostrar estas definições usaremos o grafo dirigido da figura 1.2. Os sucessores do vértice v_5 são os vértices $\{v_8, v_9, v_{10}, v_{11}\}$. Os predecessores diretos de v_9 são $\{v_5, v_7\}$. A largura do grafo é 4. Se todas as tarefas do grafo têm o mesmo peso, existem três caminhos críticos: (v_1, v_3, v_7, v_9), (v_1, v_4, v_7, v_9) e (v_1, v_5, v_8, v_{10}).

Para ilustrar a construção de um grafo de precedência a partir de um algoritmo, veremos o exemplo da multiplicação de matrizes quadradas. O algoritmo

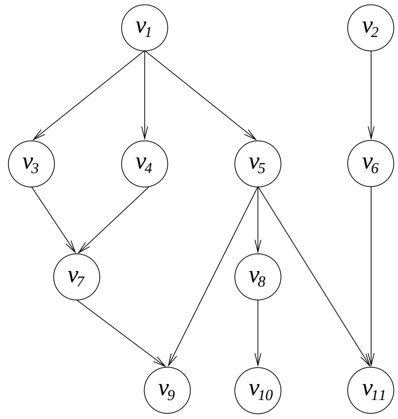

Figura 1.2: Exemplo de grafo de precedência.

funciona para matrizes com dimensões pares, caso contrário, o mesmo pode ser facilmente adaptado. O primeiro passo é a escolha de um algoritmo que resolve o problema. Nós escolhemos o algoritmo de Strassen[4].

Dadas duas matrizes $n \times n$, A e B, onde n é par, queremos calcular $C = A \times B$. Particionando as matrizes em blocos temos:

$$\begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix} \times \begin{pmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{pmatrix} = \begin{pmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{pmatrix}.$$

Cada quadrante da matriz C é calculado segundo o algoritmo 1.

Algorithm 1 Algoritmo de Strassen.

Require: Matrizes A e B particionadas em quadrantes

$T_1 = A_{11} + A_{12}$; $T_2 = A_{21} - A_{11}$;
$T_3 = A_{12} - A_{22}$; $T_4 = A_{21} + A_{22}$; $T_5 = A_{11} + A_{22}$;
$U_1 = B_{11} + B_{22}$; $U_2 = B_{11} + B_{12}$;
$U_3 = B_{21} - B_{11}$; $U_4 = B_{12} - B_{22}$; $U_5 = B_{21} + B_{22}$;
$P_1 = T_1 \times U_1$; $P_2 = T_4 \times B_{11}$; $P_3 = A_{11} \times U_4$;
$P_4 = A_{22} \times U_3$; $P_5 = T_1 \times B_{22}$; $P_6 = T_2 \times U_2$; $P_7 = T_3 \times U_5$;
$C_{11} = P_1 + P_4 - P_5 + P_7$; $C_{12} = P_3 + P_5$;
$C_{21} = P_2 + P_4$; $C_{22} = P_1 + P_3 - P_2 + P_6$;

Existem várias formas de representar um algoritmo através de um grafo de precedência. No caso extremo, o grafo de precedência pode ter somente uma tarefa, correspondente a todo o algoritmo. Por outro lado, também é possível representar cada instrução elementar por uma tarefa. A escolha da representação correta também não é uma tarefa fácil, e geralmente tem que ser feita de forma não automatizada.

Na figura 1.3, temos um grafo de precedência para o algoritmo de Strassen. Nesta figura, cada tarefa tem como etiqueta os seus dados de saída. Para simplificar, não foram colocados os pesos dos arcos.

[4] Algoritmo que efetua menos multiplicações que o tradicional.

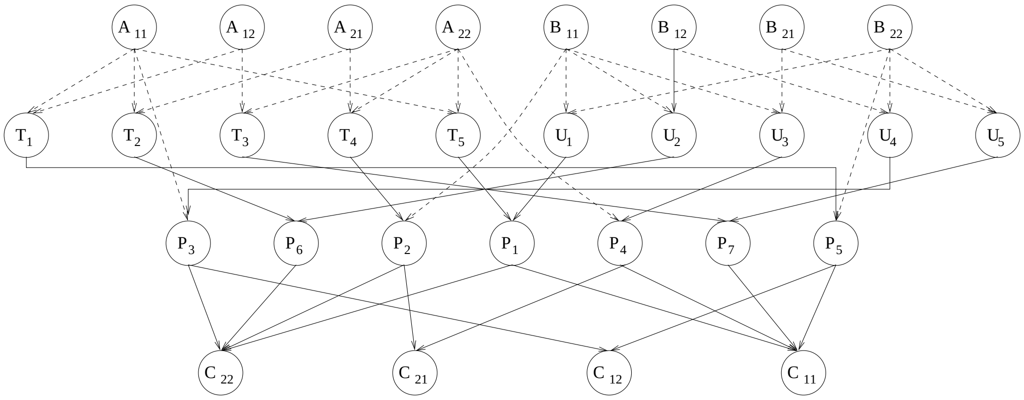

Figura 1.3: Grafo de precedência do algoritmos de Strassen.

Cada tarefa U_i, T_j corresponde a soma/subtração de matrizes. Estas tarefas tem $n^2/4$ operações aritméticas. As tarefas P_i correspondem a multiplicação de matrizes.

Duas outras formas de representar o mesmo algoritmo são: particionar as tarefas U_i, T_j e C_{ij}, ou representar as multiplicações de forma recursiva. Isto pode ser feito até que as tarefas do grafo correspondam apenas a multiplicação de escalares.

Uma outra forma de representar o mesmo algoritmo pode ser visto na figura 1.4. Neste grafo, as tarefas AT (BU) computam os valores T_i (U_i). A tarefa PC calcula os valores P_i, e também a matriz C.

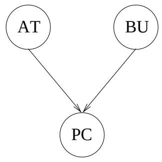

Figura 1.4: Um outro grafo de precedência para o algoritmo de Strassen.

Uma outra forma de encontrar um grafo de precedência é a partir de uma descrição em alto nível do algoritmo. Um grafo de precedência para o algoritmo 2 é dado pela figura 1.5 (os nomes dados as tarefas são evidentes). Este grafo é uma cadeia, e as tarefas f_1, f_2 e f_3 devem ser executadas nesta ordem.

Algorithm 2 Algoritmo seqüencial

$b = f_1(a);$
$c = f_2(b);$
$d = f_3(c);$

A partir de um grafo de precedência nós temos diversas informações sobre o possível paralelismo. A largura do grafo fornece o número de processadores necessários, pois, na alocação das tarefas aos processadores o número máximo de tarefas simultâneas é igual a largura do grafo. Entretanto, existem resultados clássicos onde

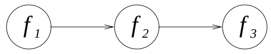

Figura 1.5: Grafo de precedência do tipo cadeia.

o número de processadores é tão grande quanto o número de tarefas [PY90]. O caminho crítico fornece um limite inferior do tempo de execução. A granularidade do grafo fornece uma estimativa da relação entre o tempo de cálculo e o tempo de comunicação.

Os grafos de precedência dados para o algoritmo de Strassen são de granularidade fina, pois a quantidade de dados a serem comunicados é grande. Se no caso do algoritmo 2 as variáveis são matrizes e as funções f_i são complexas (inversão de matrizes, valor próprio, etc) o grafo de precedência da figura 1.5 é de granularidade grossa.

De forma a permitir boas estimativas com relação ao tempo de execução, nós precisaremos de modelos de máquina e de execução.

1.3. Modelos para máquinas paralelas

Os modelos são utilizados para classificar os diversos tipos de máquinas, além disto, os modelos também são usados como base para a programação. Estes modelos são denominados, respectivamente, de modelos de máquina e modelos de execução. Os modelos de máquina estão diretamente ligados a arquitetura, enquanto que os modelo de execução servem para descrever o comportamento da máquina, e fornecer características tais como a gestão das comunicações.

1.3.1. Descrição das características das máquinas

Durante muito tempo, a classificação mais utilizada para máquinas paralelas foi a proposta por Flynn [Fly66], em 1966. As máquinas eram classificadas segundo dois critérios: o fluxo de instruções e o fluxo de dados.

Quando o fluxo de instruções e dados é único, a máquina é chamada de SISD (*single instruction single data*), o que corresponde as máquinas seqüenciais tradicionais. Os outros dois tipos comuns são SIMD (*single instruction multiple data*) que tem um único fluxo de instruções, e vários fluxos de dados, e as máquinas MIMD (*multiple instruction multiple data*) onde os fluxos de dados e de instruções são múltiplos.

Hoje em dia, a maior parte das máquinas paralelas é MIMD, logo a classificação de Flynn não é mais suficiente. Normalmente, divide-se a classe MIMD em dois grandes grupos: os multi-computadores e os multi-processadores [Tan92], sendo que as latências para a troca de mensagem é maior nos multi-computadores.

Uma outra diferença importante é a arquitetura da memória:

UMA: (*Uniform Memory Access*) Existe um espaço de endereçamento comum acessível a todos os processadores. O tempo de acesso a esta memória é aproximadamente os mesmo para todos os processadores;

NUMA: (*Non-Uniform Memory Access*) Cada processador possui a sua memória local. Neste caso, a máquina e seu sistema operacional fornecem primitivas de comunicação entre os processadores. Existem diversas formas de acesso a memória de outros processadores, entre elas: um espaço de endereçamento único como no CRAY T3E, ou troca de mensagens como no IBM SP2. Em uma máquina com arquitetura NUMA, o tempo de acesso a uma memória não local é muito maior do que o tempo de acesso local. A noção de localidade de dados é muito importante.

Por um lado, a possibilidade de se ter acesso a toda a memória da máquina de uma forma uniforme e rápida simplifica o trabalho de paralelização. Entretanto, restrições tecnológicas fazem com que o número de processadores em uma máquina UMA seja limitado (por aproximadamente 50 processadores). Além disto, máquinas UMA com mais de 16 processadores são muito caras. Logo, para dispormos de muitos processadores a única opção é baseada na arquitetura NUMA. Neste modelo, os processadores estão ligados por uma rede de interconexão. A maneira na qual esta interconexão é feita determina a topologia da rede. Existem máquinas com arquitetura mista, como por exemplo um *cluster* de PCs biprocessados, mas os problema de comunicação são semelhantes aos de arquiteturas NUMA puras.

O modelo sistólico [KL80] é ainda mais restrito que o modelo SIMD, os processadores são ligados por uma rede de interconexão, e só os processadores na fronteira podem se comunicar com o mundo exterior. Todos os processadores tem memória local muito limitada. Todo o funcionamento é síncrono, e os processadores repetem os passos: recepção, cálculo e envio. Em uma rede sistólica, os dados são fornecidos de forma regular, e após a passagem dos dados pela rede, os resultados também são fornecidos com ritmo regular.

Apesar destes modelos serem suficientes para classificar as máquinas existentes, eles não são suficientes para o desenvolvimento de aplicações paralelas. Se nós precisarmos conhecer as formas de tratamento de conflito em acessos concorrentes, ou as formas de comunicação entre processadores, precisaremos de modelos de execução.

1.3.2. Modelos de Execução

O modelo de execução está profundamente relacionado ao modelo da máquina. Por exemplo, um modelo de execução que não considera o tempo de acesso a um dado distante não é adequado a uma máquina MIMD com arquitetura NUMA. A apresentação dos diversos modelos de execução está dividida segundo a sua origem. Primeiro veremos os modelos teóricos, isto é, que não foram inspirados nas máquinas reais. Em seguida veremos os modelos baseados nas máquinas existentes.

Os modelos de execução são a base da programação de uma máquina paralela. Eles fornecem a semântica da execução. Um dos seus principais objetivos é fornecer boas previsões do tempo de execução de um programa paralelo. Nós vamos restringir este estudo inicial aos modelos onde todos os processadores têm a mesma velocidade e executam o mesmo conjunto de instruções.

Veremos inicialmente o modelo PRAM que é teórico. Em seguida veremos o modelo com atraso de comunicação, que motivou vários resultados interessantes. Logo após, apresentaremos os modelos mais próximos as máquina, como o sistólico, o uso de topologias, e o LogP. Para encerrar estudaremos os modelos que abstraem as características das máquinas. São eles BSP, CGM e tarefas maleáveis.

1.3.3. Um modelo para a complexidade PRAM

O modelo PRAM (*Parallel Random Access Machine*) foi o primeiro modelo da computação paralela [FW78]. Até hoje, é um modelo de referência, sendo ainda muito usado para a análise e comparação de algoritmos paralelos [KR90]. Neste modelo, os processadores são idênticos, funcionam de maneira síncrona, e têm acesso a uma memória global comum. O número de processadores e o tamanho da memória não são limitados. Este modelo não é realista na prática, pois o custo de manutenção de uma memória global depende do número de processadores.

Um outro modelo um pouco mais restrito, a p-PRAM, onde o número de processadores é igual a p também foi proposto na literatura. Este modelo representa o comportamento das máquinas SMP (*Symmetric MultiProcessing*), onde os recursos de memória e entrada/saída são compartilhados. Os PCs de última geração, com placas mãe que aceitam até oito processadores, são máquinas SMP.

Com o objetivo de se definir regras de acesso a memória no modelo PRAM o modelo foi refinado nos seguintes tipos. O EREW (*Exclusive Read Exclusive Write*), onde cada célula de memória só é acessível por um processador simultaneamente, o CREW (*Concurrent Read Exclusive Write*) e o CRCW (*Concurrent Read Concurrent Write*), onde o acesso a memória pode ser concorrente para a leitura, ou escrita, respectivamente. Para o último caso, também existem várias regras para a resolução de conflitos. As mais comuns são: arbitrária, prioritária e combinação (por exemplo máximo, soma, etc).

Akl e Guenther [AG89] propuseram o modelo BSR (*Broadcasting with selective reduction*) baseado na PRAM CRCW. Eles perceberam que a adição de uma instrução de difusão em tempo constante era realista. A instrução de difusão proposta permite o acesso simultâneo em escrita de todas as células de memória para cada processador. As regras de resolução de conflito são semelhantes as da PRAM.

Várias modificações do modelo original, onde as restrições de acesso a memória distante são consideradas, foram propostas na literatura. Existem duas classes principais: as que consideram a topologia da rede, como o modelo XRAM [CF91], e outros que ignoram a topologia, como a LPRAM [ACS90] (*Local-memory PRAM*).

Nos modelos do tipo PRAM os problemas de comunicação ficam escondidos. O custo da comunicação está implícito no modelo. Entretanto, diversos autores, como JáJá [BHJ95], defendem o uso deste modelo. As razões principais são as seguintes:

- Existem várias técnicas bem conhecidas para o modelo;
- No modelo não há necessidade de levar em conta "detalhes" como a comunicação, ou a sincronização.

Mas, é importante ressaltar que por ser muito poderoso, o modelo PRAM, é bem distante da realidade atual.

1.3.3.1. Alguns algoritmos para o modelo PRAM

Veremos dois algoritmos para a PRAM, o cálculo da soma de n elementos, e a soma prefixa.

Dados um vetor A de $n = 2^k$ elementos, e uma PRAM com n processadores, queremos calcular a soma dos n elementos. O algoritmo que cada processador, P_i executa é o seguinte:

Algorithm 3 Soma em uma PRAM, código do processador i.

Require: Vetor A
  read$(A(i), a)$;
  write$(a, B(i))$;
  for $h = 1$ to $\log n$ **do**
    if $i \leq n/2$ **then**
      read$(B(2i-1), x)$;
      read$(B(2i), y)$;
      $z = x + y$;
      write$(z, B(i))$;
    end if
  end for
  if $i = 1$ **then**
    write(z, S);
  end if

No final, a posição de memória S vai conter a soma de todos os elementos.

O segundo problema que veremos é a soma prefixa. Dados $n = 2^k$ elementos, $\{x_1, x_2, \ldots, x_n\}$, queremos calcular todas as somas parciais $s_i = x_1 + x_2 + \ldots + x_i, 1 \leq i \leq n$. O algoritmo seqüencial que calcula esta soma, através de operações $s_i = s_{i-1} * x_i$, leva tempo $O(n)$. Para este algoritmo, serão utilizadas duas matrizes auxiliares: $B(h, i)$ e $C(h, i)$, onde $0 \leq h \leq \log n$ e $1 \leq i \leq n/2^h$.

O algoritmo funciona em duas fases, primeiro são calculadas somas auxiliares. Veja um exemplo na figura 1.6. Neste exemplo, temos como entrada um vetor com 8 números. Cada vértice da árvore corresponde a soma de suas sub árvores. Sendo que, inicialmente, $B(0, i) = A(i)$.

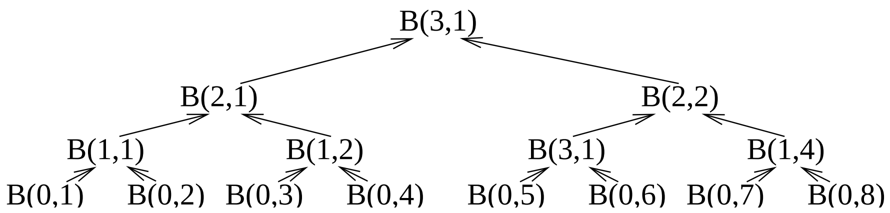

Figura 1.6: Cálculo de algumas somas parciais na matriz B.

A partir da matriz B é possível calcular a matriz A (último laço do algoritmo), sendo que a resposta final do problema estará em $C(0, i)$. Veja na figura 1.7 a forma de cálculo da matriz C, que é de cima para baixo (ao contrário do cálculo da matriz B).

Algorithm 4 Soma prefixa, código executado no processador P_i.

Require: Vetor A com os elementos
  $B(0,j) = A(j)$;
  for $h = 1$ to $\log n$ **do**
    if $i \leq n/2^h$ **then**
      $B(h,i) = B(h-1, 2i-1) + B(h-1, 2i)$;
    end if
  end for
  for $h = \log n$ to 0 **do**
    if i é par **then**
      $C(h,i) = C(h+1, j/2)$;
    end if
    if $i == 1$ **then**
      $C(h,1) = B(h,1)$;
    end if
    if $i > 1$ e ímpar **then**
      $C(h,i) = C(h+1, (j-1)/2) + B(h,i)$;
    end if
  end for

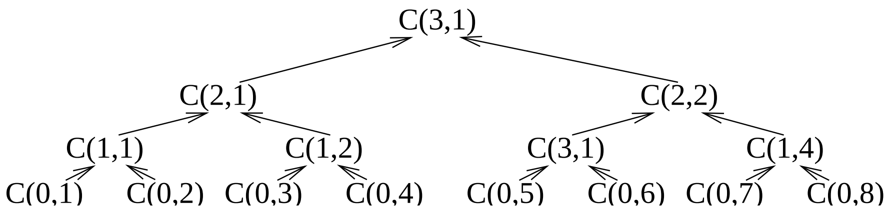

Figura 1.7: Cálculo da matriz C.

Este algoritmo tem tempo $O(\log n)$. Para mais algoritmos no modelo PRAM, [BHJ95] é uma boa referência.

1.3.4. Modelos com atraso de comunicação

Nos modelos com atraso de comunicação temos os processadores, e o suporte para a comunicação entre eles. Este suporte consiste de primitivas para o envio e a recepção de mensagens. Nestes modelos são considerados apenas o tamanho das tarefas, e o tempo de comunicação entre a tarefas consecutivas em processadores distintos.

Os modelos com atraso de comunicação e a técnicas de escalonamento para os mesmos foram propostas de maneira simultânea. Durante a transmissão, ou recepção, de mensagens em uma máquina real, o processador fica ocupado durante algum tempo (cópias de memória, alocação de *buffers*, etc). Os modelos com atraso de comunicação não consideram este tempo sobre o tempo de execução. Os processadores podem continuar executando outras coisas durante a comunicação. Estes modelos também não consideram a congestão da rede. Estes dois fatores são muito importantes para outros modelos, como o LogP, descrito na seção 1.3.7.

Primeiro, nós veremos o modelo com banda passante ilimitada, isto é, sem custo de comunicação. Em seguida, nos veremos os modelos com custo de comunicação, este custo introduz o atraso de comunicação. É interessante notar que uma das formas de evitar o custo de comunicação é através da duplicação de tarefas. Ao invés de efetuar a comunicação a partir de predecessores, também é possível duplicar alguns deles.

Para ilustrar as diferenças dos diversos modelos veremos exemplos de execução do grafo G da figura 1.2, em diferentes modelos de execução. A representação utilizada é o diagrama de Gantt (diagrama de tempo e espaço, onde o espaço corresponde a ocupação dos processadores), onde as tarefas, as comunicações e os tempos de espera são colocados segundo suas datas, processadores e duração. Os pesos das tarefas é unitário, o peso dos arcos será dado para cada modelo.

1.3.4.1. Modelo UET

Neste modelo os tempos de comunicação, entre processadores diferentes, são simplesmente ignorados. O modelo UET (*unit execution time*) foi proposto por Papadimitriou e Ullman [PU87]. O tempo de execução das tarefas é unitário, e a espera devida as comunicações não é levada em conta. Para este modelo não existe a necessidade de se duplicar tarefas.

Este modelo é bem parecido com o modelo da PRAM. Veja na figura 1.8 um esquema de execução ótimo.

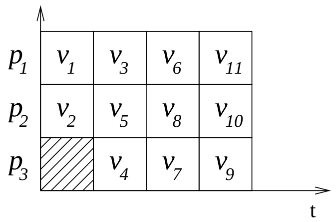

Figura 1.8: Execução no modelo UET.

Na figura, as zonas hachuradas correspondem aos tempos de inatividade dos processadores.

1.3.4.2. Modelo UET-UCT

A extensão natural do modelo UET considera de maneira simplificada o custo das comunicações. Neste caso, quando os custos de comunicação também são unitários, temos o modelo UET-UCT (*unit execution time - unit communication time*). Este modelo foi proposto por Rayward-Smith [RS87]. Os esquemas de execução, com e sem duplicação estão ilustrados na figura 1.9.

Neste exemplo, podemos ver as vantagens da duplicação. Permitindo a duplicação, foi possível diminuir o tempo de execução, em uma unidade. Na figura 1.9 e nas próximas figuras, as tarefas duplicadas de uma tarefa v serão nomeadas por v'.

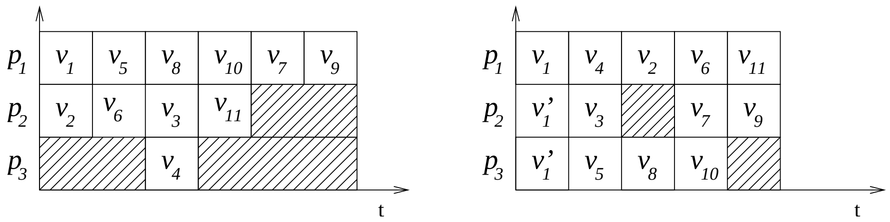

Figura 1.9: Execuções no modelo UET-UCT sem, e com, duplicação.

1.3.4.3. Modelo UET-LCT

Este modelo foi proposto por Papadimitriou e Yannakakis [PY90]. O custo das tarefas continua sendo unitário, mas o custo das comunicações é dado por $\gamma > 1$. Este modelo é chamado de UET-LCT (*unit execution time - large communication time*). No mesmo artigo onde o modelo é proposto, os autores também apresentam outros resultados como: uma prova da dificuldade de resolver este problema de forma exata e um algoritmo de aproximação.

Este modelo aproxima as redes de computadores, ou aglomerados, onde os processadores são rápidos, e as latências de comunicação são grandes, e representam o gargalo do sistema. Na figura 1.10, temos um exemplo de execução onde $\gamma = 2,5$.

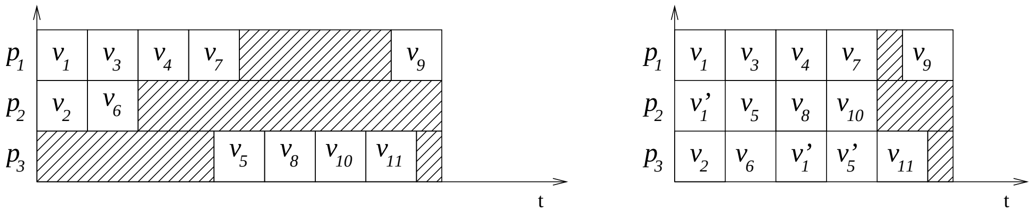

Figura 1.10: Execução no modelo UET-LCT sem, e com, duplicação.

Nos modelos que vimos até o momento, os custos de execução e de comunicação foram representados por constantes. Veremos em seguida outros modelos com custos variáveis.

1.3.4.4. Modelo SCT

Neste modelo, proposto por Colin e Chrétienne [CC91], supõe-se apenas que os tempos de execução são maiores do que os tempos de comunicação. Isto é, o maior tempo de comunicação é menor do que o menor tempo de execução. Este modelo é adequado para aplicações com pouca comunicação, ou em redes muito rápidas.

Na figura 1.11 podemos ver os esquemas de execução do grafo G, com e sem duplicação. Os tempos de comunicação na figura correspondem a metade do tempo de execução das tarefas.

1.3.4.5. Modelos mais gerais

Também existem na literatura modelos mais gerais, onde o tempo de execução das tarefas, e os custos de comunicação, são variáveis [HCAL89]. Também

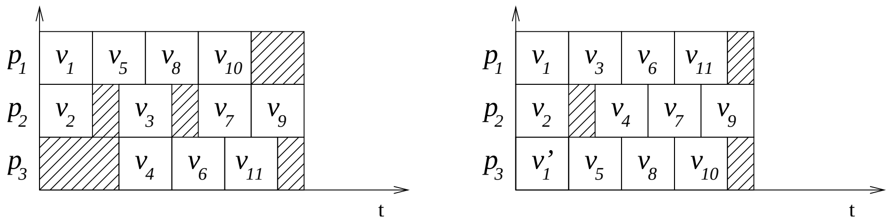

Figura 1.11: Esquemas de execução no modelo SCT sem, e com, duplicação.

em [HCAL89] foi considerada a topologia da máquina, isto é, existe uma função que corresponde ao tempo de transmissão de uma palavra para cada par de processadores. Este modelo é bem realista. Entretanto, não foi possível extrair nenhum resultado genérico interessante.

Somente neste modelo, a topologia da rede de comunicação é considerada. Neste caso, a posição de um processador na rede afeta o custo de comunicação.

1.3.4.6. Algoritmos

Nesta seção veremos dois algoritmos para os modelos com atraso de comunicação. Primeiro veremos um algoritmo simples para o escalonamento de árvores com altura 1, em seguida veremos o algoritmo clássico para grandes tempos de comunicação. É interessante notar que a maioria dos resultados para estes modelos apresentam algoritmos com garantias para tipos genéricos de grafos.

O primeiro algoritmo é para o problema do envio, dada uma árvore com um nó raiz, e vários filhos (nós folha), como encontrar o melhor escalonamento possível em um número ilimitado de processadores, sem duplicação ? Um exemplo pode ser visto na figura 1.12. No exemplo, o grafo tem 9 tarefas, e os pesos de comunicação ($c_{0,i}$) e processamento (p_i) estão na própria figura.

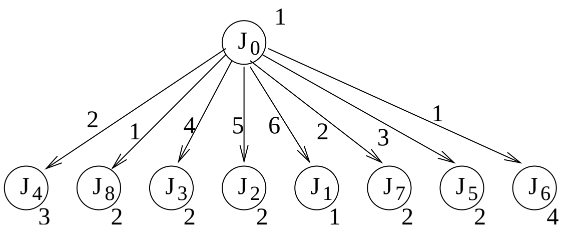

Figura 1.12: Grafo do tipo *send*.

Obviamente, nenhum outro processador, fora o processador onde a raiz é executada, deve executar mais do que uma tarefa. Além disto, cada uma das tarefas folha pode ser executada em outro processador, assim que a mensagem da raiz chegar. A idéia básica é ordenar as tarefas em ordem da comunicação mais tamanho, isto é: $p_1 + c_{01} \geq p_2 + c_{02} \geq \ldots \geq p_n + c_{0n}$.

Se a tarefa J_i não é processada pela raiz, nenhuma das outras precisa ser, isto é, nenhuma das outras vai contribuir em aumentar o tempo de processamento. Logo, o algoritmo tem apenas que determinar a tarefa J_i que minimiza $\max\{\sum_{k=1}^{i} p_k, p_{i+1} + c_{0,i+1}\}$. Veja o escalonamento ótimo na figura 1.13.

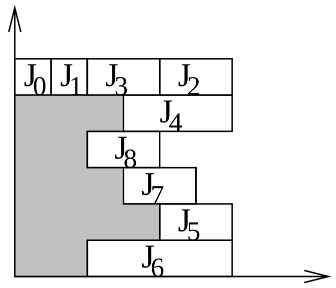

Figura 1.13: Esquema de execução de um grafo *send*.

Logo, a partir de qualquer problema com grafo de precedência na forma de *send*, é possível propor um algoritmo que o resolve em tempo ótimo. Este resultado pode ser aplicado em máquinas reais (existem certos fatores que não são considerados no modelo com atraso, mas são considerados no modelo LogP).

O segundo exemplo é um pouco mais complicado. O problema tratado é o de grafos de precedência com tarefas unitárias, tempo de comunicação grande (c), e número ilimitado de processadores [PY90]. Neste caso, primeiro foi demonstrado que encontrar a solução ótima é um problema NP-completo, isto é de difícil resolução. Entretanto, também foi proposto um algoritmo simples, que tem a garantia de ter tempo de execução de no máximo duas vezes o tempo ótimo.

A idéia principal do algoritmo está baseada nos tempos de disponibilidade (*release times*), este tempo indica o menor instante de tempo possível para o início da execução de cada tarefa. Para cada tarefa k, o tempo de disponibilidade r_k é construído incrementalmente. Inicialmente, as tarefas sem predecessores recebem $r_k = 0$. Em seguida, uma tarefa para qual todos os seus predecessores já tem foram visitados é visitada. Este passo é aplicado até que não existam mais tarefas não visitadas. De uma forma mais formal:

- Se a tarefa v não tem predecessor, $r_v = 0$;
- Caso contrário, seja $E = \{u_1, \ldots, u_p\}$, o conjunto de predecessores de v ordenando em ordem decrescente de tempo de disponibilidade. Seja $k = \min\{c+1, p\}$, $r_v = r_{u_k} + k$.

Primeiro Papadimitriou e Yannakakis[PY90] provaram que não existia escalonamento onde alguma tarefa fosse alocada antes de sua data de disponibilidade. Em seguida, propuseram um algoritmo de aproximação que garante que cada tarefa terá seu início até o dobro do seu tempo de disponibilidade, logo tem uma garantia de desempenho de 2. O algoritmo funciona da seguinte forma:

- Aloque apenas as tarefas as quais seus predecessores já foram alocados, e uma por processador;
- Tarefas v com $r_v = 0$ são alocadas no instante 0;

- Tarefas v com predecessores são alocadas no instante r_v, juntamente com os seus k predecessores com maiores valores de tempo de disponibilidade.

Veja um exemplo na figura 1.14. Nesta figura temos um grafo de precedência com tarefas unitárias, e tempos de comunicação $c = 2$. Temos os seguintes tempos de disponibilidade: $r_1 = r_2 = r_3 = 0, r_4 = 1, r_5 = 2$, para calcular r_6 temos 4 predecessores $\{5, 4, 3, 2, 1\}$, logo $r_6 = 0 + 3$, e finalmente r_7 tem 6 predecessores $\{6, 5, 4, 3, 2, 1\}$, logo $r_7 = 1 + 3 = 4$. No lado direito da figura, temos um escalonamento onde todas as tarefas são alocadas no dobro do tempo de disponibilidade.

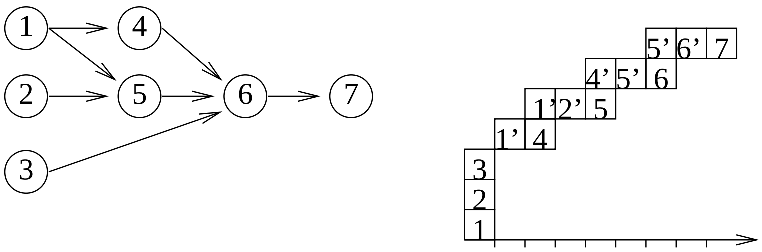

Figura 1.14: Exemplo de escalonamento com grande tempo de comunicação.

O algoritmos apresentado tem a mesma garantia de desempenho para grafos de precedência quaisquer. É interessante notar que no exemplo da figura 1.14 seria fácil diminuir o número de processadores, mas o problema genérico de se encontrar em escalonamento com a mesma garantia, com o mínimo de processadores, é um problema difícil.

1.3.5. Modelo Sistólico

Nós já vimos a definição do modelo sistólico na seção sobre modelos de máquinas. O modelo de execução confunde-se com o modelo de máquina. Veremos a implementação de dois algoritmos, um de ordenação, e outro para o cálculo do produto de duas matrizes.

A ordenação de n números pode ser feita de forma muito simples com uma rede linear de células sistólicas. Cada célula deve ser capaz de armazenar um número. A cada passo as células recebem um novo número, calculam o máximo entre o número atual e o número recebido, e enviam o mínimo para a próxima célula. Supõe-se que antes do início do algoritmos as células contém um número muito pequeno ($-\infty$), e que após a entrada dos n números são inseridos $2n$ números muito grandes ($+\infty$). O comportamento das células pode ser visto na figura 1.15.

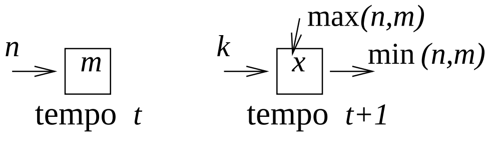

Figura 1.15: Comportamento das células para a ordenação.

Na figura seguinte temos alguns passos da ordenação de 7 números, para não sobrecarregar a figura, as células vazias contém $-\infty$, e $+\infty$ está representado apenas pelo sinal de $+$. Um simulador, feito em Java pode ser visto em: `http://www.brunel.ac.uk/~castjjg/java/`. Este simulador usado outra arquitetura, mas também resolve o mesmo problema.

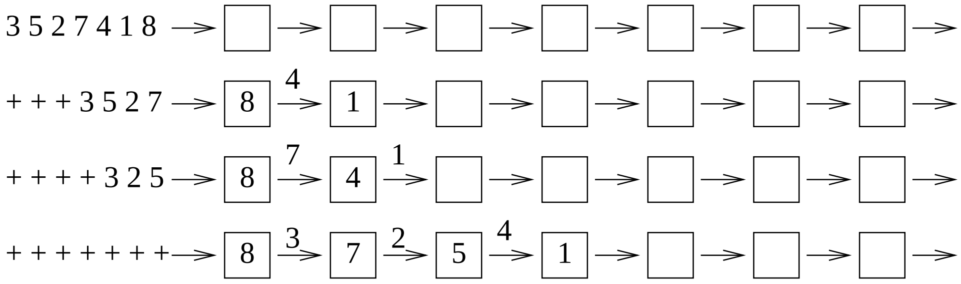

Figura 1.16: Vetor sistólico de ordenação.

O exemplo seguinte mostra a multiplicação de matrizes, para isto são necessárias células um pouco mais sofisticadas. As células tem duas entradas, e duas saídas. O comportamento pode ser visto na figura 1.17.

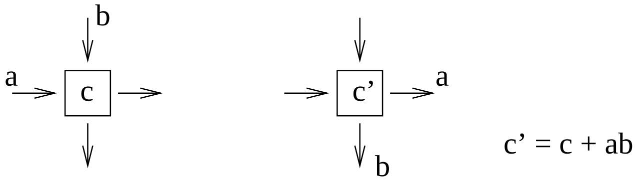

Figura 1.17: Célula para a multiplicação de matrizes.

Entretanto, para este problema, é necessário que a célula possua diversos estados diferentes:

- Início: quando o valor armazenado é zero;
- Cálculo: quando são feitos os produtos dos valores recebidos, e é feita a transmissão destes mesmo valores;
- Finalização: quando os valores calculados são enviados para a direita.

Os diversos estados são necessários, pois no modelo sistólico supõe-se que apenas as células nas fronteiras podem se comunicar com o mundo exterior. Além disto, pode se supor que as células mudam de estado, quando começam a receber dados, e quando não recebem mais dados. A alimentação dos dados pode ser vista na figura 1.18.

1.3.6. Modelos com topologias específicas

No início da computação paralela foram propostas várias formas de se conectar processadores. A tendência atual é a representação máquinas onde a comunicação

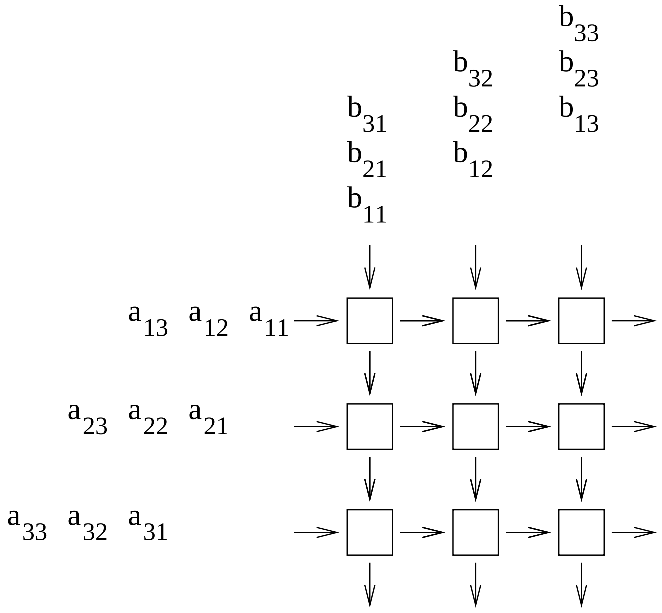

Figura 1.18: Exemplo da multiplicação de matrizes.

entre quaisquer dois processadores leva o mesmo tempo. Isto é uma conseqüência da evolução das formas de comunicação. Mas, até hoje, muitas pesquisas ainda são feitas para topologias específicas. Veremos a seguir algumas destas topologias, e em seguida exemplos de comunicação. As topologias apresentadas a seguir são geralmente modeladas através de um grafo.

1.3.6.1. Anel

Um anel (*ring*) $R_n(V, E)$ é definido como sendo um circuito de n vértices (figura 1.19). Observe que o diâmetro de um anel com n vértices é $\lfloor \frac{n}{2} \rfloor$ e todo vértice tem grau 2.

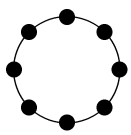

Figura 1.19: Anel com 8 vértices (R_8).

1.3.6.2. d-grade

Uma d-grade (*grid*) é o resultado do produto cartesiano de d caminhos. Formalmente uma d-grade $G^d(V, E) = P_{n_1} \times \ldots \times P_{n_d}$ onde $P_{n_i}, 1 \leq i \leq d$, são caminhos (figura 1.20). O diâmetro de uma d-grade é igual a soma dos diâmetros dos caminhos componentes, isto é, $D = \sum_{i=1}^{d}(n_i - 1)$ onde cada $n_i, 1 \leq i \leq d$, é o número de vértices do caminho P_{n_i}. O grau máximo de um vértice na d-grade é $\mathrm{gr}(V) = 2d$ (o mínimo é d).

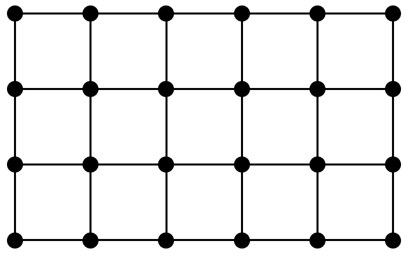

Figura 1.20: Grade $G^2(V, E) = P_4 \times P_6$.

1.3.6.3. d-toro

Assim como no ítem anterior, definimos o d-toro (*torus*) como um produto de d anéis. Um d-toro $T^d(V, E) = R_{n_1} \times \ldots \times R_{n_d}$ onde $R_{n_i}, 1 \leq i \leq d$, são anéis. O grau de cada vértice é $\mathrm{gr}(V) = 2d$ e o diâmetro é $D = \sum_{i=1}^{d} \left\lfloor \frac{n_i}{2} \right\rfloor$, onde n_i é o número de vértices do anel R_{n_i}. O toro também é chamado de grade toroidal.

1.3.6.4. Hipercubo

Um hipercubo $H_d(V, E)$ (figura 1.21) é um grafo com $N = 2^d$ vértices onde

$$V = \{(b_1, \ldots, b_d) \mid b_i \in \{0, 1\}, 1 \leq i \leq d\}$$

$$e \in E \leftrightarrow e = \{v_1, v_2\} \text{ onde } \begin{cases} v_1, v_2 \in V \\ v_1 \oplus v_2 = e_j = \ j\text{-ésima coordenada canônica} \\ (\oplus \text{ denota a operação de ou-exclusivo}). \end{cases}$$

O hipercubo tem diâmetro d e todos os seus vértices têm grau $\mathrm{gr}(V) = d$.

1.3.6.5. Transmissão entre vértices adjacentes

Supomos que dada uma rede de interconexão, representada por um grafo $G(V, E)$, o tempo de transmissão de uma mensagem M entre dois vértices adjacentes $u, v \in V$ não depende dos vértices escolhidos. Existem vários modelos em relação ao tempo de transmissão de uma mensagem entre dois vértices adjacentes, veremos uma breve descrição de alguns modelos abaixo.

Modelo constante Supõe-se que as mensagens são suficientemente curtas para poderem ser transmitidas em apenas um ciclo, que é composto de transmissão e/ou recepção de dados.

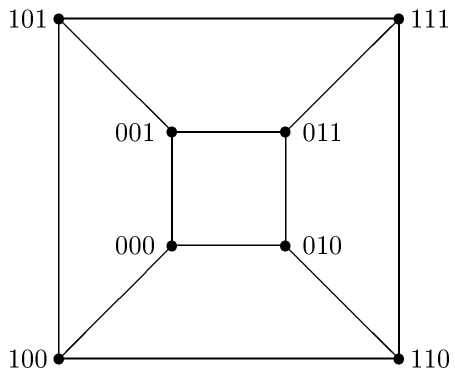

Figura 1.21: Hipercubo H_3.

Modelo linear O tempo de transmissão é proporcional ao tamanho da mensagem mais o tempo de inicialização (*start-up*). Isto é, o tempo de transmissão de uma mensagem M, de tamanho L, entre dois vértices adjacentes é

$$T(M) = \beta + \gamma L$$

onde β é o tempo para a inicialização da transmissão e γ é o tempo de propagação de uma unidade de tamanho da mensagem, $\frac{1}{\gamma}$ é a largura de banda das arestas. Neste caso a transmissão de mensagens de tamanhos diferentes não é sincronizada.

Modelo de pacotes Bertsekas e outros [BOS$^+$91] apresentam um modelo onde a transferência é realizada através de pacotes de tamanho fixo. Logo para a transmissão de uma mensagem M de tamanho L, temos

$$T(M) = \left\lceil \frac{L}{m} \right\rceil m\gamma$$

onde m é o tamanho do pacote e $\frac{1}{\gamma}$ é a largura da banda das arestas. Vemos que no modelo linear os pacotes têm tamanho unitário.

1.3.6.6. Comunicação entre vértices

Existem vários modelos para a comunicação entre vértices adjacentes. Uma aresta pode ser bidirecional, isto é, as informações podem trafegar nos dois sentidos ao mesmo tempo (ex: conversa telefônica). Pode ser unidirecional, no caso em que a aresta só pode transmitir informações em um sentido simultaneamente (ex: *walkie talkie*). O modelo bidirecional é denotado por F (*full duplex*) e o unidirecional por H (*half duplex*).

Há outro modelo importante em relação ao número de arestas que um vértice pode utilizar para a transmissão/recepção de mensagens simultaneamente. Se um vértice pode transmitir/receber por apenas uma aresta por vez chamamos de 1-porta (*processor-bound*) e denotamos por F_1 ou H_1, dependendo se as arestas são uni ou bidirecionais.

Se um vértice pode transmitir por no máximo um dado número k de suas arestas, ao mesmo tempo, chamamos de k-porta (*DMA-bound*) e denotamos por H_k ou F_k. Por último quando um vértice pode transmitir por todas as suas arestas simultaneamente temos o modelo $*$-porta (*link-bound*) cuja notação é F_* ou H_*.

1.3.6.7. Transmissão entre vértices não adjacentes

Existem modelos e técnicas específicas para o envio de mensagens entre vértices não adjacentes. A técnica mais simples é a armazene-e-envie (*store-and-forward*). Quando ocorre a transmissão entre dois vértices não adjacentes u e v, o que acontece é uma série de transmissões entre vértices adjacentes da mensagem (inteira) através de um dos caminhos entre u e v.

Logo, neste caso no modelo linear, para vértices u e v a distância d, temos

$$T_{v \leftarrow u}(M) = d\,(\beta + L\gamma)\,.$$

As outras técnicas mais comuns são:

Técnica *pipeline* Para reduzir os tempos do armazene-e-envie, uma técnica é a de decompor a mensagem M, de tamanho L, em pacotes. Suponha que uma mensagem é decomposta em pacotes de tamanho B. Para simplificar suponha que L é divisível por B. Para transmitir a mensagem de um vértice u para um vértice v à distância d temos

$$T_{v \leftarrow u}(M) = \overbrace{d(\beta + B\gamma)}^{\text{pacote 1}} + \overbrace{\left(\frac{L}{B} - 1\right)(\beta + B\gamma)}^{\text{pacotes restantes}} = \left(d + \frac{L}{B} - 1\right)(\beta + B\gamma).$$

Técnica de caminhos disjuntos A idéia é decompor a mensagem em pacotes e transmiti-los do vértice origem para o vértice destino por caminhos disjuntos. Dados u e v, à distância d, se p é o número de caminhos (aresta disjuntos), de tamanho $d' \geq d$, entre os vértices u (origem) e v (destino), decomponha a mensagem M, de tamanho L, em p blocos de tamanho $\frac{L}{p}$ (para simplificar suponha L divisível por p). Temos

$$T_{v \leftarrow u}(M) = d'(\beta + \frac{L}{p}\gamma).$$

é claro que se pode combinar esta técnica com a técnica de *pipeline*.

Quanto aos modelos existentes, os principais estão relacionados a comutação de circuitos. Os modelos são *wormhole* e *circuit-switching*. Nestes modelos são criadas rotas (de comunicação) quando se deseja transmitir uma mensagem entre dois vértices não adjacentes. Para enviar uma mensagem M de um vértice u para um vértice v, à distância d, temos

$$T_{v \leftarrow u} = \alpha + d\delta + L\gamma$$

onde L é o tamanho da mensagem M, $\frac{1}{\gamma}$ é a largura de banda, α é o tempo de inicialização e δ é o tempo para a criação de uma rota em cada vértice intermediário.

A diferença entre os dois é que no modelo *circuit-switching* a rota é criada antes do início da transmissão, enquanto que no *wormhole* a própria mensagem contém um pacote inicial que cria a rota, e um final que a libera.

É interessante notar que a diferença, nestes modelos, entre a transmissão entre vértices adjacentes, e distantes, é de apenas $d\delta$. Diversos estudos foram feitos no sentido de se mostrar que $d\delta$ não precisa ser considerado na maioria dos caso práticos, logo a noção de vizinhança deixa de ser importante.

1.3.6.8. Algoritmos

Nos casos de topologia fixa, os principais algoritmos correspondem a operações básicas como a difusão e a troca completa. Até hoje são publicados diversos artigos, com algoritmos ótimos para alguma destas operações em um modelo específico (por exemplo troca completa em um toro no modelo de comutação de circuitos).

Como exemplo veremos os algoritmos de troca completa no toro e no hipercubo. Para o anel, $R_n(V, E)$, de tamanho n, no modelo H_1 Cybenko, Krumme e Venkatamaram [CDV86] propuseram o seguinte resultado. Para um anel de tamanho N a troca completa pode ser realizada em $\lfloor \frac{n}{2} \rfloor + \lceil \sqrt{2n} \rceil + 2$ ciclos. Para chegar a este resultado propuseram o seguinte algoritmo. Dado um anel com n vértices $v_0, \ldots, v_{n-1}$ definiram P números inteiros $0 \leq s_1 < s_2 < \ldots < s_P < n$ igualmente espaçados (quando possível) e com amplo espaço entre eles. Usaram o seguinte esquema (figura 1.22).

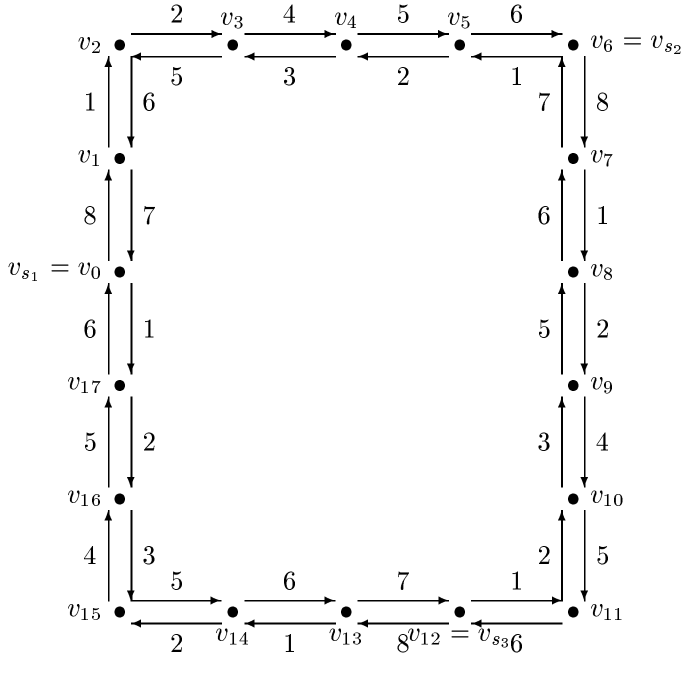

Figura 1.22: Solução ótima para o anel com 18 vértices ($s_1 = 0, s_2 = 6, s_3 = 12$).

Dado um hipercubo $H_d(V, E)$ usa-se o seguinte esquema no modelo F_1. No ciclo i, $1 \leq i \leq d$, cada vértice troca todas as suas informações através da dimensão i. Por indução em d é fácil provar que o tempo para a troca completa é $\log_2 N$.

1.3.7. Modelo LogP

O modelo LogP [CKP+96] foi proposto por especialistas de diversas áreas, arquiteturas, ambientes de execução e algoritmos. Neste modelo supõe-se que existe um número P finito de processadores com memória local. O modelo não leva em conta a topologia da rede. A sincronização entre os processadores é feita através da troca de mensagens. O modelo também considera o custo de comunicação para a troca de mensagens.

No modelo LogP os custos de comunicação são determinados pelos parâmetros L, o e g. Quando um processador envia uma mensagem, ele não pode efetuar outra operação por algum tempo, este custo adicional (*overhead*) é chamado de o. A recepção de uma mensagem também custa um tempo o do receptor. Existem também um intervalo mínimo entre o envio de duas mensagens pelo mesmo processador, este intervalo é chamado de g (da palavra *gap*). Este intervalo também tem que ser respeitado para o recebimento de mensagens. A latência L corresponde ao tempo máximo entre o final da operação de envio, e o início da recepção da mensagem, em condições normais. Para evitar a congestão da rede (excesso de pacotes circulando ao mesmo tempo), no máximo $\lceil \frac{L}{g} \rceil$ mensagens podem estar na rede ao mesmo tempo.

Na figura 1.23 podemos ver os parâmetros do modelo LogP, as tarefas pintadas de preto são as devidas ao custo adicional de transmissão, e recepção. O quadrado cinza representa a latência. Entre duas comunicações consecutivas existe um intervalo de ao menos g.

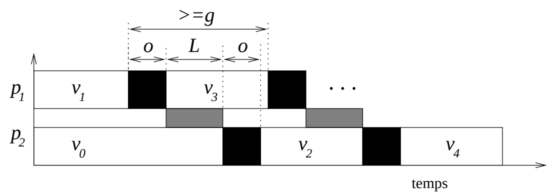

Figura 1.23: Parâmetros do modelo LogP.

Esta primeira definição do modelo LogP foi idealizada para mensagens pequenas. Com mensagens grandes, a latência pode ficar negativa, a primeira parte da mensagem pode chegar ao destino antes da transmissão do final da mensagem. Algumas variações do modelo, que consideram mensagem maiores, foram propostas em [AISS97, ELW97a].

O modelo LogP consegue capturar de uma forma bem realista o comportamento de máquinas reais. Tanto que, podemos ver na tabela 1.1 os valores dos parâmetros do LogP para diversas máquinas. O tamanho, em bytes, da mensagem é dada por x.

A figura 1.24 contém dois escalonamentos no modelo LogP. No primeiro (à esquerda) $o = 0,125$ e $L = 0,25$ do tempo de execução de uma tarefa, o parâmetro g é no máximo 1. No segundo exemplo (à direita) são utilizados os mesmo valores para o e L, mas $g = 1,5$. Logo, a comunicação de v_5 a v_{11} tem que ser atrasada.

Apesar das inegáveis vantagens do modelo LogP, como aproximar quase perfeitamente as características das máquinas atuais, a complexidade adicionada em

	L	o	g	P
CM-5	$6\mu s$	$2.2\mu s$	$4\mu s$	512
Parsytec Xplorer	$-21 - 0.82x\mu s$	$70 + x\mu s$	$115 + 1.43x\mu s$	8
ParaStation	$50 - 0.1x\mu s$	$3 + 0.112x\mu s$	$3 + 0.119x\mu s$	4
IBM SP-1	1000 ciclos	8000 ciclos	-	16
IBM SP-2	$13 - 0.005x\mu s$	$8 + 0.008x\mu s$	$10 + 0.01x\mu s$	32
Meiko CS-2	$8.6\mu s$	$1.7\mu s$	$14.2 + 0.03x\mu s$	64

Tabela 1.1: Parâmetros do modelo LogP em diversas máquinas. Fonte: [AISS97, CKP+96, DI94, ELW97b, LZ95].

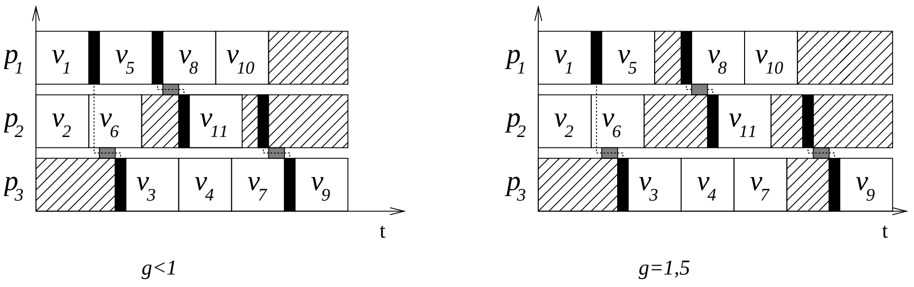

Figura 1.24: Execução no modelo LogP .

relação aos modelos com atraso fez com que os problemas de escalonamento ficassem mais difíceis. Mesmo com o modelo mais simples, boa parte dos problemas de escalonamento são difíceis, e não possuem algoritmos de aproximação [CCLL95]. Os poucos resultados disponíveis para o modelo LogP são relativamente complexos e se restringem a grafos simples como árvores [Ver97, MLZ99, KT99, HCAL89, KKT98].

Nos próximos modelos, ao contrário de aproximar as características das máquinas, nós veremos diferentes modelos que concentram o esforço apenas na programação.

1.3.8. Modelos BSP e CGM

Mais do que um modelo de execução, BSP [HMS96, McC95, Val90] (*Bulk Synchronous Parallel*) é um modelo para a programação. O seu principal objetivo é de fornecer um ambiente que permite a concepção de algoritmos ao mesmo tempo portáveis e eficazes. O modelo BSP não foi baseado em nenhuma máquina real, mas convém as máquinas MIMD.

A principal idéia do modelo é a separação explícita do cálculo e da comunicação. Os seus fundamentos são a super-etapa (*super-step*) e a sincronização. Um aplicação é dividida em super-etapas, e todos os processadores começam uma super-etapa ao mesmo tempo. Entre duas super-etapas existe uma etapa de sincronização. Os dados enviados durante uma super-etapa estarão disponíveis nos seus destinos no início da próxima super-etapa. Os três parâmetros utilizados para descrever este modelo são: p, l em g. Neste texto, nós utilizamos as mesmas notações propostas em [McC95].

p Número de processadores;

l Custo de uma sincronização global;

g Tempo para transmitir um palavra pela rede. Ou seja, $\frac{1}{g}$ é a banda passante.

No modelo original, proposto por Valiant [Val90], existe um parâmetro adicional que representa a periodicidade das sincronizações. Em [Val90], uma verificação global é feita a cada L unidades de tempo. Esta verificação serve para verificar se a super-etapa já foi completada por todos os processadores. Na versão do modelo BSP apresentada por McColl [McC95] não há referência a esta periodicidade. Esta omissão foi proposital, pois na maioria das máquinas atuais esta periodicidade pode ser tão pequena quanto o custo de sincronização.

Para estimar o tempo de uma aplicação escrita no modelo BSP são utilizados os seguintes termos:

- p_i - processadores da máquina $(0 \leq i < p)$;
- w_i^s - custo do cálculo do processador p_i durante a super-etapa s;
- h_i^s - número máximo de palavras recebidas (ou enviadas) pelo processador p_i durante a super-etapa s.

Uma máquina no modelo BSP é capaz de efetuar uma $\lceil \frac{l}{g} \rceil$-relação em cada super-etapa, esta restrição é similar a proposta no modelo LogP. Uma h-relação é uma operação de troca de dados entre processadores, onde cada processador pode enviar e receber no máximo h palavras. Uma estimativa de custo de uma h-relação feita por McColl [McC95] é de hg. Esta estimativa serve como limite inferior, mas para máquinas reais Eisenbiegler, Loẅe e Zimmermann [ELZ98] estimaram o custo como sendo $2hg$. Uma discussão sobre o custo real de uma h-relação pode ser encontrada em [HMS96].

O custo de comunicação de uma super-etapa pode ser estimado por $2h^s g$, onde $h^s = \max_{i=0}^{p-1} h_i^s$. Esta estimativa é boa quando a rede não está congestionada. O custo de uma super-etapa s é menor do que $c = max_{0 \leq i < p} w_i^s + 2h^s g + l$ [ELZ98]. Uma estimativa do tempo total de um programa BSP pode ser obtida somando-se os tempos das super-etapas.

Olhando com mais atenção, podemos ver que o custo de inicialização não é considerado para o envio de mensagens. Entretanto este custo está incluso no custo de sincronização.

Podemos ver os valores dos parâmetros BSP para diversas máquinas na tabela 1.2. Na segunda coluna temos a velocidade do processador em Mflops. g está em flops por palavra, e l em flops. Podemos perceber que o custo de sincronização é grande para todas as máquinas da tabela.

As etapas para obter uma aplicação eficiente no modelo BSP são: balancear a carga de cálculo entre os processadores em cada super-etapa, balancear as comunicações em cada super-etapa (evitar congestão) e finalmente reduzir o número de super-etapas.

Na figura 1.25 temos um esquema de execução no modelo BSP. Na parte à esquerda temos as comunicações explícitas, e à direita a forma normal de se representar uma execução no modelo BSP. No exemplo, o tempo de uma h-relação é 0,5 do tempo de execução de uma tarefa, e o tempo de sincronização é 0,25.

	(Mflops)	g (flop/palavra)	l (flop)
400Mhz PII, 100Mb ether. (8 procs.)	88	30.9	18347
Cray T3D, 150Mhz (256 procs.)	12	2.4	387
Cray T3E, 300Mhz(20 procs.)	47	1.63	880
IBM SP2, 66.7Mhz (8 procs.)	26	11.4	5412
Parsytec (8 procs.)	19.3	25.4	29080

Tabela 1.2: Parâmetros do modelo BSP. Fonte: `www.BSP-Worldwide.org`**.**

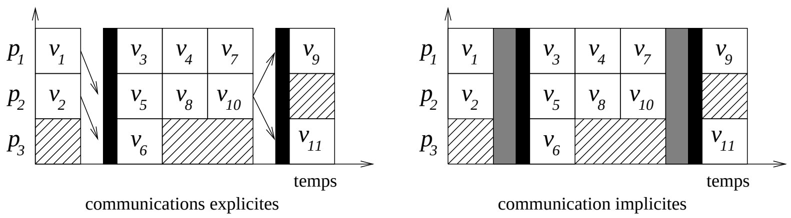

Figura 1.25: Esquema de Execução no modelo BSP.

Um outro modelo muito parecido com o BSP, mas com viés mais teórico é o CGM (*Coarse Grained Multicomputers*) [DFRC93]. Neste modelo simplificado existem apenas dois parâmetros, o número de processadores p, e o tamanho do problema n. Além disto, para que o modelo seja de granularidade grossa, supõe-se que $n/p >> p$. No modelo, também estão definidas algumas restrições como o tamanho da memória local em cada processador, esta é limitada a $O(n/p)$. Além disto, em cada rodada de comunicação é possível apenas efetuar uma h-relação, isto é, em cada rodada, cada processador envia e recebe no máximo $O(n/p)$ dados. Desta forma, o tempo no modelo CGM é dado pelo número de rodadas de comunicação.

É interessante ressaltar que apesar de não considerar diversos detalhes das máquinas, os algoritmos elaborados no modelo CGM, quando implementados apresentam resultados muito próximos aos previstos [Deh99].

Existem algumas bibliotecas específicas de comunicação no modelo BSP. São elas: BSPlib (`www.bsp-worldwide.org`), PUB (`www.uni-paderborn.de/~bsp`) e JBSP [GLC01], uma biblioteca BSP para a linguagem Java. Outra característica interessante dos modelos CGM e BSP é a possibilidade de se elaborar o algoritmo, buscando o melhor desempenho, para em seguida implementá-lo usando outros ambientes de programação, como os de passagem de mensagem (como MPI, ou PVM).

1.3.8.1. Algoritmos

Veremos os mesmos algoritmos vistos para a PRAM, a soma de n elementos e a soma prefixa. Para evitar problemas de arredondamento supõe-se que o número de processadores é múltiplo do tamanho do problema.

O algoritmo 5 tem duas super-etapas, Na primeira, onde as somas parciais são calculadas e enviadas. Na segunda, onde são adicionadas as somas parciais. É interessante observar que o algoritmo é ao mesmo tempo CGM e BSP, pois se $n >> p$ o algoritmo também está no modelo CGM.

Algorithm 5 Soma no modelo BSP/CGM, código do processador i.

Require: Vetor A, sendo que cada processador i contém o sub-vetor $B_i = A(ir : (i+1)(r-1))$ de tamanho $r = n/p$
  $z = B(1)$;
  for $k = 2$ to r **do**
    $z = z + B(k)$;
  end for
  if $i = 0$ **then**
    $S = z$;
  else
    Envie(z, P_0);
  end if
  if $i = 0$ **then**
    for $i = 1$ to $p - 1$ **do**
      receba $z(i)$ de P_i;
    end for
  end if
  sincronização
  if $i = 0$ **then**
    for $i = 1$ to $p - 1$ **do**
      $S = S + z(i)$;
    end for
  end if
  sincronização

O algoritmo 6 calcula a soma prefixa. O vetor inicial é dividido entre os processadores, e cada processador calcula a soma do seu bloco. Em seguida este valor é enviado a todos os processadores, que calculam as somas prefixas. Como saída o algoritmo calcula o vetor de soma prefixa S, sendo que cada processador P_i terá os valores $S(i * r + j), 0 \leq j \leq n/p + 1$.

1.3.9. Tarefas Maleáveis

Tarefas maleáveis [TWY92] (*malleable tasks*) é um modelo onde as próprias tarefas do grafo de precedência podem ser executadas em paralelo. Para considerar o custo de execução adicional, de uma tarefa em vários processadores, uma função de ineficiência $\mu(p)$ é utilizada, onde p é o número de processadores usado. Naturalmente, esta função depende da tarefa em questão. Geralmente, quando maior o número de processadores, maior a ineficiência.

Aplicações grandes, e trivialmente paralelizáveis, têm funções $\mu(p)$ que crescem muito lentamente. Por outro lado, aplicações fortemente acopladas, ou de tamanho pequeno, têm funções de ineficiência que aumentam rapidamente.

A figura 1.26 mostra um exemplo de alocação de tarefas maleáveis. À esquerda as cinco tarefas maleáveis estão alocadas uma por processador. À direita vemos uma alocação onde o fator de ineficiência está visível. Por exemplo, a tarefa alocada à esquerda no processador P_6 é executada em 4 processadores no esquema à direita, o fator de ineficiência é $\mu(4) = 20/18$. Neste exemplo, todas as tarefas são

Algorithm 6 Soma prefixa no modelo BSP/CGM, código do processador i.

Require: Vetor A, sendo que cada processador i contém o sub-vetor $B_i = A(ir : (i+1)(r-1))$ de tamanho $r = n/p$

$s_i = B(1)$;
for $k = 2$ to r **do**
  $s_i = s_i + B(k)$;
end for
difusão de s_i para os outros processadores;
sincronização;
$S(i * r) = s_0 + B(0)$;
for $k = 1$ to $i - 1$ **do**
  $S(i * r) = S(i * r) + s_k$;
end for
for $k = 1$ to $r - 1$ **do**
  $S(i * r + k) = S(i * r + k - 1) + B(k)$;
end for
sincronização;

fracamente acopladas.

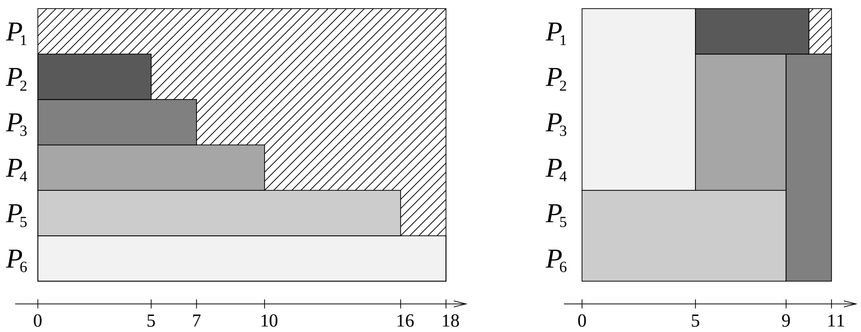

Figura 1.26: Esquema de execução de tarefas maleáveis.

1.3.9.1. Algoritmos

Apesar de ser um modelo razoavelmente recente, já existem vários resultados interessantes para o modelo de tarefas maleáveis. Os principais resultados são:

- Um algoritmo de aproximação com garantia $3/2 + \epsilon$ para o problema de tarefas maleáveis independentes. O algoritmo é baseado no problema de mochila [MRT99];

- Um algoritmo de aproximação com garantia $3 + \sqrt{5}$ para o problema genérico de um grafo de precedência de tarefas maleáveis [LTW].

A apresentação destes resultados é bastante teórica, e foge do escopo deste texto.

1.4. Modelos para Computação em Grade

Para finalizar este texto, apresentaremos este novo tipo de computação paralela, a computação em grade, e faremos uma pequena discussão sobre a aplicabilidade dos modelos vistos.

1.4.1. Motivação

Inicialmente, a computação paralela estava restrita a supercomputadores, extremamente caros. Em seguida, com o aumento do poder computacional dos computadores pessoais, surgiu a idéia de interligá-los, formando aglomerados (*clusters*) de computadores. A idéia de grade vem de duas formas diferentes, interligar vários aglomerados para formar um aglomerado maior, ou aproveitar computadores ociosos presentes em uma rede para formar um aglomerado.

1.4.2. Modelos

A característica predominante em grades de computadores é a heterogeneidade, as vezes de máquina, mas principalmente de rede de comunicação. Logo, não é possível utilizar um modelo que considera todas as características da rede, pois o grau de complexidade é muito grande. A opção válida é o uso de modelos realísticos como o BSP, ou o de tarefas maleáveis. Além disto, existem alguns agravantes, pois em alguns modelos de grade [GKG+01], existe a possibilidade de se forçar a migração de tarefas, sendo que estas podem fazer parte de uma aplicação paralela. Nenhum dos modelos visto considera o custo de eventuais migrações.

Apesar das dificuldades intrínsecas da heterogeneidade, já existem alguns trabalhos interessantes. Em [KKD02], os autores estudaram a aplicabilidade de tarefas maleáveis para aplicações facilmente paralelizáveis. Neste trabalho, foram estudados os efeitos de aumento e da diminuição do número de processadores disponíveis em tempo real, os resultados são bem promissores. Outro trabalho [EHS+02] estuda as vantagens de se executar aplicações em mais de um aglomerado ao mesmo tempo. Mesmo considerando cistos adicionais de até 40% os autores mostraram as vantagens do uso de grades de aglomerados.

1.5. Bibliografia

[ACS90] A. Aggarwal, A.K. Chandra, and M. Snir. Communication complexity of PRAMs. *Theoretical Computer Science*, 71(1):3–28, March 1990.

[AG89] S.G. Akl and G.R. Guenther. Broadcasting with selective reduction. In *Proceedings of the 11th IFIP Congress*, pages 515–520, San Francisco, California, August 1989.

[AISS97] A. Alexandrov, M.F. Ionescu, K.E. Schauser, and C. Scheiman. LogGP: Incorporating long messages into the LogP model for parallel computation. *Journal of Parallel and Distributed Computing*, 44(1):71–79, July 1997.

[BHJ95] D.A. Bader, D.R. Helman, and J. JáJá. Practical parallel algorithms for personalized communication and integer sorting. Technical report, Institute for Advanced Computer Studies, and Department of Electrical Engineering, University of Maryland, November 1995.

[BOS+91] D.P. Bertsekas, C. Ozveren, G.D. Stamoulis, P. Tseng, and J.N. Tsitsiklis. Optimal communication algorithms for hypercubes. *Journal of Parallel and Distributed Computing*, 11:263–275, 1991.

[CC91] J-Y. Colin and P. Chrétienne. CPM scheduling with small interprocessor communication delays. *Operations Research*, 39(3), 1991.

[CCLL95] P. Chrétienne, E.G. Jr Coffman, J.K. Lenstra, and Z. Liu. *Scheduling Theory and its Applications*. John Wiley & Sons Ltd, 1995.

[CDV86] G. Cybenko, Krumme D.W, and K.N. Venkataraman. Gossiping in minimum time. *SIAM Journal on Computing*, 15(1):89–97, 1986.

[CF91] M. Cosnard and A. Ferreira. On the real power of loosely coupled parallel architectures. *Parallel Processing Letters*, 1(2):103–111, December 1991.

[CKP+96] D.E. Culler, R.M. Karp, D. Patterson, A. Sahay, E.E. Santos, K.E. Schauser, R. Subramonian, and T. von Eicken. LogP: A practical model of parallel computation. *Communications of the ACM*, 39(11):78–85, November 1996.

[Deh99] F. Dehne. Coarse grained parallel algorithms. *Special Issue of Algorithmica*, 24(3/4):173–176, 1999.

[DFRC93] 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.

[DI94] B. Di Martino and G. Iannello. Parallelization of non-simultaneous iterative methods for systems of linear equations. In *Parallel Processing: Compar 94 – Vapp VI*, volume 854 of *Lecture Notes in Computer Science*, pages 253–264, Linz, Austria, 1994. Springer.

[EHS+02] C. Ernemann, V. Hamscher, U. Schwiegelshohn, A. Streit, and R.Yahyapour. On Advantages of Grid Computing for Parallel Job Scheduling. In *Proceedings of the 2nd IEEE International Symposium on Cluster Computing and the Grid (CC-GRID 2002)*, 2002.

[ELW97a] J. Eisenbiegler, W. Lowe, and A. Wehrenpfennig. On the optimization by redundancy using an extended logP model. In *International Conference Advances in Parallel and Distributed Computing*, pages 149–155. IEEE Computer Society Press, 1997.

[ELW97b] J. Eisenbiegler, W. Löwe, and A. Wehrenpfennig. On the optimization by redundancy using an extended LogP model. In *International Confe-*

rence on Advances in Parallel and Distributed Computing (APDC'97), pages 149–155. IEEE Computer Society Press, 1997.

[ELZ98] J. Eisenbiegler, W. Loewe, and W. Zimmermann. BSP, LogP, and oblivious programs. In *Europar'98*, number 1470 in lncs, pages 865–874, Southampton, England, September 1998. LNCS 1470, Springer-Verlag.

[Fly66] M.J. Flynn. Very high-speed computing systems. In *Proceedings of the IEEE, 1966*, volume 54, pages 1901–1909, December 1966.

[FW78] S. Fortune and J. Wyllie. Parallelism in random access machiness. In *Proceedings of the 10th ACM Symposium on the Theory of Computing, 1978*, pages 114–118, May 1978.

[GKG+01] A. Goldchleger, F. Kon, A. Goldman, M. Finger, and S.W. Song. Integrade: Rumo a um sistema de computação em grade para aproveitamento de recursos ociosos em máquinas compartilhadas. Technical report, Instituto de Matemática e Estatística da USP, 2001.

[GLC01] Y. Gu, B-S. Lee, and W. Cai. JBSP: A BSP programming library in Java. *Journal of Parallel and Distributed Computing*, 61(8):1126–1142, 2001.

[HCAL89] J-J. Hwang, Y-C. Chow, F.D. Anger, and C-Y. Lee. Scheduling precedence graphs in systems with interprocessor communication times. *SIAM Journal on Computing*, 18(2):244–257, April 1989.

[HMS96] J.M.D. Hill, W.F. McColl, and D.B. Skillircon. Questions and answers about BSP. Technical report PRG-TR-15-96, Oxford University Computing Laboratory, November 1996. to appear on Journal of Scientific Programming.

[KKD02] L.V. Kale, S. Kumar, and J. DeSouza. A malleable-job system for timeshared parallel machines. In *2nd IEEE/ACM International Symposium on Cluster Computing and the Grid (CCGrid'02)*, 2002.

[KKT98] T. Kalinowski, I. Kort, and D. Trystram. Scheduling general task graphs under LogP model using a genetic algorithm. In *International Conference on Parallel Computing in Electrical Engineering (Parelec'98)*, Byalistok, Poland, September 1998.

[KL80] H.T. Kung and C.E. Leiserson. *Systolic Arrays, in: Introduction to VLSI Systems*, chapter 8.3. Addison-Wesley, 1980.

[KR90] R. M. Karp and V. Ramachandran. Parallel algorithms for shared-memory machines. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume A: Algorithms and Complexity, chapter 17, pages 870–941. North Holland, 1990.

[KT99] I. Kort and D. Trystram. Some results on scheduling flat trees in LogP model. *Journal of Information Systems and Operational Research (INFOR)*, 37(1), 1999.

[LTW] R. Lepère, D. Trystram, and G.J. Woeginger. Approximation scheduling for malleable tasks under precedence constraints. *International Journal of Foundation in Computer Science*. to appear.

[LZ95] W. Löwe and W. Zimmermann. Programming data-parallel – executing process parallel. In P. Fritzson and L. Finmo, editors, *Parallel Programming and Applications, 1995*, pages 50–64. IOS Press, 1995.

[McC95] W.F. McColl. Scalable computing. In J. van Leeuwen, editor, *Computer Science Today: Recent Trends and Developments, 1995*, volume 1000 of *LNCS*, pages 46–61. LNCS 1000, Springer-Verlag, 1995.

[MLZ99] M. Middendorf, W. Löwe, and W. Zimmermann. Scheduling inverse trees under the communication model of the LogP-machine. *Theoretical Computer Science*, 215(1–2):137–168, February 1999.

[MRT99] G. Mounié, C. Rapine, and D. Trystram. Efficient approximation algorithms for scheduling malleable tasks. In *Eleventh ACM Symposium on Parallel Algorithms and Architectures (SPAA'99)*, pages 23–32, June 1999.

[PU87] C.H. Papadimitriou and J. Ullman. A communication time tradeoff. *SIAM Journal on Computing*, 16(4):639–646, 1987.

[PY90] C.H. Papadimitriou and M. Yannakakis. Towards an architecture-independent analysis of parallel algorithms. *SIAM Journal on Computing*, 19(2):322–328, April 1990.

[RS87] V.J. Rayward-Smith. UET scheduling with unit interprocessor communication delays. *Discrete Applied Mathematics*, 18:55–71, 1987.

[Tan92] A.S. Tanenbaum. *Modern operating systems*. Prentice Hall, New Jersey, 1992.

[TWY92] J. Turek, J.L. Wolf, and P.S. Yu. Approximate algorithms for scheduling parallelizable tasks. In *Proceedings of the 4th Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 323–332, San Diego, California, June 29–July 1, 1992. Sigact/Sigarch.

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

[Ver97] J. Verriet. Scheduling tree-structured programs in the LogP-model. Technical Report UU-CS-1997-18, Dept. of Computer Science, Utrecht University, 1997.